Deep Extended Kalman Filter

Easton Potokar

1 Background

The Kalman Filter is used in a variety of applications including finance, medicine, robotics, etc. It attempts to clean or "filter" a sequence of states given state measurements. Namely, if x_n, u_n, z_n are the states, control, and measurements at timestep n respectively, the assumption is that the evolution of x and z are given by noisy linear systems and that x_n is normally distributed. The Kalman Filter then tracks the mean and covariance of x_n through each time step.

However, in practice, most systems $f(x_n, u_n)$ and $h(x_n)$ are nonlinear. To overcome this, the Extended Kalman Filter (EKF) linearizes the system about the current state estimate using system Jacobians. These assumptions and approximations are far from perfect since propagating a gaussian through a nonlinear function is no longer a gaussian. However, the EKF has still been the defacto method in industry for decades.

An alternate to the EKF is the Particle Filter. It propagates samples, or particles, through the distributions at each timestep, thus resulting in a near perfect representation of the distribution, given enough samples. The downfall of this method is that the required number of particles is often too large to calculate online.

The goal will be to merge the pros of the EKF and the pros of the Particle Filter - fast and exact distribution tracking - with Deep Learning. All code used is available at github.

2 Generating Data

The system we chose to use is that of a robot navigating through six landmarks whilst taking two range and bearing measurements at each timestep. This is a video of the EKF implemented on this dataset.

After setting up various parallelizations and just-in-time compiling, particle filters with 1,000 particles were run on 10,000 trajectories of 200 timesteps of this system, resulting in 2,000,000 data points. You can see an exact trajectory of the particle filter in 2(a). At each timestep, the first four centered sample moments of the distribution formed by the particles were calculated and saved. If $x_{i|j}$ is the jth element of the ith particle, the first three of these are given by

$$\mathbb{E}[X_j] \approx \mu_j = \frac{1}{N}\sum_{i=0}^{N} x_{i|j} \qquad \mathbb{E}[(X_j-\mu_j)(X_k-m_k)] \approx \frac{1}{N}\sum_{i=0}^{N}(x_{i|j}-\mu_j)(x_{i|k}-\mu_k)$$

$$\mathbb{E}[(X_j-\mu_j)(X_k-m_k)(X_l-\mu_l)] \approx \frac{1}{N}\sum_{i=0}^{N}(x_{i|j}-\mu_j)(x_{i|k}-\mu_k)(x_{i|l}-\mu_l)$$

Note if we were to gather enough moments, we could uniquely specify exactly what the distribution is. However, similar to including higher order terms when linearizing, the first few terms are generally enough. Further, two datasets were made - one with the correct starting point and one with a random (but close) starting point. The correct staring point data can be found here, and the random here. They were then split 9,000/1,000 into test/validation sets.

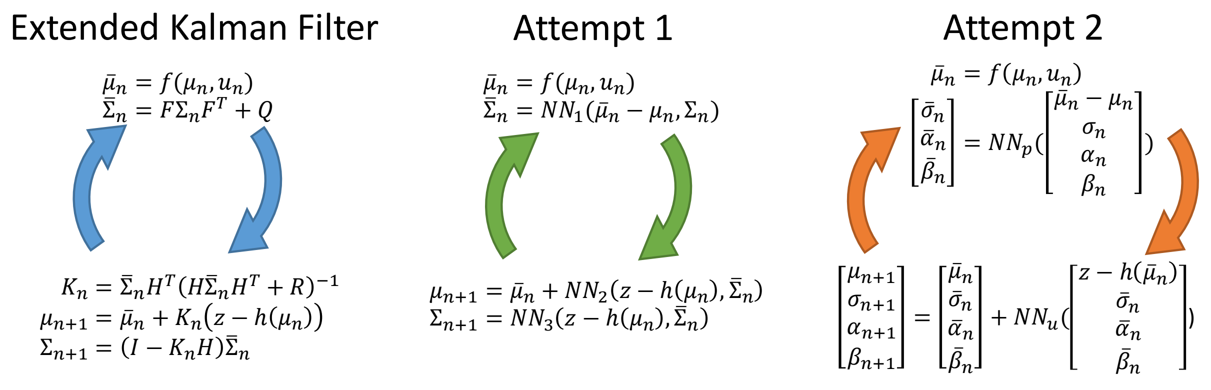
Extended Kalman Filter
$\bar{\mu}_n = f(\mu_n, u_n)$
$\bar{\Sigma}_n = F\Sigma_n F^T + Q$
$K_n = \bar{\Sigma}_n H^T(H\bar{\Sigma}_n H^T + R)^{-1}$
$\mu_{n+1} = \bar{\mu}_n + K_n(z - h(\mu_n))$
$\Sigma_{n+1} = (I - K_n H)\bar{\Sigma}_n$
Attempt 1
$\bar{\mu}_n = f(\mu_n, u_n)$
$\bar{\Sigma}_n = NN_1(\bar{\mu}_n - \mu_n, \Sigma_n)$
$\mu_{n+1} = \bar{\mu}_n + NN_2(z - h(\mu_n), \bar{\Sigma}_n)$
$\Sigma_{n+1} = NN_3(z - h(\mu_n), \bar{\Sigma}_n)$
Attempt 2
$\bar{\mu}_n = f(\mu_n, u_n)$
$\begin{bmatrix}\bar{\sigma}_n \\ \bar{\alpha}_n \\ \bar{\beta}_n\end{bmatrix} = NN_p(\begin{bmatrix}\bar{\mu}_n - \mu_n \\ \sigma_n \\ \alpha_n \\ \beta_n\end{bmatrix})$
$\begin{bmatrix}\mu_{n+1} \\ \sigma_{n+1} \\ \alpha_{n+1} \\ \beta_{n+1}\end{bmatrix} = \begin{bmatrix}\bar{\mu}_n \\ \bar{\sigma}_n \\ \bar{\alpha}_n \\ \bar{\beta}_n\end{bmatrix} + NN_u(\begin{bmatrix}z - h(\bar{\mu}_n) \\ \bar{\sigma}_n \\ \bar{\alpha}_n \\ \bar{\beta}_n\end{bmatrix})$

Figure 1: The 2 different attempts at replicating the Kalman Filter using Neural Networks.

3 Deep Learning Approaches

The Kalman Filter brings with it many mathematical gaurantees, like being the best linear unbiased estimator and various convergence guarantees, that we might get if we mimick the structure. The Kalman Filter has two steps: the predict step that integrates a control and a predict step that integrates measurements. Separating them like this also gives the flexibility to repeat steps as necessary. Our goal will be to model the nonlinearities anywhere there's an F or H in 1 whilst still maintaining the same basic structure.

Immediately you'll notice the similarilities to a RNN, in that the outputs become the inputs to the next iteration, with the modification that the two steps (predict and update) take turns performing. I considered using LSTMs, but didn't see the masking of the hidden state being a good fit since the moments we're trying to model are constantly changing.

My first simple attempt was to replace all nonlinearities with a neural network, as in 1. Due to shifting by the mean, the 2nd-4th moments will all be centered around 0 for the most part, and will be relatively bounded regardless of what's going on with the robot. However, the mean (the actual location of the robot) and potentially the measurements are extremely unbounded and can easily veer into areas where our networks weren't trained on. To combat this, we always input the differences for our mean and measurements. In the predict step, using the system $f(x_n, u_n)$ is exact and should be used.

For the topology of the network, I simply flattened the covariances and then used a simple Dense Neural Net. I trained on each network separately as a sort of "burn in" period, then used them together as a sequence alternating with and without teacher forcing, all using the Adam Optimizer. I ran into a number of issues with this method. First was the complexity. It was difficult to shuffle three networks all with different inputs along with all the ground truth data. Second, I wanted to avoid normalizing to be able to generalize the network in the future, but this just caused performance issues. Next, I had repeated inputs since covariance matrices are symmetric. Finally, I had issues with both exploding *and* vanishing gradients. In some training sessions the network would return the same output for all inputs, and in others it would output ridiculous values.

I fixed all of these, along with beginning to use 3rd and 4th moments, in my second attempt. I simplified things down to two networks, as seen in 1. I also began only using the unique values of the moments. This was six values for covariance (covariance is a 3x3 matrix in this case), 10 for 3rd moment (skew), and 15 for the 4th moment (kurtosis). To fix the exploding gradients, I began normalizing my inputs and outputs to have mean 0 and standard deviation 1. For vanishing gradients, I began using residuals, which consequently also helped training, along with trying a few different methods of weight initialization including Xavier and Kaiming and some weight regularization. I attempted various combinations of all of these along with combinations of learning rate, L1/L2 loss, network depth, and network width on both datasets.

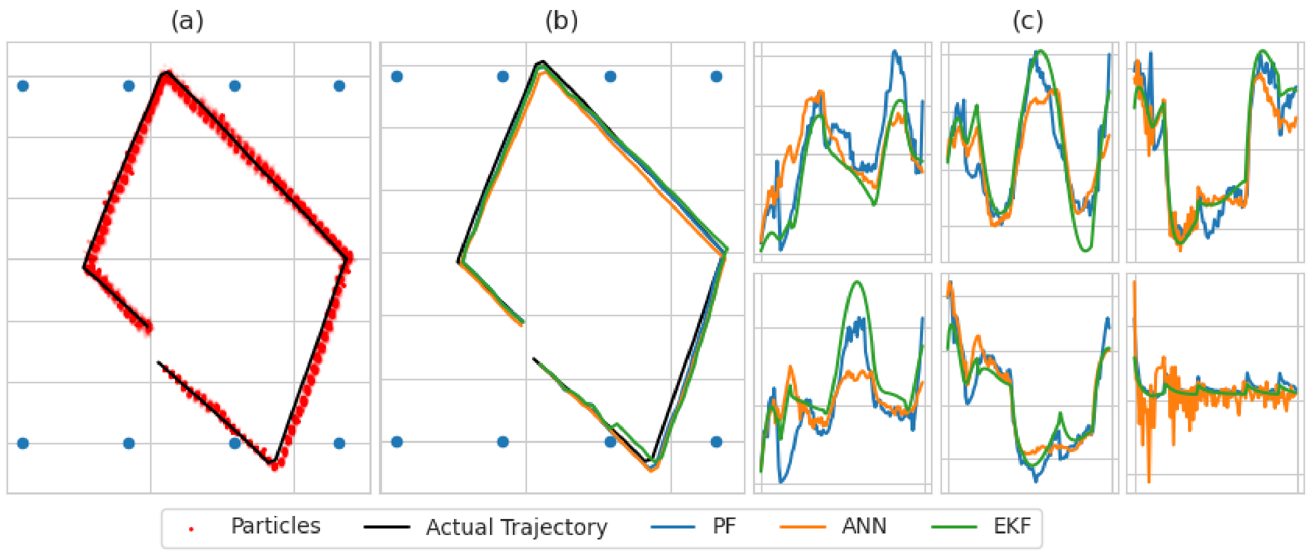

Figure 2: (a) An example of the data generation. At each time step, the first four moments were calculated and used as training data. Resulting means (b) and covariances (c) of ANNs as compared to the true trajectory, Particle Filter, and Extended Kalman Filter. This was using a trajectory from correct starting position validation data.

I also noticed that after all this, it was training decently on all the outputs. However, while I want decent tracking of the higher order moments, I care a lot more about the mean and covariance. I weighted my loss accordingly, multiplying the mean and covariance portion by constants > 1 to make them train better.

4 Results

The results look promising and can be seen in 2(b,c). In (b) you can see the that our ANNs track the robot state very well. It performs slightly better than the EKF. In (c) you can see the covariance also tracks very well. The exception is the bottom right plot, where things look extra noisy. This is largely a scale issue, the y-axis scale of that plot is about 10^{-1}.

While we got good results like this for the correct starting point dataset, the random start dataset didn't perform nearly as well. I believe it was due to the data being very skewed; there was large covariances and innovations at the start, but these quickly reduced to normal levels. Perhaps with a dataset with shortened trajectories and more time localizing it would have more information to learn from.

Unfortunately, I don't believe the ANNs were much faster than the PF in this case either, but for higher dimensional systems I imagine they would be due to a larger number of particles being needed.

5 Further Work

While our ANNs did the job they were supposed to, I don't see them replacing EKFs or PFs anytime soon (at least for this system). I'd like to find a system that the EKF performs very poorly on and the PF is extremely slow on to see if our ANN model performs better than both.

I'd also like to experiment with more exotic network topologies. I believe 1D convolutions might capture the relationship between states/covariances better, and perhaps I was wrong to rule out LSTMs and other RNNs. Given more time, I would also do a proper hyperparameter search as well, since I mostly just tried various combinations by hand. Here is my time log for the project:

Date	Hours	Type	Description
10/13/2020	2	R	Spent time digging into specifics of kalman filter
10/14/2929	2	R	Derived extended kalman filter to understand where shortcomings are
10/28/2020	3	D	Began setting up dataset generation, spent extra time to "jit" to be able to generate LOTS of data FAST
10/30/2020	3	D	Got EKF, UKF, PF all set up for systems. Can use any at this point to generate data.
11/11/2020	1	D	Generated data for simple odometry example using PF. Took ~2.5hrs to generate ~2million datapoints
11/11/2020	1	N	Outline all variations of networks to try, and set up gameplan for exactly what inputs/outputs are
11/13/2020	2	N	Tried various inputs for Predict_Sigma network
11/14/20	3	N	Tried changing architecture some for Predict_Sigma network
11/16/2020	2	N	Finished setting up dataloader class for other 2 networks
11/16/2020	1	N	Finished setting up network for Update_Mu and Update_Sigma
11/17/2020	2	N	Start setting up all training - each network seperately along with one that iterates through them all
12/1/2020	3	N	Improved training together sequence - each trajectory has 200 pts, does n trajectories at a time to leverage GPU
12/2/2020	1	N	Investigate/solve NaNs when not using teacher forcing.
12/4/2020	1	N	Figured out NaN problem, but now UpdateMu ALWAYS returns same value, even before training.
12/7/2020	1	N	UpdateMu (and all networks) were too deep, had to reduce depth to get to train better. (Alternate fix - residuals)
12/7/2020	1	N	Got it working! Did a little normalization by hand to get it to work.
12/8/2020	1.5	N	Did better normalization, works even better now. Sigma is predicting way off though.
12/8/2020	0.5	N	Fixed UpdateSigma not working well
12/9/2020	1	D	Modified data simulator to track higher order moments
12/9/2020	1.5	N	Moved onto doing higher moments, cleaned all network code EVERYTHING as I shifted for it.
12/9/2020	0.5	N	Tried ResNets, they train MUCH better

Key		ACTIVITY TYPES	MY TOTAL	CLIPPED TOTAL		CATEGORY MAX
R	Researching	R	4	4		5
D	Dataset Prep	D	8	8		10
N	Coding Networks	N	22	22		1000
		Total	**34**	**34**	**Allowable Total**	