Learning Dynamical Systems for Model Predictive Control

Kevin Chen, **chen4066@purdue.edu**, Maurice Chiu, **chiu93@purdue.edu**,
Brandon Lee, **lee3008@purdue.edu**, Gloria Ma, **ma571@purdue.edu**

***Abstract*—In this paper, we explore Model Predictive Control methods in simulations of both the Unitree A1 Robot Dog and UR5 Robot arm. More specifically, we use the Cross Entropy Based Method (CEM) along with the Pybullet physics engine to get our A1 and UR5 robot simulations to walk forward and move smoothly to an end position respectively. The results for this part of the project are largely successful as they satisfy our goals laid out thus far. In addition, we experimented using neural networks with ReLU activation layers to learn the dynamics of these systems. In theory, these dynamical models should be able to replace the need for Pybullet while also being more computationally efficient. However, our results indicate that using a "vanilla" neural network to learn these systems may not generate a sufficiently accurate dynamical model, which may in part stem from an inadequate amount of training data, undesirable data, or a sub-optimal neural network structure. More research about graph neural networks (GNNs) is presented as potential future work.**

***Index Terms*—Model Predictive Control, Cross-Entry Method, Neural Network, Machine Learning, Unitree A1, UR5.**

I. Introduction

Robots can be designed and programmed in many different fashions. In simple terms, they all apply a series of actions to perform a desired task. However, using an analytical approach to calculate optimal actions to apply can quickly become unscalable in a complex dynamical system. This is due to the increasing combinatorially complex nature of these calculations with each additional joint/link in the system. A more scalable and practical approach would be using a Model Predictive Control (MPC) framework. MPC is an iterative control process that tries to optimize actions based on predicted future states and a given cost function. This process is both more practical and scalable than an analytical approach. In this project, we explore the MPC framework with simulations of an Unitree A1 robot dog and a UR5 robot arm in a simple 3D environment given an arbitrary set of goals. We also explore using neural networks to learn the dynamics of our system and act as a dynamical model. Most of the work presented in this paper was our attempt to implement what Sanchez-Gonzalez et al. proposed in [1]

II. Proposed Framework

The goals of this project are twofold: (1) use MPC to successfully get the A1 robot to walk forward and the UR5's end effector to reach arbitrary goals, (2) replace the Pybullet engine with learned dynamical models of the robots generated using neural networks. The robots are presented in Figures 1 and 2. For part 1 of our goal, the framework is to implement a Cross-Entropy Based Method (CEM) to generate a series of actions for the A1 and UR5 robot to accomplish specific goals in a simulated Pybullet environment. The environment will be a flat plane with the robots loaded in at the origin. For part 2 of our goal, the framework is to implement two neural network with ReLU activation layers. One neural network will be used to learn the dynamics of the A1 robot and the other will be used to learn the dynamics of the UR5 robot. Training data will be generated from multiple runs using the CEM algorithm established in part 1. These learned dynamics models will then replace Pybullet to predict the next state of a robot given a current state and action.

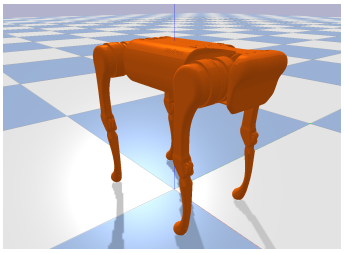

Figure 1. The A1 Robot Dog at its Starting Pose

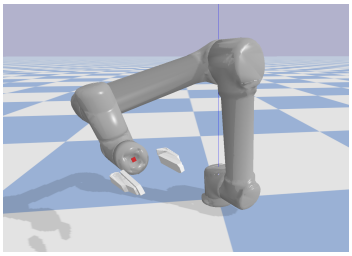

Figure 2. The UR5 Robot Arm at its Starting Pose

III. Implementation Details

A. Model Predictive Control

Our implementation of MPC planning uses the CEM algorithm, which is known to be the more practical approach used in modern day experiments because it considers the dynamical system as a black box. This method inputs the initial state of a robot S_0 and some action A_0 through the dynamical system to output the robot's next state S_1. Now, we will introduce terms that are involved in this method. A *plan*, $g \in G$, consists of a set of predicted actions that our system will generate from a normal distribution ($N(\mu, \sigma)$) to perform on a robot. The

samples generated from the distributions are limited to the range of forces that each joint can produce. Note, we only consider the robots' *revolute* and *prismatic* joints–not fixed joints. The length of the set of predicted actions is called the *horizon length*, H. The idea is that for each environment step, $n \in N$, for the robots, we want to generate G plans, each of length H, and calculate the total cost of each plan w.r.t a defined cost function. This cost function measures how well an action, or set of actions, is for the robot to execute. Then, we choose the top K plans according to our cost function, generate a new mean and standard deviation from these plans to update our distribution ($N'(\mu', \sigma')$), and generate a new set of G plans from the derived distribution for the next iteration. After a set number of training iterations, T, we will choose the single best action and execute it on the robot. This process is iterated for N environment steps.

It is worth noting that in order to avoid getting stuck in a local minimum during the process of finding a more suitable mean and standard deviation for our normal distributions, two approaches can be taken. The first is to tune our K value so that we include just the right amount of original samples. Too little or too many original samples could skew the derived mean and standard deviation values. The second is to add a bit of *noise* to the newly computed standard deviation. This is to add stochasticity to our data and prevent the model from adhering too closely to our samples.

For the work spaces, we gave our environment Earth-like gravity and loaded our robot into the world on a horizontal plane centered at the origin with no obstacles. We utilized libraries such as PyTorch and NumPy to handle complex matrix calculations and PyBullet to simulate the Unitree A1 and UR5 robots on a set of actions.

Finally, we will differentiate the finer details between the two simulated robots below:

1) Unitree A1 Robot Dog: The Unitree A1 robot is constructed with 18 joints: the floating base (1), the IMU joint (1), hip joints (4), thigh joints (4), calf joints (4), and the feet (4). Our naive implementation only considers 16 of these joints, which include all the joints related to the robot dog's lower limbs (i.e. hip, thigh, calf, foot). A full list of the active joints and their ranges can be found in **Appendix A**. These are the approximate parameter ranges we used for the CEM algorithm:

1) N: 100
2) G: 10-100
3) H: 5-80
4) T: 5-25
5) K: 20-40%

Throughout our experiments, we tested several cost functions to urge the A1 robot to walk forward. Initial problems ranged from the robot flinging itself forward or flying airborne to the robot sitting down and awkwardly falling onto its back. The final cost function we chose took into account the robot's center of gravity, height, and distance from goal. To approximate center of gravity, we looked at the robot's pitch, roll, and yaw (which we represent below as P, R, and Y respectively). It should be noted that the angles for pitch, roll, and yaw were not calculated because it would be an unnecessary computational cost. Instead, pitch was represented as the difference in height between the front and back of the robot, roll was represented as the difference in height between the right and left side of the robot, and yaw was represented as the difference in the y direction between the front and back of the robot. Note that yaw was calculated in this way because the robot's goal was to walk in a straight line along the x-axis after being loaded in at the origin. These three values roughly represent the center of gravity of the robot because the pitch penalized any tilting forward or back, the roll penalized any tilting side to side, and the yaw penalized any deviation from a straight path. In addition, the height from the ground to the center of robot was recorded when the robot was loaded in. Any change from this initial height (represented below as h) was incorporated into the cost function to penalize the robot from sitting down, jumping, or falling over. Finally, the distance from some arbitrary goal on the x-axis (represented below as D) was included in the cost function to penalize the robot for not walking forward.

Cost function:

$$C_{\text{A1}} = 2000P + 300R + 300Y + 3000h + 2D$$

In addition to the cost function above, to further penalize the robot to not jump, if $h > 0.25$, 1000 would be added to the cost. Moreover, if D at the beginning of an episode is less than or equal to D at the end of that episode, 10000 is added to the episode's cost. This severely penalizes the robot for not moving forward.

The coefficients for the above cost function and the additional checks and costs were all tuned with trial and error.

2) UR5 Robotic Arm: The UR5 robotic arm consists of 8 active joints and 2 fixed joints which are all connected in series. A full list of the active joints and their ranges can be found in **Appendix B**. The parameters for the CEM for UR5 we've chosen are the following:

1) N: 3
2) G: 1200
3) H: 1
4) T: 20
5) K: 15%

The goal for UR5, after being loaded into the environment and moved into a random initial pose, is to have its end-effector reach some arbitrary goals. Originally, we tried to control the UR5 through torque control, however, with much effort, it was too tricky to make work. We then moved on to do position control and that the robot arm can actually follow a trajectory.

Although it seems like what we had was working, we are experiencing a few issues at this stage. First, even though

the end-effector was following the trajectory points in **this video**[10], the arm is switching between elbow-up and elbow-down configurations. This is unwanted behavior because we wanted the motion to be smooth and cost efficient. Applying torques to the joints have energy cost and waiting for the arm to move into place have time cost which negatively impact the performance. Second, the end-effector was not very accurate in terms of being spot on at following the points along the trajectory as can be seen in Figure 3. Third, the robot arm sometimes move backwards before advancing further until reaching its destination. This behavior could be observed in **this video**[11]. Fourth, the revolute joints of the UR5 have position ranges from $-\pi$ to π, our current implementation does not check whether a configuration of the UR5 is collision free with it self. Some of the positions that we sampled will make the links run into each other and get stuck. Lastly, the robot arm sometimes grazes the plane which it was based on.

Every issue except for the fourth issue described above were address by our final version which we are going to explain below how they were resolved. The first and last issue was addressed with an updated cost function. The second and third issue was later resolved by allowing the robot enough time steps to carry out its actions and by greatly increasing the number of plans (G) we sample. The fourth issue regarding self collision would require collision detection to be implemented which we plan to do in the future. In designing our cost function for the UR5, we took into consideration the distance cost d which is the distance between the end-effector and the goal, elbow cost e which is the distance between the elbow joint's position in the previous and current state, ground-collision cost $g = \Sigma_{j_z \in Joints} \mathbb{1}_{(j_z < 0.15)}$ where j_z stands for the z coordinate of an active joint. Additionally, we keep track of the remaining distance to goal in each iteration. Another cost of 10 would be imposed if d is greater than the current distance to goal. This ensures the best effort for the arm to not move further away from its goal. The cost function, after applying weights, is as follows:

Cost function:

$$C_{\text{UR5}} = 10d + 1e + 2g + 10 \cdot \mathbb{1}_{(d > \text{distToGoal})}$$

With this cost function, the unwanted elbow movement was penalized by the elbow cost and that any configuration that causes any joints to dip below a certain height is penalized by the ground-collision cost. The distance cost ensures that the end-effector is moving towards the target.

We found that, in previous versions, we were not giving the robots enough time steps to carry out its entire movements. The effect of not giving enough time steps and the use of the MPC-CEM algorithm does not mix well. In order for the algorithm to pick out the best action, it depends on the end states of the robot at the end of every action. Without letting the actions to be carried out fully, the algorithm would still move on to the next iteration resulting in inaccurate end-effector positions. We then updated our action-applying function to check that if every joint has arrived at their assigned position or if a maximum iteration has been reached, which ever comes first, before allowing the algorithm to move on. The backwards movement of the arm, we believe, was due to not enough sampled plans. With not enough samples, the best action determined by the algorithm could move the arm's end-effector past the goal point. If it is overshooting by more than one way points away, it will move backward towards a point that was overshot. By making the algorithm generate 1200 plans per each training steps (T), a configuration that gets the end-effector close enough to the goal position can usually be sampled in the first iteration and the algorithm would terminate early for having achieved its goal.

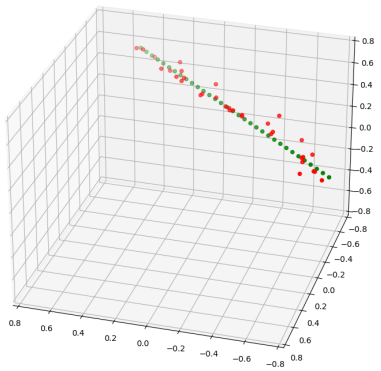

Figure 3. UR5's End-Effector's Trajectory (Red) Trajectory to Follow (Green)

B. Neural Network

The neural net implementation consisted of 3 phases: (1) collecting data, (2) training the neural network, and (3) using the newly trained model in place of Pybullet to run our CEM implementation.

1) Data Collection:

a) Unitree A1 Robot Dog: The data we needed to collect were tuples in the form of (S_1, A_1, S_2), where S_1 is the initial state of the robot, A_1 is the action applied to the robot, and S_2 is the state after A_1 is applied to the robot in state S_1. We also needed to collect both "good" and "bad" data. Here, we classify "good" data as data from a run where the robot successfully walks forward and does not fall over and "bad" data as data from a run where the robot falls over or does not walk forward at all. Note, it is important to collect data of all sorts, such that the neural network is able to learn the "good" dynamics of the model and steer away from the "bad" ones. In other words, the neural network should be able to distinguish between effective results where the robot is upright and walking and produce less of the states that make it fall. However, since we also wanted it to mostly favor the robot in walking states, it was decided that the majority of our data should be "good" data. We employed 2 methods to collect this data.

The first method was to collect "good" from the final runs generated by the CEM algorithm. In other words, we saved the final (S_1, A_1, S_2) tuples generated from running the CEM

algorithm. We got the "bad" data from running the A1 robot dog on a series of random actions at different locations (which would likely produce a sequences of undesirable tuples that represented the robot falling over or not moving forward). This first method proved to be costly both in computation and time. This is because 1 run of CEM with parameters N, G, H, T, and K, there will only be N data points generated. When generating data, we usually set $N = 300$ to generate 300 data points per run. We wanted to have at least 100,000 "good" data points which would have taken weeks to generate with this method. We tried using free resources to run paths, but a lot of our data-generation jobs got killed or did not finish. To save computational resources and time, we devised another way of generating the data.

The second method of collecting data involved looking at the top 2 paths of each iteration in the CEM algorithm. When running the CEM algorithm, the only time we apply a "final action" is once per iteration when we grab the first action from the best episode/rollout. When we inspected what this "best episode" looked like, we discovered that it was almost always a "good" run. In fact, the top 2 episodes were almost always a "good" run where the robot walked forward and did not fall over. To make data generation more efficient, we just saved the (S_1, A_1, S_2) tuples from these top 2 runs during every iteration of one run of the CEM algorithm. This means that if we run the CEM algorithm with parameters N, G, H, T, and K, instead of getting N data points, we'd be getting $N * 2 * H$ data points. Choosing $N = 150$ and $H = 90$ and running the CEM algorithm 6 times, we got 162,000 data points this way. Note that a majority of these points are "good" data, but there are a couple runs of "bad" data here too.

b) UR5 Robotic Arm: For the UR5, the neural net version of the CEM algorithm has been through two versions. The training data of the first version, the one we presented in our presentation, came from all of the best episodes produced by our CEM algorithm. We used a script to run through the trajectory and the actual end-effector positions recorded for each of the best episodes and calculate the MSE to determine how good an episode is. 74 from a total of 112 episodes, that is about 66.07% of the episodes, are good.

Since we heavily modified the CEM algorithm for the UR5, we had to modify the neural net version as well as generate new training data. We determine the bad data from the good data by looking at the length of the episodes generated by the CEM algorithm. We mark a piece of data as bad if the CEM algorithm terminated at the maximum iteration and a piece of data as good if the CEM algorithm can terminate early. We determine it this way because, after we've modified the CEM algorithm, it is most likely that the end-effector could move to the goal position in one action. Based on our experience, if it takes more than 2 iterations, there is a very high chance that it may never terminate for a self collision might have caused the arm to get stuck or that the goal generated was, for some reason, unreachable. Out of 221 episodes, 165 of them are good. That is about 78.20% of the episodes are good.

2) Training the Neural Network: We opted to use three hidden, 256-node layers with ReLU as the activation function in each layer. The input to the neural network is the state of the robot and an action to be applied, and the output is the state the robot is in after the action is applied. In this training step, we also normalized all the states and actions with respect to the minimums and maximums of our dataset. This normalization factor also helps keep the values of the states and actions closer together to prevent confusing the neural network. It should be noted that we trained the neural network with two different versions of state components. The first version was only using the Cartesian coordinates of the hips and center of the robot. The second version also included the Cartesian coordinates of the thigh, calf, and feet joints.

3) Using the Trained Model for MPC: With our newly trained model, we utilized the same CEM algorithm as before, but this time we input a state and action to the model to retrieve the next state instead of applying the action to the robot in the environment and retrieving the state information afterwards. In addition, we added a normalization factor to the input and the reverse to the output for the same reasons as adding it during the training of the model.

IV. Results

A. Unitree A1 Robot Dog

For simplicity, the main goal of the robot was to walk forward along the x-axis from the origin point that it was loaded into the environment. As a basis, we set the Cartesian coordinates for the goal point to (10,0,0).

1) MPC: As mentioned before, our initial results with a naive implementation of the cost function shows the robot flinging forward and **flying airborne**[1] or **falling down**[2]. In addition, we attempted to stitch together relevantly "good" action sets where one of the robot leg's gait movement completes a full step and apply that repetitively to all legs. Theoretically, the idea sounded promising, but the result **here**[3] did not turn out well either.

After incorporating the pitch, yaw, and roll components into our cost function, we started to see an improvement to the movement of the robot shown **here**[4]. We can observe the algorithm calculating actions where the robot does not fall down, but instead, trying to balance on its front two legs. After some more fine-tuning with heavy penalization to the pitch component and the distance from the goal point, the robot finally managed to move forward without falling down shown in this **video**[5].

2) MPC with Neural Network: We trained the model through the neural network on both **good**[6] and **bad paths**[7] by the robot.

For the model, we experimented with the amount of hidden layers as well as the number of epochs it was trained on. In Figure 4, we show the average MSE between the training and testing data over 50 epochs. The average MSE was not high, but it was not convincingly low and did not converge. As a result, the resulting MPC runs with the neural network did not produce great results. An example is provided **here**[8].

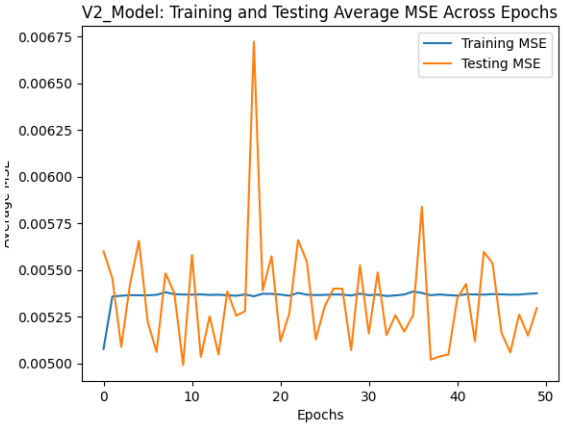

Figure 4. A1 Model 1's Average MSE between Training and Testing Data Across 50 Epochs

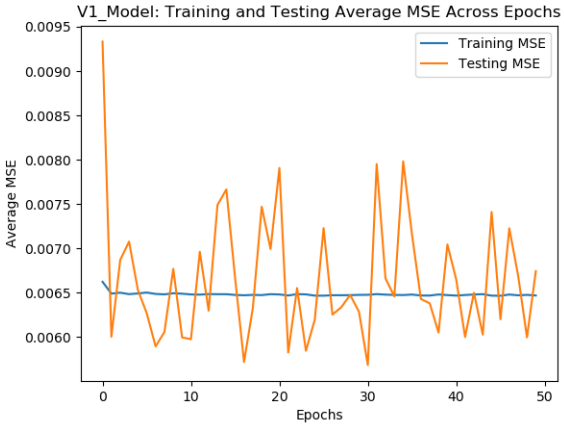

Figure 6. A1 Model 2's Average MSE between Training and Testing Data Across 50 Epochs

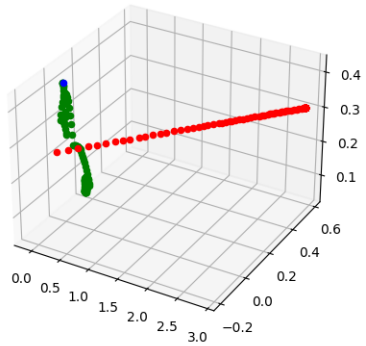

Figure 5. A1 Model's Predicted State (red) vs Actual State (green) of Body Position

For further evaluation, we compared the Cartesian coordinates of the center position of the robot's body between the model's predicted state and the actual state. In Figure 5, we can see that these positions differ significantly. While the model thinks the body position is somewhat stable after applying each action, the robot's body position actually falls down, which correlates with the video presented.

Some further adjustments we made include adding to our state vector input and normalizing them (and the reverse for the output). For the previous model, we included the hip and body coordinates. We believe the neural network needed to learn more information about the robot to make more accurate state predictions. As a result, we added the thigh, calf, and foot coordinates to the state vector. Unfortunately, after adding normalization and modifying the state vector, the results did not improve. The average MSE plot in Figure 6 still did not converge and the run was similar as before shown **here**[9]. Discussions about revising our implementation with graph neural networks are below.

B. UR5 Robotic Arm

1) MPC: The best episodes produced by our final version of the MPC-CEM algorithm was smooth, cost efficient, and collision free with the ground as can be seen in **this video**[12] and **this video**[13]. This is the version we used to generate the final training data.

2) MPC with Neural Network: Here, we present the outcomes of the two versions which we have worked on. **This video**[14] shows the first version of our neural net MPC-CEM algorithm. It was moving toward the goal but proceeded to overshoot the goal. **This video**[15] is the second version. For some reason that we have yet to understand, it is not really performing in the way we predicted which we shall further investigate.

Despite the unsuccessful outcomes, we have verified the benefit of replacing PyBullet with a neural network with a learned dynamical model of the robot. Generating 1 action by calling the dynamical model using PyBullet with our MPC-CEM algorithm takes on average 26 minutes to complete where as the nerual net version only takes 10 seconds on a MacBookPro with 2.9 Ghz 6-Core Intel Core i9 processor, 32 GB 2400 MHz DDR4 memory, and Radeon Pro Vega 20 4GB graphics card which is a 15600% improvement.

Links to all videos of the robots in action during training and working can be found in **Appendix C**.

V. Discussions & Conclusion

It seems apparent that our neural networks were unable to consistently and/or effectively predict well-behaved states. Moreover, it is also possible that our operations were not completely flawless. However, there's merit in revising parts of our implementation to consider graph neural networks [1]. With representing each link as edges and the ends as nodes, the data points would be able to learn hidden relationships within each edge and build a more wholly representation of the robot's state. In addition, we could extend it even further

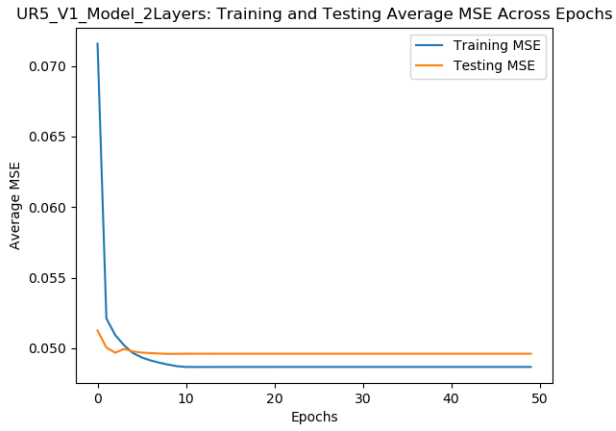

Figure 7. UR5 Model's Average MSE between Training and Testing Data Across 50 Epochs

and train one neural network by feeding both the robot's state information as input and get a generalized model.

This work walks through a motion planning framework implemented to generate a sequence of actions that get the Unitree A1 and UR5 robot to follow a simple, arbitrary trajectory. The framework consists of using the Cross Entropy-based Method (CEM) on a learned dynamics model. More work can be extended to include graph neural networks in the future.

VI. Statement of Contributions

Kevin: Most of my time was spent working on the implementation related to the A1 robot. I helped implement the first versions of the MPC algorithm. Afterwards, I spent an arduous amount of time tweaking different parts of the implementation bit by bit, including fine-tuning the cost function and switching the design from torque control to position control. As the Milestone 3 deadline drew closer, I shifted my focus on the implementation of the neural network. I spent a lot of effort reading over the Sanchez-Gonzalez paper numerous times and researching further about the design of Graph Neural Networks (GNNs). Shortly after, I helped devise the framework of the NN-MPC version without GNNs for both the A1 and UR5. On the side, I also tried to generate as much data within the timeframe we had by utilizing the GPUs on the scholar computers and researching other platforms that had free GPUs. After the presentation, I added to the state vector input, trained a new model to run with MPC, but the results were still ineffective. Unfortunately, we did not have time to implement GNNs completely, which may have made the difference.

Maurice: I mainly contributed to the UR5 implementation. In the beginning stages of our project, having Brandon's first implementation of A1's MPC-CEM algorithm and Gloria's first adapted version of the MPC-CEM algorithm for the UR5, I started to learn the intricacies of PyBullet especially around friction, joint resistance, time step (a PyBullet simulation parameter), minimum required forces for each joints of both robots, and ways to quickly adjust the mean and covariance required for the multivariate normal distribution which we sample our actions from and shared my findings with the team. In the mid-stages, I kept working on the UR5 implementation which includes rewriting the MPC-CEM algorithm multiple times (first version was adapted from Brandon's latest implementation) and tested multiple versions of the cost function for the UR5. During this stage, I have also worked on ways to measure the performance of the UR5 by writing multiple helper programs to generate graphs and to playback paths generated from our algorithms. In the final stages, I finalized the final version of the MPC-CEM algorithm for the UR5 and the neural net version of the MPC-CEM algorithm for the UR5 (framework of the NN-MPC code was provided by Kevin and I added codes specific to the UR5). Lastly, I generated the training data using the final version of our MPC-CEM algorithm and trained and tested the final version of the NN-MPC algorithm. During the entire semester, I tried my best to understand and keep up with all the updates regarding changes made to the A1 implementation.

Brandon: I mainly contributed to the A1 implementation. This included writing multiple versions of the CEM algorithm (including both torque and position control versions as well as the final version), testing and fine-tuning many cost functions, devising and implementing methods of collecting training data for the Neural Net, training the Neural Net multiple times, and incorporating the learned model into the CEM algorithm. This process included numerous late nights of testing and researching various avenues of implementation (a majority of which are not included in the final submission). I also worked on several different methods of saving and retrieving relevant data for later inspection/debugging and the playback capabilities of the A1 robot dog. This included a 3d trajectory grapher to see where the learned NN model thought the A1 robot was moving vs how it was actually moving. A lot of these efforts on the A1 robot dog, especially with the Neural Net, were done in conjunction with Kevin Chen. For the UR5, my contributions were limited to some early work on implementation, general guidelines and templates for implementation of CEM and Neural Network, suggestions in group discussions, and keeping up with current updates. Near the second half of this semester, I also took some responsibility of planning and assigning work for the team as well as establishing a rough timeline.

Gloria: I adapted Brandon's implementations of A1's MPC-CEM into prototypes for UR5 MPC, and worked on refining it with Maurice. For A1, I documented the runs, experimented with alternative methods for generating working paths when we're stuck due to cost-functions. I explored multiple computational resources and methods for collecting and recording data. I made general suggestions in group discussions, and tried to keep up with the updates.

References

[1] A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, J. Merel, M. Riedmiller, R. Hadsell, and P. Battaglia, "Graph networks as learnable physics

engines for inference and control," *arXiv e-prints*, p. arXiv:1806.01242, Jun. 2018.

APPENDIX A
UNITREE A1 ACTIVE JOINTS AND RANGES

The following information are Joint Number; Name; Type; Joint Lower Limit; Joint Upper Limit respectively.

- 0; Floating Base; Fixed; 0; 0
- 1; IMU Joint; Fixed; 0; 0
- 2; FR Hip Joint; Revolute; -0.802851455917; 0.802851455917
- 3; FR Thigh Joint; Continuous; -1.0471975512; 4.18879020479
- 4; FR Calf Joint; Revolute; -2.69653369433; -0.916297857297
- 5; FR Foot Fixed; Fixed; 0; 0
- 6; FL Hip Joint; Revolute; -0.802851455917; 0.802851455917
- 7; FL Thigh Joint; Continuous; -1.0471975512; 4.18879020479
- 8; FL Calf Joint; Revolute; -2.69653369433; -0.916297857297
- 9; FL Foot Fixed; Fixed; 0; 0
- 10; RR Hip Joint; Revolute; -0.802851455917; 0.802851455917
- 11; RR Thigh Joint; Continuous; -1.0471975512; 4.18879020479
- 12; RR Calf Joint; Revolute; -2.69653369433; -0.916297857297
- 13; RR Foot Fixed; Fixed; 0; 0
- 14; RL Hip Joint; Revolute; -0.802851455917; 0.802851455917
- 15; RL Thigh Joint; Continuous; -1.0471975512; 4.18879020479
- 16; RL Calf Joint; Revolute; -2.69653369433; -0.916297857297
- 17; RL Foot Fixed; Fixed; 0; 0

APPENDIX B
UR5 ACTIVE JOINTS AND RANGES

The following information are Joint Number; Name; Type; Joint Lower Limit; Joint Upper Limit respectively.

- 1; Shoulder Pan Joint; Revolute; $-\pi$; π
- 2; Shoulder Lift Joint; Revolute; $-\pi$; π
- 3; Elbow Joint; Revolute; $-\pi$; π
- 4; Wrist Joint 1; Revolute; $-\pi$; π
- 5; Wrist Joint 2; Revolute; $-\pi$; π
- 6; Wrist Joint 3; Revolute; $-\pi$; π
- 8; Left Gripper Motor; Prismatic; 0; 0.04
- 9; Right Gripper Motor; Prismatic; -0.04; 0

APPENDIX C
VIDEO LINKS

1) **A1 Flying Dog: https://drive.google.com/file/d/1-7qbWTJFzScGOMsGvvFXM3IMC6kNv6UA/view?usp=sharing**
2) **A1 Sitting Dog: https://drive.google.com/file/d/19O9IRGyU_ZZutKZfJrZ4lFZ3mQGRB4oo/view?usp=sharing**
3) **A1 Stitched Set: https://drive.google.com/file/d/13iTYi4Odk-W0_-RmHlLRVpg3jkVfANnL/view?usp=sharing**
4) **A1 MPC Backwards Run: https://drive.google.com/file/d/1ZUN80e2i5I9Wk2suykIB0ZWewxxP714E/view?usp=sharing**
5) **A1 MPC Best Run: https://drive.google.com/file/d/10-ciuAwrGPFZQvH824suqKURa_BHcAOz/view?usp=sharing**
6) **A1 MPC Ground Truth Good Path Data: https://drive.google.com/file/d/1nCytfdjDxuheNHbSlkHJnoWD3hCXyqcu/view?usp=sharing**
7) **A1 MPC Ground Truth Bad Path Data: https://drive.google.com/file/d/1n33Xcs2gWrMSXuBdvPbcsEQkhgT0nXpI/view?usp=sharing**
8) **A1 NNMPC Model 1 Run: https://drive.google.com/file/d/1q7DsLOGiwnvpAfLeDUKaWtE6TxAC2KNY/view?usp=sharing**
9) **A1 NNMPC Model 2 Run: https://drive.google.com/file/d/1i8nq8gq12nYJ7w2jnNAPAjX43SCYeNfs/view?usp=sharing**
10) **UR5 Bad Efficiency: https://drive.google.com/file/d/1uX_HahzFvqrlN6scI52-b1SDJG8YTUs6/view?usp=sharing**
11) **UR5 Backward Movements: https://drive.google.com/file/d/1dY8-y-IZW83Ui9O-thAwedpK6ylxMIBZ/view?usp=sharing**
12) **UR5 Smooth Movement: https://drive.google.com/file/d/1GDlNoGZCigsCsRdCXy5KY41qQsTbQ7JO/view?usp=sharing**
13) **UR5 No Grazing: https://drive.google.com/file/d/1Km4_NbsV5LcPHgHaenI80XV9zaFPE0rj/view?usp=sharing**
14) **UR5 NNMPC Version 1: https://drive.google.com/file/d/1dpnKk5dnjMjJ3CRta9Tg76HZ1Zle7--s/view?usp=sharing**
15) **UR5 NNMPC Version 2: https://drive.google.com/file/d/1Sbguht2WScVuKh9iSO689iucTd1AKh2X/view?usp=sharing**