A DNN with Confidence Measure as an MPC Copycat

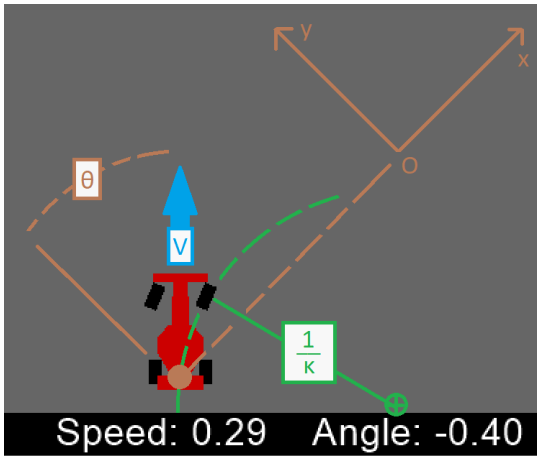
1st Preston Rogers
Dept. of Electrical Engineering
Stanford University
parogers@stanford.edu

2nd Polo Contreras
Dept. of Electrical Engineering
Stanford University
jcontr83@stanford.edu

I. Abstract

The project consisted of the implementation of three parts: the implementation of a Model Predictive Controller to solve a constrained navigation problem, the use of that controller to train a neural network to imitate its behavior, and the conjunction of this network with tools used to classify the reliability of the network output. The aim of the project is to use a neural network to reduce the amount of computation used to solve a stochastic problem, particularly the type of problem which would regularly require multiple iterations for a solution (e.g. non-convex problems), while being able to ensure during deployment that the outputs of the controller are reliable. The results achieved so far have shown that the implemented MPC optimizer can reliably solve the specified problem, that the neural network can learn using information from the solver as training and validation datasets, and that the added tools can estimate a measure of confidence (using out-of-distribution detection) for each prediction so, if there is a low degree of confidence in network predictions for a particular situation, the original solver can be called upon to solve the problem.

II. Introduction



With a wide range of applications, adaptability to perturbations from the environment and an ability to produce reliable results in an autonomous manner, Model Predictive Control is a method for control that has been gaining popularity in a variety of fields, including navigation and dynamic control of robots. However, optimization using MPC suffers from a particular problem: an MPC controller is obliged to generate a complete trajectory when it prepares to take an action, but will follow only a small fraction of that trajectory before it must recompute the solution. This results in a large amount of required computation for each step, which can be problematic when the method is used to control agents that are restricted in terms of computational power or memory, particularly when the specifics of the problem require that the trajectory be computed in real time. Prior work in [1] has demonstrated that this can difficulty can be circumvented in large part by training a neural network to imitate the actions taken by the MPC optimizer, but previous implementations did not have a means for detecting when the network failed to produce reasonable control actions. Using a method previously detailed in [2], we can efficiently detect when a neural network controller is making predictions outside of its trained distribution, and combine both approaches to exploit both the computational efficiency of the neural network and the reliability of MPC.

The project uses an MPC optimizer and neural network to solve a common control problem: trajectory planning for a car moving on a 2D plane, which may or may not require the ability to dodge obstacles in the process. During a data collection phase, the implementation of Model Predictive Control first solves a number of iterations of the problem, and in the process generates data which can be used to form training and validation sets for a neural network. Once the network is trained, the agent can be placed into the environment to solve the same problem, by default using the neural network to do so. A method using matrix sketching allows to efficiently estimate a measure of the characteristics of the network centered around the current input values, in order to detect in real time when an input value is out of the trained distribution. If our estimated confidence in the solution falls below a given threshold, the MPC solver will be recalled to solve the problem, ensuring reliability of the solution (and producing additional data which can be aggregated to future training epochs).

The agent and environment are implemented using Box2D, a physics simulator that operates in 2D space. The MPC takes in information about the current state of the agent and environment and outputs data (i.e. the optimal action given the

current state), which can then be converted into the network input and output. The inputs to the neural network are the differences between the goal location and the current position, as well as information about the current state of the car (i.e. its current velocity, acceleration, turn radius, and derivatives of these quantities), while the output of the network is the optimal action at the current time step. The MPC optimizer to control the car generates an optimal trajectory based on the linearized dynamics of a Reed-Shepp car, able to accelerate forward and in reverse, and able to steer left and right.

This project was developed jointly for two classes: CS230 (Deep Learning) and AA203 (Optimal and Learning-Based Control). The MPC solver used in this project as an oracle for generating training data, and its underlying dynamics, are in the domain of AA203 and were designed in part with techniques covered in that class. The design and implementation of the neural network used to learn to imitate the controller were carried out with methods that were covered extensively in CS230. Finally, although the method used to estimate the reliability of network outputs was not covered in CS230, the underlying principles (such as the consideration for the effects of slightly varying network weights on outputs of the network) are directly related to the fundamental principles of the operation of a deep neural network and an extension of material presented in the class.

III. Related Works

Use of MPC to train a neural network, with the network being designed to mimic the output of an MPC optimizer via policy search, has previously been implemented in [1]. The success of the method detailed therein serves as a clear indication of the capability of neural networks to learn to imitate Model Predictive Control, and the measured improvement in the time taken to compute a subsequent action (from 38ms with MPC to 0.125ms with NN) demonstrates the advantages of the approach. However, the method does not implement a capacity to change between neural network control and Model Predictive Control, leaving the approach potentially vulnerable to reacting in unexpected ways when presented with an unfamiliar input. Meanwhile, a method incorporating matrix sketching for out-of-distribution detection (SCOD) is documented in [2] while an additional method for out-of-distribution detection is documented in [3]. Both approaches are designed to find measures of confidence in a specific output of the network, but in our implementation we favored the former approach as it does not have the same difficulty in scaling to large models or datasets.

IV. Dataset and Features

This project uses pybox2d as a realistic 2D Simulator. In addition, we use a modified version of OpenAI's environment, CarRacing-v0 [5]. These modifications are few but include edits that yield a simplified race-track and allow for the car to go in reverse. Using information about the current state of the environment, a MPC solver using Sequential Convex Programming (SCP) determines an action trajectory that would

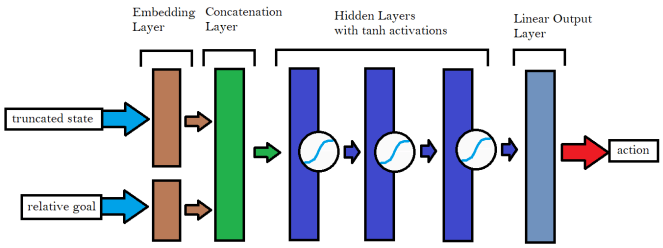


Fig. 1: Structure of Neural Network

get the car close to the end goal by a certain measure elaborated upon in our Methods section. Actions from the trajectory are used until the current state deviates from the state trajectory expected by the SCP solver (which may occur due to dynamics discrepancies or unpredictable noise factors). Once this occurs, the SCP solver is used once more. In regards to the dataset, each executed action is saved into an HDF5 file along with the current state and goal. With this data, we perform a slight modification wherein we do not train the NN using state, goal, action pairs directly but instead train the network using the relative goal pose (shown below) along with the truncated state (state information besides the pose) as input and the chosen action as label.

$$
\begin{aligned}
(x, y, \theta, V, \kappa, a, p)_c, (j, u)_c &\leftarrow \text{Current state, action} \\
(x, y, \theta)_c, (x, y, \theta)_g &\leftarrow \text{Current pose, Goal pose} \\
(x, y, \theta)_c - (x, y, \theta)_g &\equiv \text{Relative goal} \\
(V, \kappa, a, p)_c &\equiv \text{Truncated state} \\
(\text{Relative goal}, \text{Truncated state}), &(\text{action}) \equiv \text{Input, Label}
\end{aligned}
$$

We do not perform extensive feature engineering (besides the construction of the relative goal). However, we do opt to have the NN's first layer be separated into two distinct modules so that the NN can learn an embedding of the relative goal and truncated state, respectively. By separating out these embeddings, we ensure that distinct state information is not misconstrued as being dependent by the learning process. The location of the embedding layers can be viewed in Figure 1.

V. Methods

The MPC optimizer uses an implementation of sequential convex programming, including affine approximations of nonconvex constraints, with Euler approximation used for differential dynamics equations. Obstacles, if they are being considered by the solver, are represented as circles and can be assigned individual radii. The optimization problem is run multiple times, until convergence of the solution. Initial trajectory estimates, for the very first iteration of the convex approximate problem, are repeated instances of the starting state. After the vehicle moves and takes a step, its initial state is updated to include values from its current state, and the rest of the trajectory is initialized by advancing the previous solution of the trajectory by one step (the final state

is repeated once, and is then finally updated by the next iteration of the solution).
After convergence is reached, the MPC optimizer provides a data tuple comprising of a truncated current state (state information minus pose information), difference between the goal and current state, and the recommended action. The data provided from numerous roll-outs is saved into an HDF5 file. In the training stage, this information is used to train a Neural Network that takes as input the relative state and outputs the recommended action. With enough training, it is probable that the Neural Network would recommend actions akin to what the MPC optimizer would have recommended, but with much less computational expense. The loss function used to train the Neural Network is currently a simple ℓ^2-norm difference between the action provided by the MPC optimizer given a state and the action provided by the Neural Network given the same state. The Neural Network structure begins with an embedding layer for the truncated state and relative goal (which are then concatenated), two hidden layers with hyperbolic tangent activation to provide non-linearity, and a final linear hidden unit (output size equal to the action dimension).

The SCP implementation uses the following terms:
x, y, θ: vehicle pose
V: velocity in the forwards direction
κ: curvature (reciprocal of turn radius)
a, p: forwards acceleration (centripetal not included) and pinch (derivative of curvature)
j, u: control inputs, forwards jerk (derivative of acceleration) and juke (derivative of pinch)
$\mathbf{x}_i = (x_i, y_i, \theta_i, V_i, \kappa_i, a_i, p_i)^\intercal$: state vector
$\mathbf{u}_i = (j_i, u_i)^\intercal$: control input vector
N, $h = \Delta t = (t_f - t_0)/N$: number of time steps and time step magnitude
$\mathbf{x}_f$: goal state
λ: scalar hyperparameter to gauge importance of final proximity to goal. In the current implementation, $\lambda = 1$.

The function J that is optimized (excluding constraints involving forward dynamics, ensuring the initial state is equal to the current state, etc.) is shown below. This function is tries to balance, with regularization parameter λ, two competing goals: one being the difference between the final state of the optimal trajectory found and the goal state and the second being the amount of effort put into controlling the car (which is measured using sum of the squares of the ℓ^2-norms of the control inputs, parallelizable via a Frobenius norm).

$$\underset{x}{\text{minimize}} \quad J = \lambda \|\mathbf{x}_N - \mathbf{x}_f\|_1 + h \sum_{i=1}^{N} \|\mathbf{u}_i\|_2^2$$

After Neural Network training is completed, it can be used to make decisions akin to those advised by the MPC oracle. However, with small perturbations and error aggregation, it is possible for the car to go out of the distribution of the training data. To remedy this, we use SCOD [2] that determines if a data point is out of distribution by estimating a measure of how much the output of the network changes given subtle changes of the network weights. This method interprets large variations in the probability distribution of the output due to network weight variations, conditioned on input data points, to be a direct result of the test data-point being out-of-distribution. If a particular input (truncated state, relative goal) is deemed out-of-distribution, the Model Predictive Controller takes control and provides more data for further training of the Neural Network.

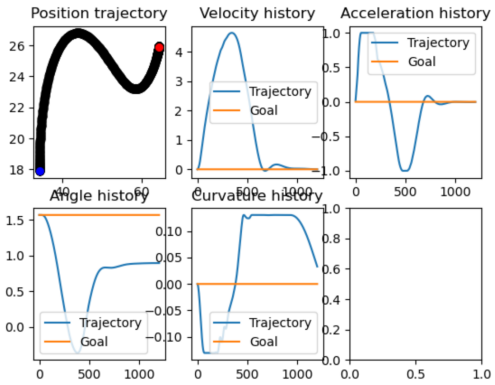
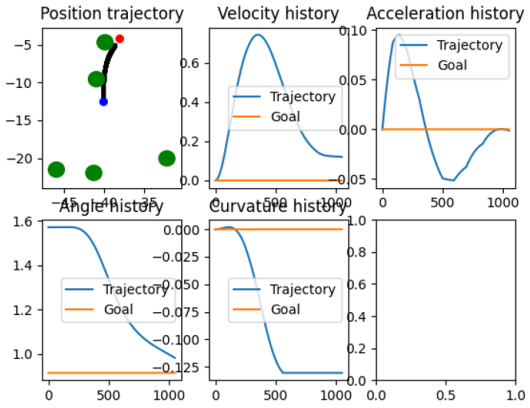


Fig. 2: MPC-guided trajectory: initial position (not inlcuding angle) is highlighted in blue, goal position is highlighted in red

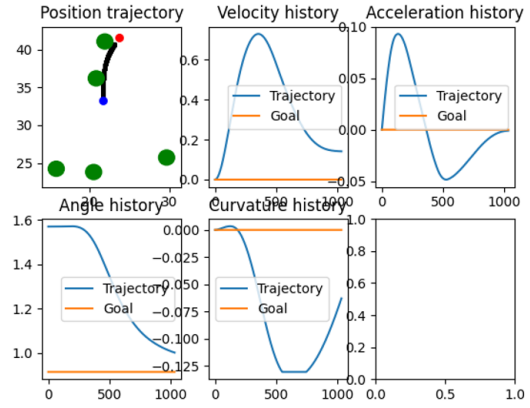
VI. Experiments/Results/Discussion

Our MPC agent is able to navigate from a starting position and orientation to a final position and orientation, and come to a stop. In Figure 2 we provide plots of the agent's actual trajectory over $N_{sim} = 1200$ time steps of simulation.
Using data from one roll-out, we trained a Neural Network to mimic the actions prescribed by the MPC oracle. Figure 3 juxtaposes the trajectory obtained when using the Model Predictive Controller and the trajectory obtained when using the Neural Network trained on the saved data. For various random initializations, the NN behaved akin to the MPC oracle. We found that the Neural Network results in a smoothing affect that is noticeable wen comparing the acceleration history of the MPC trajectory and the acceleration history of the NN trajectory in Figure 3. We believe this behavior is due to the ability of the NN to interpolate between states and the fact that tanh activations are smooth resulting in the learned NN function being smooth as well.

After successfully training a NN to copy the trajectory given by the oracle, we moved onto a different approach where we utilized information from both agents, the MPC and the trained NN, in order to minimize dynamic computation time as well as ensure that the final goal is reached to some degree. To do this, we incorporated the out-of-distribution detection method SCOD to tell us when the vehicle has gone out-of-distribution. We begin the simulation with actions being



(a) MPC trajectory



(b) NN Trajectory

Fig. 3: Juxtaposition between the MPC trajectory and the trajectory produced by the NN post-training (the blue dot, red dot, and green dots correspond to the initial position, goal position, and random obstacle positions, respectively).

provided by the trained NN and once the state is deemed out-of-distribution, a regime wherein the NN would provide unreliable recommended actions, actions are instead deduced by the MPC oracle. We found that SCOD threshold uncertainty values close to 1 worked well for our application. Figure 4 shows a full trajectory that employs both the NN and MPC oracle. In this figure, the bottom right plot makes explicit at what time it was necessary to switch to using the MPC oracle (where the plot switched from 0, denoting the NN, to 1, denoting the MPC oracle).

The code used to generate these trajectories, simulate and navigate through the environment, and which contains the NN we plan to train and run with, can be found at the following location on GitHub: `https://github.com/dtch1997/neural-car.git`

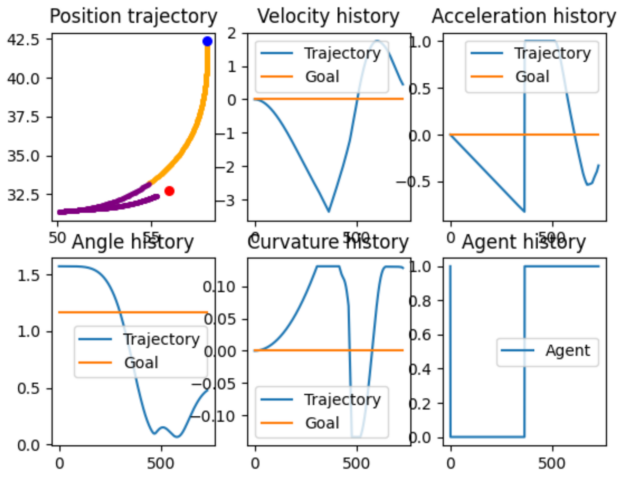


Fig. 4: Trajectory illustrating out-of-distribution detection. In orange is the subset of the trajectory wherein the NN is used, and in purple is the subset of the trajectory wherein MPC is used.

VII. Conclusion/Future Work

Our current solver has been able to generate trajectories and input values that the agent in the simulated field has been able to use to effectively navigate its environment, while the neural network has been able to effectively imitate the behavior of the optimizer when presented with similar problems. Finally, the agent is able to switch between using the neural network and the MPC optimizer for control, depending on the value of the estimate of the metric for out-of-distribution detection. Our next steps are as follows:

1. At present, the MPC optimizer can be called to solve the problem and generate additional training data in the process, but as of yet the neural network needs to be retrained in separate, dedicated training epochs to utilize this information. We intend to set up the neural network so that it can be continually trained by the MPC optimizer when we must switch to it for control, allowing us to gradually aggregate information learned from the optimizer.
2. The current MPC solver is able to efficiently integrate constraints to avoid obstacles in its environment. At present, time constraints have forced us to disable this functionality to simplify neural network training. We intend to find a neural network architecture that can efficiently learn how to avoid obstacles in its environment as well, including the ability to account for non-constant numbers of obstacles in the environment, and pre-processing steps to ease the network's ability to integrate the respective obstacle-related features in the generation of a trajectory.
3. The MPC optimizer developed for this project was created specifically for this application, but the methods detailed in this document can theoretically be applied for any applications that use computationally intensive control

algorithms (e.g. Dynamic Programming, iterative Linear Quadratic Regulation). We intend to implement a similar system on a simulated truss robot, with an existing MPC solver, to generate trajectories and train a neural network for control of a different system, and demonstrate that the method can be applied to existing controllers that use specialized tools.

VIII. Contributions

1) Leopoldo Contreras came up with the original concept for the project, designed the mathematical formulation and the original implementation of the Model Predictive Control solver, and identified Sketching Curvature for Out-of-distribution Detection as a method that could be used for real-time evaluation of the network outputs. He co-designed the architecture of the neural network used to learn controller behavior, particularly with regards to the reformatting of state information when forming the training dataset, and authored multiple sections of this report.
2) Preston Rogers co-designed the network used to learn the features of the controller, in particular the use of hyperbolic tangent functions as activation functions, and the object-oriented implementation of the agent that allows for rapid transition between neural network and MPC based trajectory optimization. He also co-designed the integration of SCOD tools into the existing code base, and authored multiple sections of this report.
3) Daniel Tan, an external collaborator associated with a different class (AA203), co-designed the object-oriented implementation of the agent as well as the integration of SCOD tools into the code base for this project. He was also the principal designer of a method using Linear Quadratic Regulation to augment an existing training dataset with additional state-action pairs, based on states near to those processed by the MPC solver.

References

[1] J. Carius, F. Farshidian, and M. Hutter, "Mpc-net: A first principles guided policy search," 09 2019.
[2] A. Sharma, N. Azizan, and M. Pavone, "Sketching curvature for efficient out-of-distribution detection for deep neural networks," 2 2021.
[3] D. Madras, J. Atwood, and A. D'Amour, "Detecting extrapolation with influence functions," 5 2019.
[4] "NumPy," 2019. Software available from numpy.org.
[5] "Carracing-v0," 2021. Software available from gym.openai.org/envs/CarRacing-v0/.