HW2 — 6.884: Computational Sensorimotor Learning

Cameron Hickert

April 2020

1 Problem 1: Learning Inverse Model

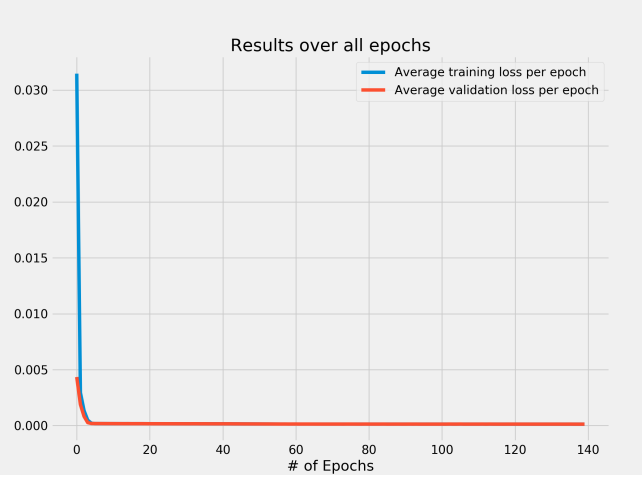
The source code is included (for all the problems) in Python files in the zipped folder. Each is labeled with p1, p2, or p3 as a prefix in accordance with the problem to which it corresponds. Files helpful to multiple problems do not include such a prefix. To train the inverse model, run the p1_training_inverse_model.py file. After training, this file saves the model's parameters and plots the loss over the course of training. Then run the p1_inverse_execute.py script to generate the pushes (along with the graphs and the error calculations). Finally, p1_make_vids.py outputs the videos.

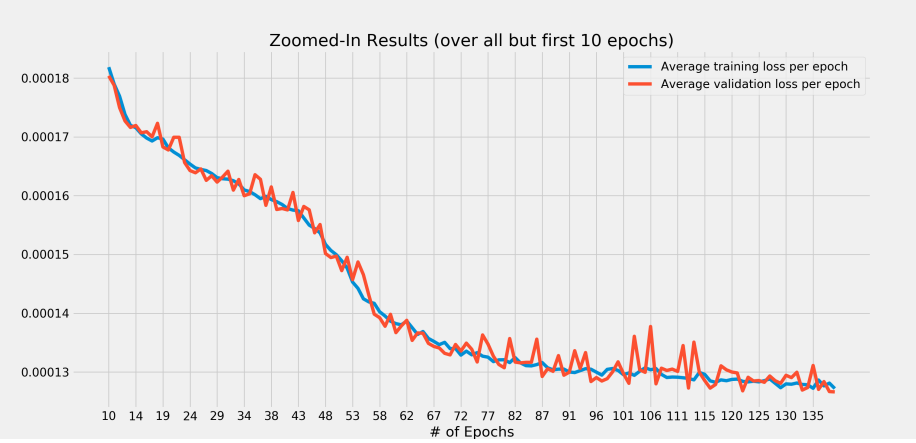
The inverse model is a feed-forward neural network that takes as input the current state and the desired future state, and outputs a predicted action. In our case, the combined input is vector of four values total (two each for the current state and goal state), and the output is also a vector of four values. I use two hidden layers, of size 64 and 32 respectively, with ReLU activation functions after each.

The loss function used for training is the mean squared error between the true action a_{true} and the action predicted by the inverse model $a_{predicted}$ (averaged across all n true-predicted action pairs):

$$L_{inverse} = \frac{1}{n}\sum_{i=1}^{n}(a_{true,i} - a_{predicted,i})^2 \tag{1}$$

The training plots showing loss as a function of time are below. First is a full version, followed by a zoomed-in version which better represents the behavior at later epochs.





The final train and test loss is 0.0001272 and 0.0001266, respectively.

The videos showing a ground truth push and the corresponding push performed by the forward model are included in the zipped folder. The number in the file title pairs the true and predicted pushes, so running them side-by-side allows for direct visual comparison. (For example, pred_pushes8.mp4 is the predicted push corresponding to true_pushes8.mp4.) The videos demonstrate that the the inverse model learned to perform fairly well in predicting the action necessary to achieve the output state from the initial state.

The state error (Euclidean norm between the true state and model's output state) and the action error (Euclidean norm between the true action and the action output by the model) are below:

Video	Action Error	State Error
0	0.0126	0.230
1	0.0366	0.237
2	0.0279	0.169
3	0.0202	0.173
4	0.0148	0.321
5	0.0134	0.216
6	0.0104	0.226
7	0.0149	0.0735
8	0.0301	0.0769
9	0.00655	0.122

2 Problem 2: Learning Inverse Model

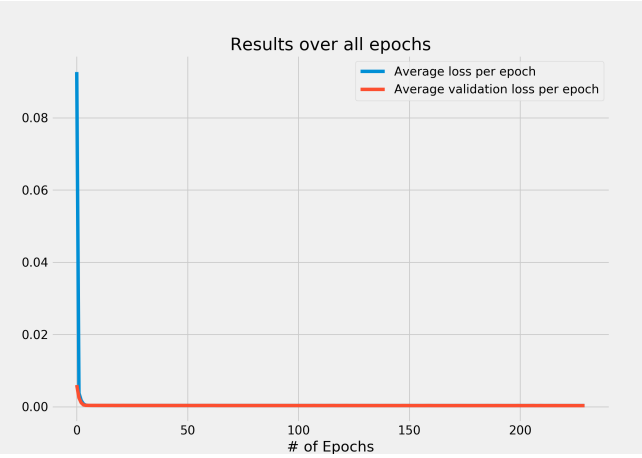
To train the forward model, run the p2_training_forward_model.py file. After training, this file saves the model's parameters and plots the loss over the course of training. Then run the p2_forwardcem_execute.py script to generate the pushes (along with the graphs and the error calculations). Finally, p2_make_vids.py outputs the videos.

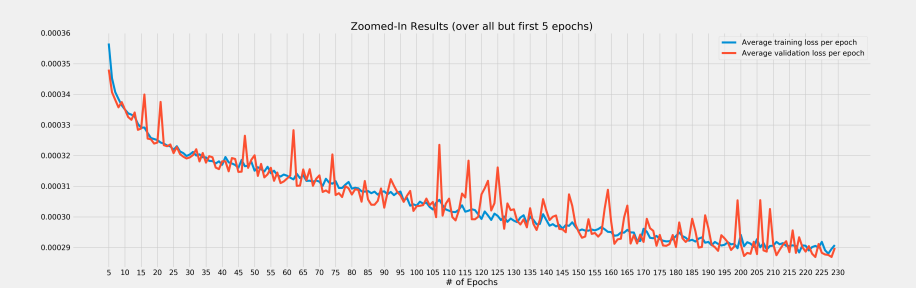
The forward model is a feed-forward neural network that takes as input the current state and the action, and outputs a predicted future state. In our case, the combined input is vector of six values total (two for the current state and four for the action), and the output is a vector of two values. Beyond these differences, the overall architecture I use here is identical to that used in the inverse model: two hidden layers, of size 64 and 32 respectively, with ReLU activation functions after each.

The loss function used for training is the mean squared error between the true goal state s_{true} and the end state predicted by the inverse model $s_{predicted}$ (averaged across all n true-predicted end state pairs):

$$L_{forward} = \frac{1}{n}\sum_{i=1}^{n}(s_{true,i} - s_{predicted,i})^2 \tag{2}$$

The training plots showing loss as a function of time are below. First is a full version, followed by a zoomed-in version which better represents the behavior at later epochs.





The final train and test loss is 0.000291 and 0.000290, respectively.

The core of the cross-entropy method (CEM) is available in the CEM.py file, with helper files as necessary. It leverages the learned forward model to to plan for pushing an object. The method itself is a Monte Carlo method for importance sampling, and operates as an "evolutionary" algorithm. That is, on each iteration it samples a variety of push angles and push lengths (each from a normal distribution explicitly parameterized by a mean and standard deviation). It then takes actions specified by those angles and lengths, and evaluates the loss associated with each. These results are sorted so that we can identify the best actions, and characterize those as the "elites." The parameters for the next iteration are then chosen to be the mean each of the set of push angles and push lengths that produced the elites. I originally included a smoothing parameter that would balance out the updated parameter and its previous version, but ultimately opted to set it to "1", which eliminates any smoothing effect – that is, the final result each iteration is simply the mean of the values from that iteration's elites for that parameter. This process continues for the specified number of iterations. I explored a few different configurations for the population size and the number of iterations, and found the results to be fairly stable to changes in these values.

The videos are included in the zipped folder. The number in the file title pairs the true and predicted pushes, so running them side-by-side allows

for direct visual comparison. (For example, pred_pushes8.mp4 is the predicted push corresponding to true_pushes8.mp4.) The videos demonstrate that the the forward model learned to perform fairly well.

The state error (Euclidean norm between the true state and model's output state) and the action error (Euclidean norm between the true action and the action output by the model) are below:

Video	Action Error	State Error
0	0.00538	0.157
1	0.0146	0.105
2	0.00230	0.254
3	0.0189	0.0506
4	0.0336	0.120
5	0.0167	0.0116
6	0.0475	0.0155
7	0.0283	0.145
8	0.0225	0.167
9	0.0110	0.337

3 Problem 3: Extrapolating with Learned Model

Since both models are trained by this point (assuming the training files have already been run for the previous two questions), simply run the p3_inverse_extrapolate.py and p3_forward_extrapolate.py files. Then run p3_make_vids.py to output the corresponding videos.

The videos showing the ground truth two-step push and the corresponding two-step push performed by the inverse model are included in the zip file. The same is included for the forward model as well.

The distances between the final object pose after the push and the goal object pose for 10 pushes with the inverse model are below, along with the Euclidean norm between the true action and the action output by the model for each of the two steps:

Video	Action 1 Error	Action 2 Error	End State Error
0	0.103	0.119	0.0559
1	0.0875	0.112	0.0557
2	0.0895	0.111	0.0283
3	0.0914	0.0930	0.0272
4	0.0707	0.0844	0.0609
5	0.119	0.107	0.0202
6	0.0825	0.0960	0.0364
7	0.104	0.0627	0.0236
8	0.0838	0.117	0.105
9	0.112	0.0485	0.0306

The distances between the final object pose after the push and the goal object pose for 10 pushes with the forward model (and CEM planner) are below, along with the Euclidean norm between the true action and the action output by the model for each of the two steps:

Video	Action 1 Error	Action 2 Error	End State Error
0	0.00944	0.0232	0.0268
1	0.0113	0.0367	0.0380
2	0.00970	0.0174	0.0197
3	0.00794	0.0151	0.0198
4	0.00844	0.0520	0.0557
5	0.0235	0.0461	0.0391
6	0.0391	0.0314	0.00708
7	0.0302	0.0344	0.0210
8	0.0268	0.0468	0.0350
9	0.0204	0.0571	0.0414

In general, the videos in both cases show that the models are able to extrapolate fairly successfully. And while they perform comparably in each case, I found that the forward model (with CEM planning) slightly outperformed the inverse model. The mean end state error over the 10 trials for the inverse model was 0.044, while the mean end state error over the 10 trials for the forward model was only 0.030.

Acknowledgements: In working on this problem set, I worked with Anurag, Jordan, and Josh.