Model Learning and MPPI
Due date: Never

1 Introduction

In this homework, we revisit the concept of local control for robot navigation, as well as integrate our local controller into a global planner. This document covers the former, and more information about the latter will be released soon. In the previous homework, we saw that we could perform feed-forward control by mapping controls to templates and then matching these templates to the robot's observations at runtime. Two of the fundamental weaknesses of that implementation are that it made strong assumptions about the motion of the car, and the robot is limited to navigating paths that fit its templates well. In this assignment, we will tackle the first issue by learning a motion model from data generated by the car itself, and handle the second issue by replacing template matching with Model Predictive Path Integral control, a method that iteratively refines the robot's trajectory by rolling out a large number of possible trajectories and weighting them by their cost.

Skeleton code can be found here, although you are under no obligation to use it for this assignment.

2 Model Learning

We wish to learn the forward motion model for our robotic car: $state_{t+1} = f(state_t, controls_t)$. You have previously seen two such motion models: the odometry and kinematic models. You have also seen that these models may not capture the dynamics of the system very well. Previously, we added noise to these models to account for unmodeled dynamics. Here, we will instead learn a deterministic model (i.e. we will not add noise to the output of the model) from data of the car driving around. We suggest that you implement your model as a neural network that predicts the change in the robot's state from the applied controls and its previous change in state. For example, to predict the robot's state x_{t+1} with the neural network f:

$$[\Delta x_{t+1}, \Delta y_{t+1}, \Delta\theta_{t+1}] = f([\Delta x_t, \Delta y_t, \Delta\theta_t, cos(\theta_t), sin(\theta_t), v_t, \delta_t, dt_t])$$

$$[x_{t+1}, y_{t+1}, \theta_{t+1}] = [\Delta x_{t+1}, \Delta y_{t+1}, \Delta\theta_{t+1}] + [x_t, y_t, \theta_t]$$

Note that the Δ terms attempt to capture the velocity of the robot. In addition to providing these terms to the input of the network, we also include the robot's rotation θ because this value constrains the freedom with which the robot can move in the x and y directions. Instead of giving the network the raw θ value, we provide it $cos(\theta)$ and $sin(\theta)$ to avoid wrap around issues. Most of the code you've already written in this course has assumed $-\pi \leq \theta \leq \pi$; this is an assumption your neural net code, and your MPPI controller code will have to specifically handle. Also in the above, we have $dt_t = Time_t - Time_{t-1}$. You may find that the calculation of $\Delta x_t = x_t - x_{t-1}$ to be sufficient, instead of dividing by dt_t. Given that we will be calculating these values from an assumed correct pose provided by a particle filter running at 10hz, the average dt is 100ms.

2.1 PyTorch

To get PyTorch on your cars, you will have to download the PyTorch files from Canvas, and extract them to the `/usr/local/lib/python2.7/dist-packages` *directory. There will be a* `/dist-packages/torch` *and* `/dist-packages/torch-0.4.0a0+28f056f-py2.7.egg-info` *directories after doing this.*

PyTorch is a machine learning framework built around two tools: Tensors and Autograd. Tensors are functionally equivalent to numpy's ND-arrays: they are n-dimensional structures that contain data that can be manipulated arithmetically. Most operations that you can perform on a numpy array can be performed on a PyTorch tensor, with the added benefit that these operations can be run on your robot car's GPU for a big increase in performance.

Autograd is PyTorch's automatic differentiation toolbox. Auto-diff is the computational tool that can, with little overhead, find derivatives from functions you've specified in code. This is different than symbolic differentiation ($dx^2 = 2x$) in that it never discovers the derivative function ($2x$) explicitly, instead relying on the forward computation of x^2 to pre-calculate the result of the backwards calculation $2x$ for a given value of x. What this means is that for a given function you define, you can quickly calculate derivatives with respect to variables you care about. In this homework, you will create a neural network of parameters that gets called in a function to perform a task: act as a motion model. You need to differentiate the function w.r.t the neural network parameters in order to improve your neural network. The error in your neural network's performance as a motion model is used to change the parameters; changes in the parameters result in changes in the performance, and we want to increase performance.

2.2 Data Collection

We have provided a bag file containing 'ground truth' positions of the robot. These positions were generated using a noisy particle filter, so you may find that your model improves if you first apply some filtering to the data. In particular, *scipy.signal.savgol_filter* may be useful - Fig. 1 illustrates its effect.

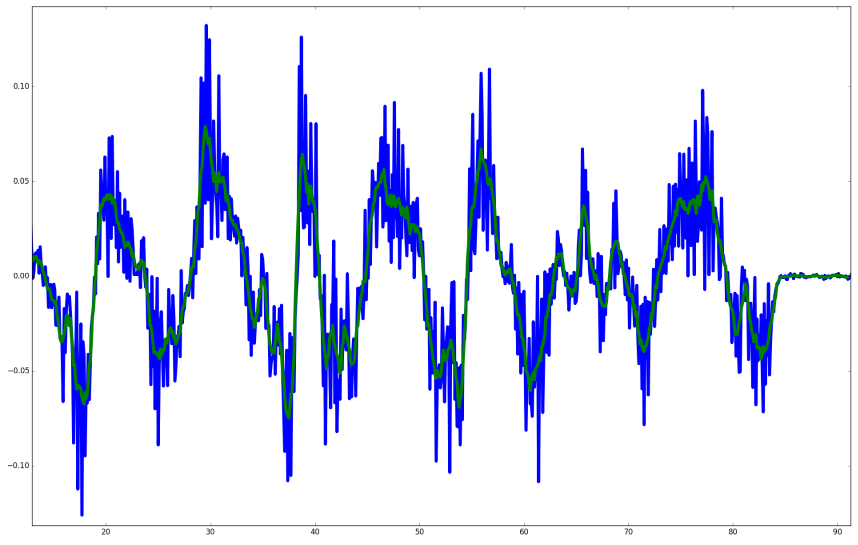

Figure 1: Your raw values may need to be filtered and smoothed. This plot is of Δx, with raw differentiated values in blue, and the filtered values overlayed.

2.3 Learning

A few tips for doing the learning:

- The network does not need to be very deep - 2 or 3 layers should suffice
- The network also does not need to be very wide - it should be on the order of 10's of hidden units per layer
- Explore how changing your activation function affects the performance of the network
- Try using different optimizers in torch.optim to see which one provides the best performance

- Shuffle the data before training and/or use random mini-batches
- Split your data into train and validation sets. Intermittently have the network perform inference on the validation set in order to approximate its performance on unseen data. This can also help you decide when to stop training the network (as to avoid overfitting)
- Use the GPU. It should not take more than a few minutes for your neural network to converge during training.
- A simple loss function for training your model is the mean squared error function. Feel free to try other loss functions.

2.4 Submission

- On average, what is the absolute value of the error between the labels in the validation set and your model's predictions? (We are expecting 3 values, where the first two have units of length and the second has units of radians - note that this is different from the value computed by your loss function)
- Perform rollouts of length 10 with your model where the controls are constant across time steps. Specifically, provide a figure for each of the following situations:
  1. $v = 0, \delta = 0$
  2. $v = v_{MAX}, \delta = 0$
  3. $v = -v_{MAX}, \delta = 0$
  4. $v = v_{MAX}, \delta = \delta_{MAX}$
  5. $v = v_{MAX}, \delta = -\delta_{MAX}$

  Do your rollouts match what you expect the car to do given the corresponding controls? (If not, you probably need to retrain your model)
- What neural network model did you end up using, and why did you pick it? Explain how many layers and the activations you chose.

2.5 Extra Credit: Learning Kinematic Model Residuals

Up to this point, your neural network has directly predicted the change in the robot's state. An alternative approach is to use the neural network to improve a known model, in this case the Kinematic car model. We now train our neural network to predict the error between the Kinematic car model's output and the ground-truth change in state. This error is called the residual. In other words, our old predictions were of the form:

$$x_{t+1} = x_t + f_1(x_t, \Delta x_t, u_t)$$

But we now make predictions of the form:

$$x_{t+1} = x_t + KinModel(u_t) + f_2(x_t, \Delta x_t, u_t)$$

The motivation for this approach is that it should be easier to learn the deficiencies of the Kinematic car model (i.e. the residual) rather than learning the whole motion model from scratch. Update your labels to be the residual between the Kinematic car model and the original ground truth labels, retrain your model, and again report the average absolute value of the error between your predictions and the labels. Does this model outperform your previous model?

2.6 Extra Credit: Collecting your own data

The data provided in the bag file is not from your car, so there may be some bias or offset in terms of predictive performance from your robot. If you have confidence in your own particle filter, you can attempt to collect your own data in a bag file, and learn from those values. You will probably need at minimum 20 minutes of data, with the car performing all necessary actions to fully explore the state and action space; driving backwards, forwards, turning fast and slow, etc.

If you want your car to drive faster than the default speed with the MPPI controller, you will have to do this process.

2.7 Extra Credit: Additional data inputs

The car has a few other sensors that may be useful to reducing the variance of the model. For example, the car's IMU provides values for angular momentum and linear accelerations. If you wanted to use these values as inputs to your model, you would have to also formulate how these values are expected to change in order to simulate future IMU values when performing rollouts. This could enable higher run-time performance, but would increase the complexity of your model.

If you do this, you will have to show how your model uses the data, and how you can perform multi-step rollouts with said data. Feel free to ask the TA's on approaches for this if you are not sure.

2.8 Extra Credit: Neural Network Particle Filter

After learning a neural network forward model from data, you can use your neural network as the forward model in your particle filter. To get the points for this extra credit, you will have to show us your neural network working as the motion model for the particle filter. Additionally, if you decide to do this (it would be an excellent verification of your model), you can use your Particle filter to collect additional data, and iteratively improve the neural network: if each data collection uses a better and better NN, then the Particle Filter should give better and better estimates, thus providing better data for better models. You may need to tune some PF parameters again for this part.

3 Model Predictive Path Integral

Once we have learned a forward model that can predict the next state given current state and control, we can use it for finding control sequences that give the car a trajectory to a target goal pose.

3.1 Path Integral Policy Improvement

Like the name suggests, Path Integral Policy Improvement, or PI^2, is a method that improves a parameterized policy function $a = \pi_\theta(s)$ that gives controls a for some state s. The theory behind PI^2 is beyond the scope of this homework assignment, but we give a brief overview here. We use a policy π_θ–think of a neural network that calculates actions for a given state input–to rollout a large number of simulated trajectories. Each trajectory starts from the same state, the state of the robot, and varies due to noise added to the control inputs. This requires a model to simulate the rollouts on (see section 2). From these large number of trajectories, we calculate the cumulative cost of each one through the use of a *cost function*. This function defines what properties of the trajectory we wish to reward and penalize, thus defining the task. For example, we can penalize deviation from a certain speed, reward smooth control changes, or put a cost on a distance to a target. PI^2 uses theses costs (one per trajectory), to adjust the policy parameters θ to more likely produce the low-cost trajectories. We will not use the full PI^2 algorithm, but instead a modification that should provide more robust performance for our robotic system.

3.2 Model Predictive Path Integral

MPPI combines the path integral update rule with a model predictive setting. Model predictive control (MPC) is the calculation of a short trajectory, and the use of only one or a few of that trajectory's controls, instead of solving for a longer and more computationally complex trajectory that optimizes for the task. The benefit of MPC is that we can constantly re-calculate the controls, making the run-time behavior robust to perturbations and modeling errors, as well as being faster to generate behavior.

To use PI^2 in an MPC fashion, the first obvious step is to reduce the horizon T, and only apply the first control values (v_0, δ_0) to the system during run-time, and recalculating at the next state. The next, less obvious step, is to forgo the policy π_θ itself. The reason to do this is that the θ most likely has many parameters, so would need a large number of samples to effectively learn/update. Instead, we represent the output of the policy: a sequence of controls $U = [u_0, u_1, ...u_T]$. Instead of changing the parameters θ, we directly change the values of U to minimize the cost. After we apply the first control values $u_0 = (v_0, \delta_0)$ to the robot, we recalculate U with MPPI again; even if the first applied controls were not long-term optimal, the hope is that eventually we will still accomplish the goal.

```
Algorithm 2: MPPI
Given: $\mathbf{F}$: Transition Model;
$K$: Number of samples;
$T$: Number of timesteps;
$(\mathbf{u}_0, \mathbf{u}_1, ...\mathbf{u}_{T-1})$: Initial control sequence;
$\Sigma, \phi, q, \lambda$: Control hyper-parameters;
while task not completed do
    $\mathbf{x}_0 \leftarrow$ GetStateEstimate();
    for $k \leftarrow 0$ to $K-1$ do
        $\mathbf{x} \leftarrow \mathbf{x}_0$;
        Sample $\mathcal{E}^k = \{\epsilon_0^k, \epsilon_1^k, \dots \epsilon_{T-1}^k\}$;
        for $t \leftarrow 1$ to $T$ do
            $\mathbf{x}_t \leftarrow \mathbf{F}(\mathbf{x}_{t-1}, \mathbf{u}_{t-1} + \epsilon_{t-1}^k)$;
            $S(\mathcal{E}^k) \mathrel{+}= \mathbf{q}(\mathbf{x}_t) + \lambda \mathbf{u}_{t-1}^{\mathrm{T}} \Sigma^{-1} \epsilon_{t-1}^k$;
        $S(\mathcal{E}^k) \mathrel{+}= \phi(\mathbf{x}_T)$;
    $\beta \leftarrow \min_k [S(\mathcal{E}^k)]$;
    $\eta \leftarrow \sum_{k=0}^{K-1} \exp\left(-\frac{1}{\lambda}(S(\mathcal{E}^k) - \beta)\right)$;
    for $k \leftarrow 0$ to $K-1$ do
        $w(\mathcal{E}^k) \leftarrow \frac{1}{\eta} \exp\left(-\frac{1}{\lambda}(S(\mathcal{E}^k) - \beta)\right)$;
    for $t \leftarrow 0$ to $T-1$ do
        $\mathbf{u}_t \mathrel{+}= \sum_{k=1}^{K} w(\mathcal{E}^k) \epsilon_t^k$;
    SendToActuators$(\mathbf{u}_0)$;
    for $t \leftarrow 1$ to $T-1$ do
        $\mathbf{u}_{t-1} \leftarrow \mathbf{u}_t$;
    $\mathbf{u}_{T-1} \leftarrow$ Intialize$(\mathbf{u}_{T-1})$;
```

Figure 2: This is the MPPI algorithm from the paper *Information Theoretic MPC for Model-Based Reinforcement Learning*, which you can read here. For more details in the MPPI math justification, or for additional details in their implementation, including their cost functions, their paper is a useful reference. This figure is to serve as an additional guide, but may need interpretation from the paper for certain symbols. For example, the ϕ symbol represents a final cost that we do not use here, although you are welcome to devise one. In the above, you can see that **F** is your neural network model.

3.3 Cost Function

The task we will implement for this class is for the car to drive to a goal pose. Therefore, your cost function will contain a term penalizing the distance between where the car currently is, and where you want the car to go. Additionally, you should include a term that *heavily* penalizes the car going out of bounds / through walls; this code will be similar to culling particles in your Particle Filter code. Finally, there is a penalty for large changes in controls u; this is written out in the inner for-loop over T in figure 2, with $q(x_t)$ being your cost function.

You should attempt to independently verify each term of your cost function acts as expected for various states and controls. This algorithmic debugging will help avoid bigger head-aches down the line when testing the system end-to-end. There are suggestions provided in the skeleton code, but you should thoroughly test your implementation to convince yourselves it is doing the right thing. MPPI assumes all costs to be ≥ 0.

3.4 Implementation

We assume your MPPI controller is operating along-side your Particle Filter, as we need a source of inferred poses.

At a high-level, your MPPI node should do the following; you should also consult the skeleton code for additional notes:

1. Pre-allocate GPU memory through torch for all the tensors required during initialization. Select hyper parameters Σ and λ for MPPI. Σ relates to the noise injected into the controls, and λ relates to the sensitivity to change in your update rule. Also create a nominal control trajectory $U = [u_0, u_1, ...u_T]$.
2. Your MPPI loop will first get the current state estimate; you should decide if you want to use this value directly, or smooth it over time with a low-pass or other kind of filter. This depends on your particle filter implementation. From this current state, you will perform rollouts.
3. Perform K rollouts for T time-steps. Your neural network model can accept a Kx(state size) matrix, outputting a Kx(control size) output matrix; this performs K rollouts for 1 time-step. You will accumulate K costs values, one for each rollout, summed along the length of the trajectory. The controls applied during each rollout is the main control sequence $U = [u_0, u_1, ...u_T]$ with additive Gaussian noise $N = (0, \Sigma)$, so $U^k = [u_0 + \epsilon_0^k, u_1 + \epsilon_1^k, ...u_T + \epsilon_T^k]$.
4. The MPPI procedure continues with calculating weights based on the costs. To do this, you first find the minimum cost value from all K costs (this should be > 0 if your cost function was formulated correctly). You then subtract this baseline β from each cost term, scale by $-1/\lambda$, and take the exponential: $exp(-\frac{1}{\lambda}(C^k - \beta))$; this, when divided by the sum of all exponentiated, scaled cost terms is weight $w_k = \frac{1}{\eta} exp(-\frac{1}{\lambda}(C^k - \beta))$ where $\eta = \sum_k^K exp(-\frac{1}{\lambda}(C^k - \beta))$. These K weights are then used to scale the contribution of the additive noise ϵ_t^k to the main control sequence U. The algorithm in figure 2 denotes cost as $S(\epsilon^k)$ for theoretical proof reasons.
5. After taking controls u_0 and sending them to the actuators of the car, you can shift each value of the controls down by one, $u_0 = u_1$..., to prepare for the next iteration. You should observe the values that MPPI outputs, and if they look noisy or volatile, you may need to tune your Σ and λ parameters, or even directly filter the control outputs to be smooth.
6. This process is repeated with each new estimated pose until the desired goal is completed; in this case you can specify a distance from the goal pose the car must reach before stopping, and waiting for the next input goal pose. This accuracy depends on the accuracy of your model and particle filter, the performance of your MPPI controller, and perturbations during run-time.

You should take care to debug your system in pieces before hoping that it works end to end. All matrix / tensor manipulations should result in the correct sized values outputs. You can test your controller on simple test values to make sure your cost functions and weighting system is indeed scaling things correctly. Visualizing the rollouts would be helpful. Only after you have convinced yourself that each component is behaving correctly can you reasonably expect the system to work as intended.

3.5 Compute and Car Performance

MPPI efficiently uses the data it collects, but still needs a lot of data. As such, you should keep in mind how to optimize your code after you have tested that it works. A useful feature of PyTorch is being able to quickly switch between processing on CPU vs GPU; you can still do all testing and development on your personal / lab computers, but of course, they are more powerful than the CPUs on the robot cars.

MPPI, being MPC-based, should be able to perform very interesting behaviors, such as 3-point turns, U-turns, and driving around corners. Does your controller perform these behaviors?

3.6 Submission

1. Explain what values your final MPPI controller used: T, K, Σ, λ. Explain how your car's behavior changed when these values were modulated. Which is more important, T or K, in generating good MPPI control?
2. How long did your final MPPI controller take to compute the next controls? Did you decide to use the GPU or CPU for the processing. Generate two plots, one for

timing vs $T = 5, 10, 15, 20, 25, 30, 40, 50, 60$, with $K = 500$, and one for $K = 100, 200, 400, 800, 1600, 3200, 6400$ with $T = 20$. Are they linear? Do they plateau? What explains their behaviors? Note whether the plots were generated with code running on the CPU or GPU.

3. Is your MPPI controller able to perform interesting behaviors such as 3-point turns and avoid bumping into walls? List the interesting and useful behaviors that your controller produced, and explain why you think your controller produced them. What could make your MPPI controller better?

4 MPPI Extra

4.1 Extra Credit: Perception based control

Incorporate the laser scanner into your MPPI controller. One way is to temporarily impose your most recent LIDAR scans into your map as obstacles your car needs to avoid; this would involve projecting the scan data into the world frame, and performing bounds-checking on this new, temporary map. You may also have to mask off areas behind scanned obstacles to avoid rollout trajectories thinking they can drive through them.

Another way to do this is to increase your neural network input size to include some subset of the laser data. When you now perform rollouts, you will have to send the rollout pose to `range_libc` to, on the GPU, calculate simulated laser scans, upon which the neural network would process. This would be giving your controller the ability to dynamically perceive obstacles, and drive around them. You may want to discuss this with the TAs before attempting this as this can be a bit more involved.

5 Demo

1. You will show us the expected error in your trained neural network model.
2. We will use your car's MPPI controller to dynamically drive to selected targets. There will be a progression of increasing difficulty, as we direct the car forwards, backwards and turning. We will note the number of collisions with objects already present in the map (which your controller should be able to avoid).
3. If you've found that your car can perform interesting behaviors, you will have the chance to demonstrate them to us.
4. You will be asked about your car's controller design and performance characteristics, and each team member's role in the implementation.