Bayesian Nonlinear Filter for Training Neural Networks
-An Investigation-

Jason Nezvadovitz

4/26/2017

Lead and Supporting Documents:

[1] Arasaratnam, Ienkaran, and Simon Haykin. "Nonlinear Bayesian filters for training recurrent neural networks." MICAI 2008: Advances in Artificial Intelligence (2008): 12-33.

[2] Haykin, S., Ed.: Kalman Filtering and Neural Networks, Wiley, New York (2001). Chapter 2.

The textbook chapter [2] primarily helped me with implementation of the theoretical ideas which I found were better handled by the paper [1]. No surprise, [1] references [2]. They also share an author; hopefully that is okay.

Usually I like to provide lengthy self-contained write-ups, but this time I'll just refer you to the above documents for that. Instead, here I'll provided a *simpler*, more straight-forward review.

Summary

As you know, a neural network is a nonlinear function of its weights $\hat{w}$ and the input u,

$$y = h(\hat{w}, u)$$

For example, with $u \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$, the simplest multilayer perceptron (a feedforward, fully-connected neural network) can be expressed as,

$$h(\hat{w}, u) = \hat{W}_2 g(\hat{W}_1 u)$$

where the matrices $\hat{W}_1 \in \mathbb{R}^{l \times n}$ and $\hat{W}_2 \in \mathbb{R}^{m \times l}$ are just a reorganization of the elements of $\hat{w} \in \mathbb{R}^{l(n+m)}$. (The function $g : \mathbb{R}^l \to \mathbb{R}^l$ is just an element-wise application of the l neuron functions). Even recurrent neural networks (which have activation feedback) can be expressed similarly if the internal states are included in u.

The fundamental idea presented in [1] is that *training a neural network can be viewed as a stochastic estimation problem* (or at least, that was the idea that got me hooked). There is a reason for putting a hat on $\hat{w}$. The Universal Approximation Theorem (UAT) basically promises us (under some easy conditions) that for all training data input-output pairs (u_i, z_i) there exists a w such that,

$$\sum_i ||z_i - h(w, u_i)|| < \epsilon$$

where ϵ can be made arbitrarily small with proper choice of neuron function g and a finite number of neurons $l < \infty$. In [1], we are not concerned with network architecture (designing g and l), but rather with training, i.e. *estimating* w.

Let us encode the UAT equation as a stochastic process with state variable $w_t \sim \mathcal{P}_t$. First, note that for a specific set of data, this ideal w is fixed, so from "time" (iteration, sample, etc...) t to $t+1$,

$$w_{t+1} = w_t$$

We can't measure w directly; we only get partial information in the form of the training data,

$$z_t = h(w_t, u_t) + \nu_t$$

where $\nu_t \sim \mathcal{R}_t$ is a random sequence that encodes both the ideal error ϵ and the possibility that our training data z_t has noise (i.e. z_t is not *exactly* what we want, but rather a noisy version of it). In [1] and [2], they include an "artificial" noise in the w_t process equation as well, but later say that they practically always make it ≈ 0. Also, note that we could have just written $h(w_t, u_t)$ as $h(w, u_t)$ since w is static. Here, time t is just the iteration of the training process, so in a sense the training data is considered a known timeseries. When the end of the data set has been reached, it can be randomly reordered and then presented again (with each presentation being called an "epoch").

This is a hidden Markov model (HMM), which is the name for a POMDP when there is no "decision" to be made; just a state to estimate. Additionally, the process equation is trivial. However, we cannot directly attack this with the nonlinear filter covered in class, because both the state and observation spaces are continuous. Well I mean, we can, but the sums will be integrals (multidimensional integrals, in fact). So instead we will discretize our spaces into grids... haha no. Never again.

What we will actually do is what [1] does: make assumptions.

- w_0 is Gaussian, i.e. our known prior is $\mathcal{P}_0 = \mathcal{N}(\hat{w}_0, P_0)$
- ν_t is zero-mean Gaussian, i.e. $\mathcal{R}_t = \mathcal{N}(0, R_t)$
- h is linear in w_t, i.e. $h(w_t, u_t) = Hw_t + h_u(u_t)$

Hold on! That linearity assumption could not be farther from the truth. Well, these are the assumptions that would make the following algorithms *optimal*. The fact that they end up working well is mostly heuristic, as we'll discuss later.

At the core of the nonlinear filters we will discuss, is a little-known linear thing some call the Kalman filter (sarcasm). I will not derive it here (Google can give you thousands of derivations), but rather outline its logic. If the prior distribution $\mathcal{P}_0$ is Gaussian, the noise distribution $\mathcal{R}_t$ is Gaussian, and the function h is linear (hurts to say that), then the belief $\mathcal{P}_t$ will remain Gaussian for all time. This is because linear combinations of Gaussian random variables are also Gaussian. The belief certainly encodes all possible statistics we could want for estimating w_t, but conveniently if it is Gaussian, it is sufficient to just know a mean $\hat{w}_t$ and covariance P_t. Additionally, the mean and mode of a Gaussian coincide, so $\hat{w}$ is both the maximum a-posteriori (MAP) and minimum mean-square-error (MMSE) estimate of w. The Kalman filter just analytically carries out recursive Bayes for this special case. Applied to our HMM with initial condition $(\hat{w}_0, P_0)$, it is,

$$K_t = P_t H^T (H P_t H^T + R_t)^{-1}$$

$$\hat{w}_{t+1} = \hat{w}_t + K_t (z_t - h(\hat{w}_t, u_t))$$

$$P_{t+1} = P_t - K_t H P_t$$

Much like the Kalman filter, the nonlinear filters that [1] employs attempt to estimate the first and second moments ($\hat{w}_t$ and P_t) of $\mathcal{P}_t$ as well, and use $\hat{w}_t$ as the current estimate of w. They become nonlinear filters when they add on their own special heuristics to engage our assumptions. For example, the "cubature Kalman filter" carries out numerical integrations with Gaussian functions

to employ the equations that otherwise analytically simplify to the Kalman filter equations. The "central-difference Kalman filter" uses finite-differencing to compute,

$$H_t := \frac{\partial h}{\partial w}\Big|_{w=\hat{w}_t}$$

at each timestep and replaces H with H_t in the above Kalman filter equations. Lastly, the "extended Kalman filter" (EKF) does the same thing but using a known analytical derivative of h. There is also the unscented Kalman filter (my personal favorite) which uses the unscented transform and is only discussed in the last chapter of [2]. Any of these algorithms are run until you are satisfied with your neural network's fit on the training set or a validation set.

The paper also covers an interesting demonstration of how the EKF is equivalent to Newton's method (just read section 4.1 of [1]). Compared to the notorious backpropagation-gradient-descent algorithms, [1] claims and supports that these Bayesian methods provide increased accuracy / superior convergence due to their approximate second-order nature. However, despite needing less iterations with them, [1] does not claim that they are always faster since each iteration takes longer.

Critique / Discussion

After reading [1] and [2], I was really only left unsure about the strictness of the conditions for convergence. Gradient-descent (GD) just "moves down", ya know? With GD, there *should* be a bell deep enough for your chosen step-size to rattle in. On the other hand, these algorithms do "extra stuff" and I don't have a perfect picture of how they will move. Ideally they will only move "up" for a moment if it helps them move very down soon after.

Additionally, if the EKF is Newton-Raphson, it will fail whenever the Hessian is ill-conditioned (i.e. if the inverse defining K_t doesn't exist). Does this happen often with neural networks? Should I be worried? Well, as an engineer, my approach to answering that was to try a couple simple examples and then say "yup, always works".

I chose to implement the EKF method so I can call it "stochastic Newton-Raphson" to bother Sean (since it may not be exactly what he calls stochastic Newton-Raphson). The code for my Python implementation can be found here: https://github.com/jnez71/kalmaNN. It is very user friendly and well documented in the docstrings (not the README). The file knn.py contains a class for creating and training a neural network by either EKF or GD. The other files are just demos.

The only real point of concern in implementation is computing that damn Jacobian $\frac{\partial h}{\partial w}$. I chose to use a multilayer perceptron (MLP) with one hidden layer, so with a little tensor math (hours of expanding scalar equations and pulling my hair out) we have,

$$h(\hat{w}, u) := \hat{W}_2 g(\hat{W}_1 u)$$

$$\frac{dh}{d\hat{W}_2} = \text{block\_diag}(g(\hat{W}_1 u)^T)$$

$$\frac{dh}{d\hat{W}_1} = \frac{dh}{dg}\frac{dg}{d\hat{W}_1} = \frac{dh}{dg}\frac{dg}{d\hat{W}_1} = \left(\hat{W}_2 \odot \frac{dg(\hat{W}_1 u)}{d\hat{W}_1 u}\right) \otimes u$$

where $\odot$ is the Hadamard product (element-wise multiplication) and $\otimes$ is the outer product. We can consistently construct $\frac{\partial h}{\partial w}$ from the elements of $\frac{dh}{d\hat{W}_1}$ and $\frac{dh}{d\hat{W}_2}$. The reason I formulated it this way was to take as much advantage of memoization and vectorization as backpropagation does (the equations are very similar).

Another important implementation detail is the inclusion of biases. The neural network equation we saw at the beginning looked like neurons applied to linear combinations of the input, but really it is supposed to be affine combinations if you want to satisfy the UAT. To reconcile this, the input u is given one extra element that is always 1, and an extra neuron g_{l+1} is set to permanently output 1. That breaks no prior derivations.

The rest of the implementation is pretty straightforward, so lets look at some results. In the following, I initialized the network weights random uniform on $[-10, 10]$.

For the first test, 100 points were evenly sampled over the domain $[-10, 10]$ of the following function,

$$z = e^{-u^2} + 0.5e^{-(u-3)^2} + \nu$$

where $\nu \sim \mathcal{N}(0, 0.0025)$. Pretending we don't know what function produced that data anymore, the MLP was trained on those samples to approximate it. A logistic activation function was used for $l = 10$ neurons in the hidden layer. The initial weight covariance P_0 was set to $0.5I$, and the sensor variance R was accordingly set to a constant 0.0025. The training was run for 100 epochs (presentations of *randomly ordered* training data). The weight covariance trace decreased steadily during training, implying an increasing confidence in the fit. To compare the results, GD was also used to train an identical MLP (over the same data that was used for the EKF training). The faster GD was given a step-size of 0.05 and 400 epochs. The results are shown in Figures 1-4. Note that a lot more trails with different parameters were run; these just represent the common result.

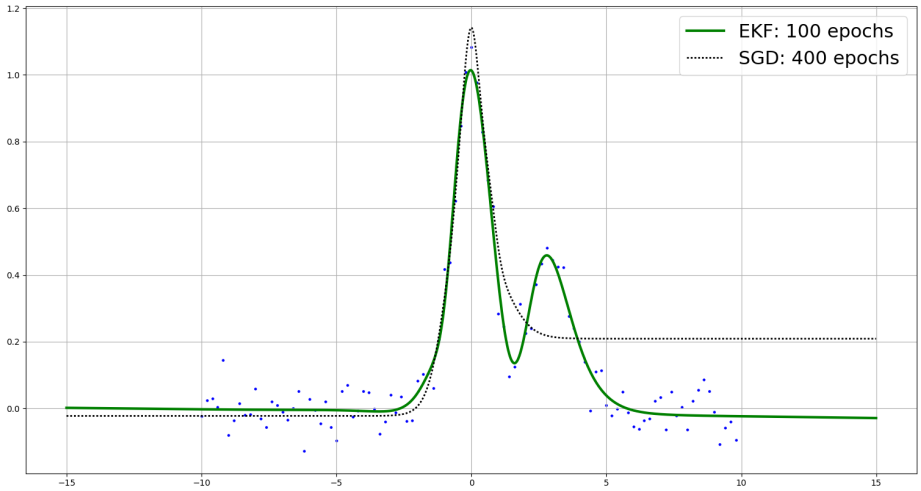

Figure 1: A test to fit an arbitrary nonlinear function with only sparse noisy samples. Compared to GD, the EKF training method took longer per iteration, but was able to achieve a much better fit in only a quarter of the number of epochs.

For the second test, the MLP was used to classify 2D data. The MLP output can be interpreted as classifications by selecting the class integer closest to the floating-point output of the MLP. The classes for this test "spiral" into one-another, making them not linearly-separable. The configuration (number of neurons, initial weight covariance, etc...) was initialized exactly like in test 1, but with GD only given 200 epochs. Results (with a comparison to GD) are shown in Figure 5. The two MLP's were trained and reset 50 times each (over the same randomized data as each other). The MLP trained by EKF had an average accuracy of 90% (with 2% standard deviation) while the MLP trained by GD had an average accuracy of 84% (with 5% stdev).

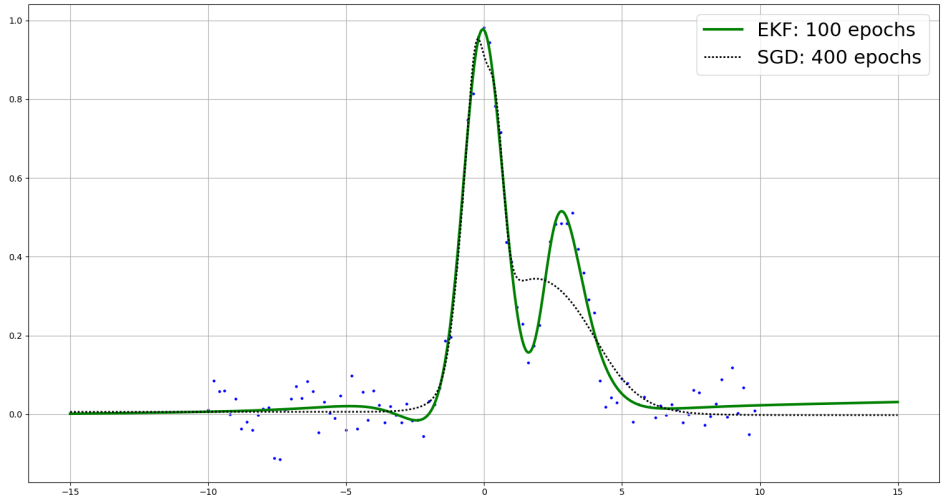

Figure 2: Another run of the 1D data fit test. Comparing this to that of Figure 1, notice the similarity in convergence provided by the EKF method, despite random initial conditions.

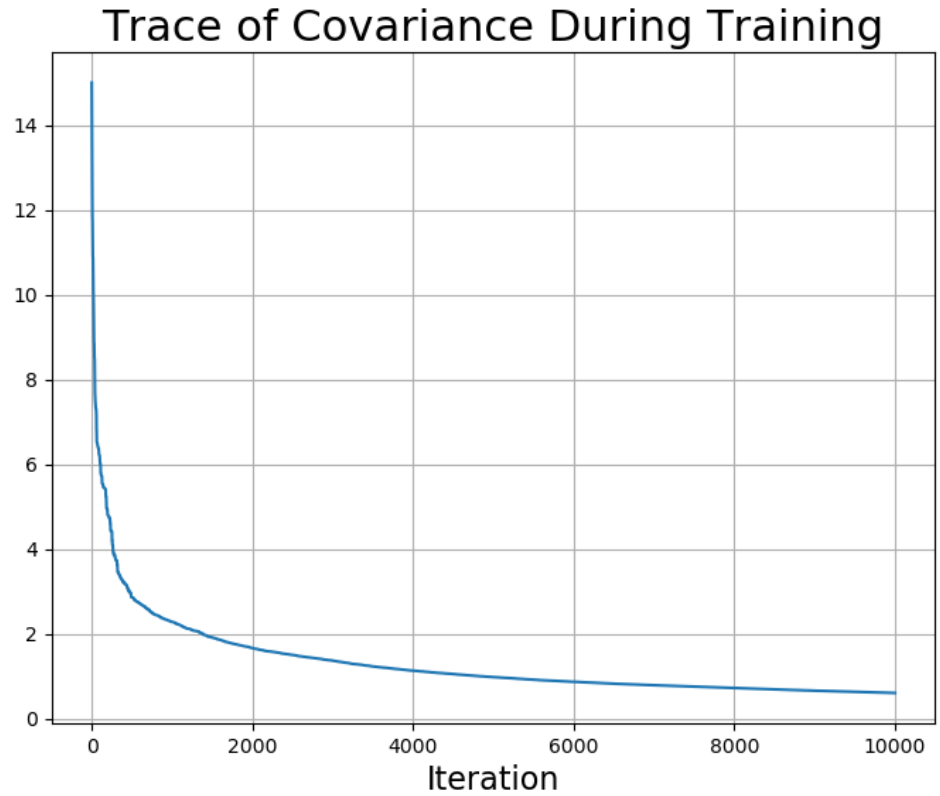

Figure 3: The decay of the trace of the state covariance matrix P during the training.

For the final test data, 100 seconds (timestep 0.01 seconds) of a 3D Lorenz Strange Attractor dynamic was simulated, with the MLP's objective being to learn the mapping from state to state-

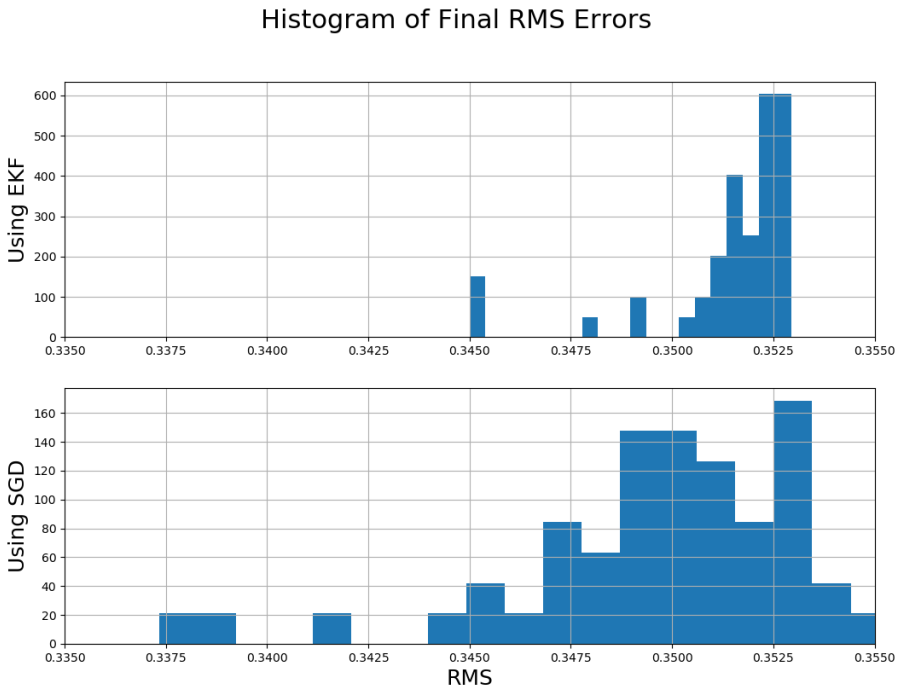

Figure 4: The same 1D fit problem was run 50 times, and the RMS error of the final fit was recorded. This histogram shows that the EKF method provides a tighter convergence variance than GD.

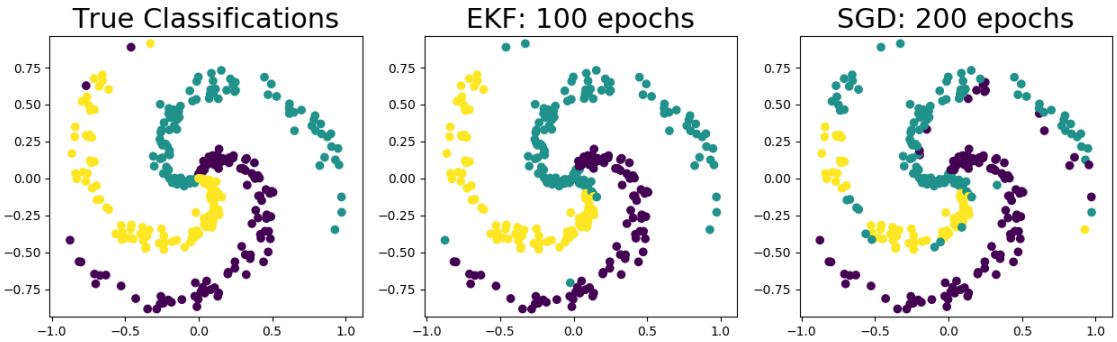

Figure 5: The second test involved training to classify noisy non-linearly-separable 2D data. A couple of purposeful "miss-classifications" were including in the training set to further noise the data. The EKF-trained MLP was 90% accurate on average, while the GD-trained MLP with 84% accurate. Also, the EKF training took half as many epochs (although computational time was about even between the two).

derivative, which is analytically given by,

$$\dot{x} = \begin{bmatrix} 10(x_1 - x_0) \\ x_0(28 - x_2) - x_1 \\ x_0 x_1 - 2.6 x_2 \end{bmatrix}$$

The MLP was given 30 neurons each using tanh for activation. Both P_0 and R were set to $0.5I$. The data was only presented once (one epoch). Once trained, another simulation of the Lorenz dynamic was performed starting from the same initial condition, but this time using the MLP to compute the state derivative. The resulting evolution is shown in Figures 6 and 7. The Lorenz dynamic

is chaotic, and thus just the fact that the MLP's autonomous predictions produced a somewhat similar trajectory is impressive (a recurrent neural network would work a lot better for this specific test; I mostly did it for fun). No comparison to GD for this test (I need to turn this in now).

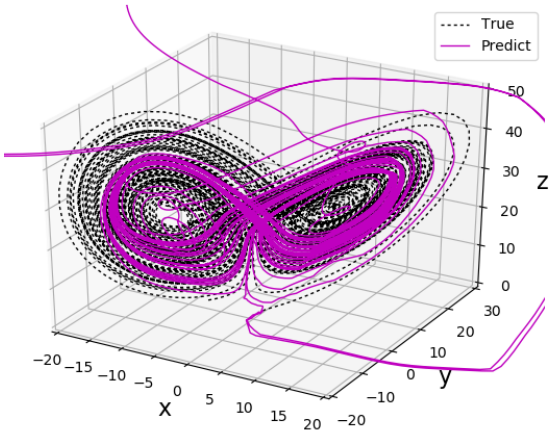

Figure 6: The third test involved attempting to predict the behavior of a chaotic Lorenz system after training on a short simulation of one (the black dashed line).

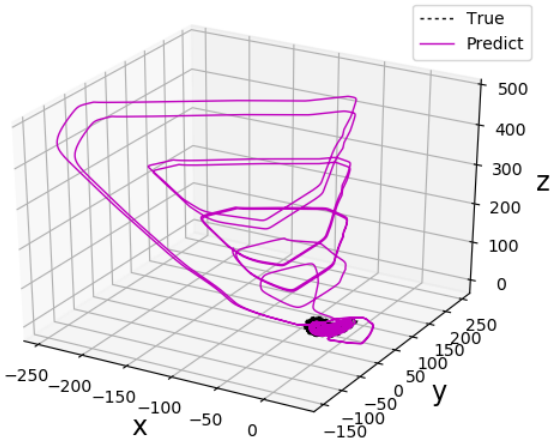

Figure 7: A zoomed-out view of Figure 4, showing that the learned dynamic was not unstable, but rather had large strange-attractor-like oscillations.

Overall I am pretty impressed with the methods presented by [1] and [2]. The EKF training actually did provide both better fits and a *smaller convergence variance* (i.e. more consistent results) which is really nice. The theory was right, and at least on these tests I never ran into an issue with infinite K_t. And as expected, EKF training converged in less iterations than GD but took longer per iteration.