Extreme Learning Machine Application

Qi Wang[1]
Department of Statistics, University of California, Santa Cruz[1]

Abstract

In this report, I reproduced the ELM method in estimating the target function value of both a 1-dimensional function and a 2-dimensional function. The target functions are proportional to Gaussian Mixture density function, which is not linear. Both uniformly and normally generating weights and intercepts of hidden layers are considered, and the predictive result of them are very similar to each other. Also, for the activation function, after trying ReLU and sigmoid function, sigmoid function performs better than the other one for this case. Number of hidden nodes are also considered, for the 1-dimensional case, if the number of nodes is too large, it will have an effect of overfitting for the ReLU case, but it is not so obvious for the sigmoid case. When it comes to the 2-dimensional case, due to the limitation of computing ability, I only tried the case for 200 nodes, the prediction error keeps going down for ReLU case but increases a little for sigmoid case, so the overfitting problem still exist. Overall, the best combination is to use sigmoid function with 200 hidden nodes, either normally or uniformly generating the weights does not matter so much.

KEY WORDS: Extreme Learning Machine, Linear Models, Least Square Estimation, Single Hidden Layer Feedforward Networks

1.Background

1.1 Model Overview

Neural networking is a hot topic in deep learning. It is like a black box where you input values of covariates, then it would output a predicted value by using tons of linear and nonlinear functions in the black box. There could be many steps of operations in the middle, and each operation is called a hidden layer, each function in the hidden layer is called hidden units. Notice that if all the functions in the hidden layer is linear operation, the output could only be a linear function of the inputs, so we must do some operations that are nonlinear, and these operations are carried out after a linear operation in each hidden layer, called activation functions. There are many popular functions like:

$$sigmoid : f(x) = \frac{e^x}{1+e^x}$$

$$ReLU : f(x) = max\{0, x\}$$

With these functions, we can break the linearity of the functions then simulate other nonlinear functions that we want. In this report, we are mainly talking about SLFN, which the information only flows from input to output in a single direction, and with only one hidden layer, called single hidden layer feedforward networks. The model can be expressed as follows, for each x_i

$$j^{th} Hidden\ Unit(Not\ Activated) : W_j x_i + c_j$$

$$Hidden\ Unit(Activated) : h_{ij} = g_j(W_j x_i + c_j)$$

$$Output\ Layer : t_i = \beta(h_{i1}, h_{i2}, ..., h_{ik}) + b$$

For classical neural networking, there are tons of hyperparameters in each hidden layer and hidden unit. So how to get the best set of them is a necessary step to fit our model. In this report, We are going to use the extreme learning machine method combined with linear models to make the regression. It includes a combination of content in linear models and also in the machine learning concepts. Therefore, I am going to use the simulated data to reproduce the ELM method including both one-dimensional and multidimensional.

1.2 Extreme Learning Machine Theory

In this section, two basic theorem about extreme learning machine method is introduced, which guarantee the convergence of the model.

Theorem 1.2.1

Given a standard SLFN with N hidden nodes and activation function $g : R \to R$ which is infinitely differentiable in any interval, for N arbitrary distinct samples (x_i, t_i), where $x_i \in R^n$ and $t_i \in R^m$, for any w_i and b_i randomly chosen from any intervals of R^n and R, respectively, according to any continuous probability distribution, then with probability one, the hidden layer output matrix H of the SLFN is invertible and $||H\beta - T|| = 0$

Theorem 1.2.2

Given any small positive value $\epsilon > 0$, and activation function $g : R \to R$ which is infinitely differentiablein

any interval, there exists $\tilde{N} \leq N$ such that for N arbitrary distinct samples (x_i, t_i) where $x_i \in R^n$ and $t_i \in R^m$, for any w_i and b_i randomly chosen from any intervals of R^n and R, respectively, according to any continuous probability distribution, then with probability one, $||H_{N\times\tilde{N}}\beta_{\tilde{n}\times m} - T_{N\times m}|| \leq \epsilon$.

Conclusion

Therefore, based on the two theorems, we can simply use least squares estimate to get the BLUE of β and b based on the randomly generated hidden layer weights and hidden layer intercept. Also, if we use the Moore–Penrose generalized inverse to solve the normal equation, the result will perform better.

2.Dataset

2.1 Data Generating Machanism

As mentioned above, I am going to use two data sets simulated manually. The first one should be a simple regression which only includes one covariate and the other includes two covariates. Furthermore, the response variable are both one-dimensional for them. And the data generating processes are as follows:

$$Y_i = f(X_i) + \epsilon_i \ , \ \epsilon_i \sim_{iid} N(0, \sigma_1^2)$$
$$f(x) = D_1(x) + 7 * D_2(x) + 2 * D_3(x)$$

with D_1 is the PDF of $N(\nu_1, \tau_1^2)$, D_2 is the PDF of $N(\nu_2, \tau_2^2)$, D_3 is the PDF of $N(\nu_3, \tau_3^2)$. And

$$\nu_1 = 1, \nu_2 = 5, \nu_3 = 9, \tau_1 = \tau_2 = \tau_3 = 1$$

And it looks like a Gaussian mixture PDF as in figure 1

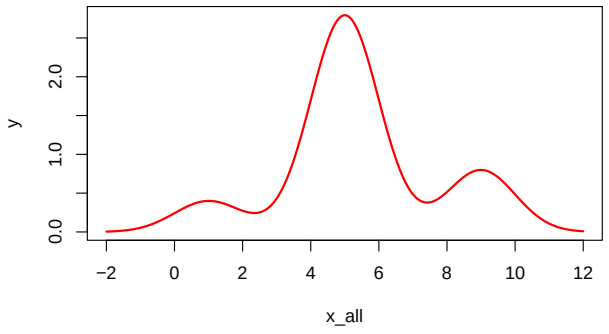

Figure 1: Data Generating Process for Model 1

Furthermore, the other data set will be generated as follows:

$$Y_i = f(X_{1i} \ , \ X_{2i}) + \epsilon_i \ , \ \epsilon_i \sim_{iid} N(0, \sigma_2^2)$$

$$f(x_1, x_2) = 4 * D_1(x_1, x_2) + 6 * D_2(x_1, x_2)$$

D_1 is the PDF of $MVN_2(\mu_1, \Sigma_1)$ and D_2 is the CDF of $MVN_2(\mu_2, \Sigma_2)$ with

$$\mu_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \Sigma_1 = \begin{bmatrix} 1 & 0.4 \\ 0.4 & 1 \end{bmatrix}$$

and

$$\mu_2 = \begin{bmatrix} 4 \\ 4 \end{bmatrix}, \Sigma_2 = \begin{bmatrix} 1 & -0.7 \\ -0.7 & 1 \end{bmatrix}$$

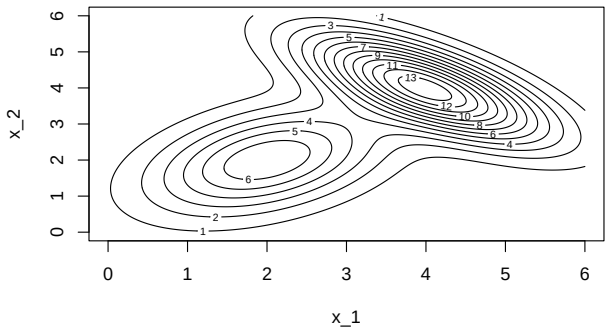

Figure 2: Data Generating Process for Model 2

Here, be careful that the Gaussian density is only the data generating process for y instead of the distribution function of a random variable. When we are generating the data, we will use the y calculated by these two function plus an error term, which follows another normal distribution.

2.2 Sampling with Replacement

In the first 1-d model, I will sample 10000 covariate with replacement within the range $[-2, 12]$ calculated the corresponding response variable, and choose the $\sigma_1 = 0.5$. In the second 2-d model, I will also sample 10000 pairs of (x_1, x_2) in the rectangle from 0 to 6, and calculate response variable. Here, I also will choose $\sigma_2 = 0.5$. In this way, we got the simulated data and ready for next neural networking step. Furthermore, I will use all the points as the train set, then use the sequence from 0 to 6 with distance 0.01 as the test set, then compare the predictive value with the function to evaluate the goodness of fit.

3.Model Setting

Guang-Bin Huang, Qin-Yu Zhu and Chee-Kheong Siew discovered a neural networking model which simplifies the process of getting the weight and intercept within each hidden unit in SLFN model. In this case, we can use random generated weights and intercepts to fit the hidden layer. The model can be expressed as follows: For each input x:

$$h_i = g_i(W_i^T x + c_i)$$

And after calculating the hidden units value h_i:

$$\hat{y} = \omega^T(h_1, h_2, ..., h_k) + b$$

in which, g_i is an activation function, which destroys the linear properties to fit various different type of functions.

As we can see from the formula, if we generate W_i and c_i randomly first, then they can be treated as fixed parameters, then after setting the activation function $g_i(x)$, each h_i is known. Therefore, the only step needing calculation is the last step, which optimizes the ω and b. Since we have known each h_i, the last step can be easily calculated by using the least square estimation which is related to linear models we learn this quarter. I will explore different kinds of distribution function for W_i and c_i

4.Model Fitting

4.1 One-dimensional Case

Since I am also interested in the effect of the number of hidden nodes within the hidden layer, I will explore the case when $k = 10$, $k = 50$, $k = 100$ and $k = 500$. Furthermore, for the activation function, I will first take the Sigmoid function, and then ReLU functions. To check the goodness of fit, I will generate x in an increasing order from -2 to 12 with distance 0.01, and fit the y then check the similarity of the generated function and true function.

4.1.1 Uniformly Generating Hidden Layer Slope and Weights

This is the case that I generated the W_i and c_i by a uniform distribution $U[-1, 1]$. After fitting the model with the sample, the comparisons between the true data generating function and estimated function are as follows in figure 3, in which I used the sigmoid function as the activation. Then in figure 4, I used the ReLU function. It seems that ReLU function didn't perform well when k goes too large even if I used the generalized inverse to make the function smoother. I think it's because the the model is overfitting the data.

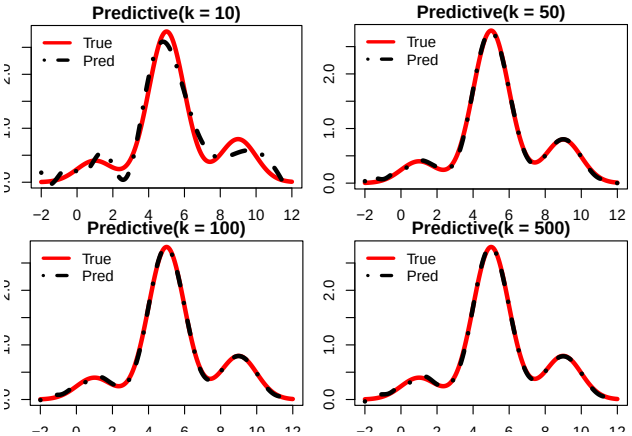

Figure 3: Uniformly Generating(Sigmoid Activation)

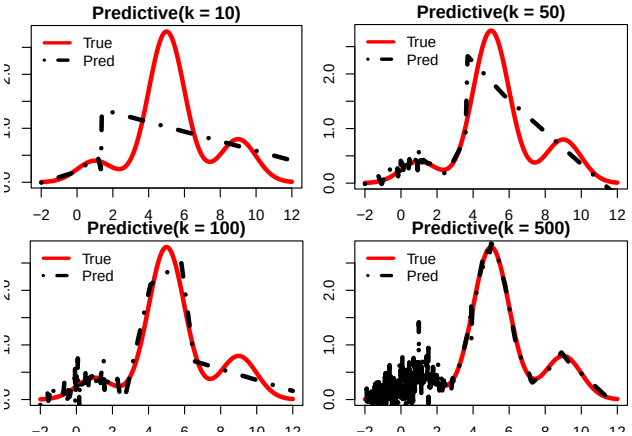

Figure 4: Uniformly Generating(ReLU Activation)

4.1.2 Normally Generating Hidden Layer Slope and Weights

This is the case that I generated the W_i and c_i by standard normal distribution $N(0,1)$. After fitting the model with the sample, the comparisons between the true data generating function and estimated function are as follows in figure 5 for Sigmoid function and 6 for ReLU function.

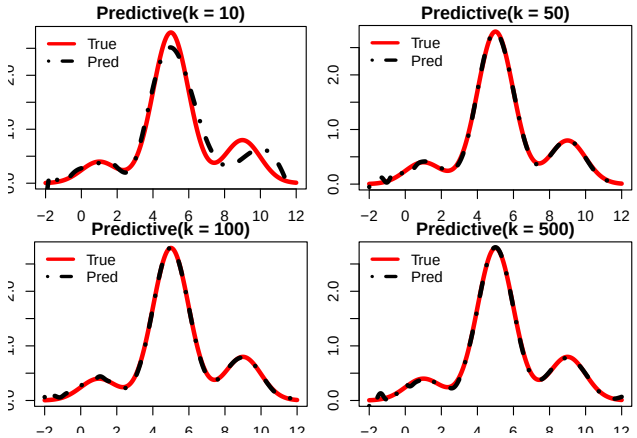

Figure 5: Normally Generating(Sigmoid Activation)

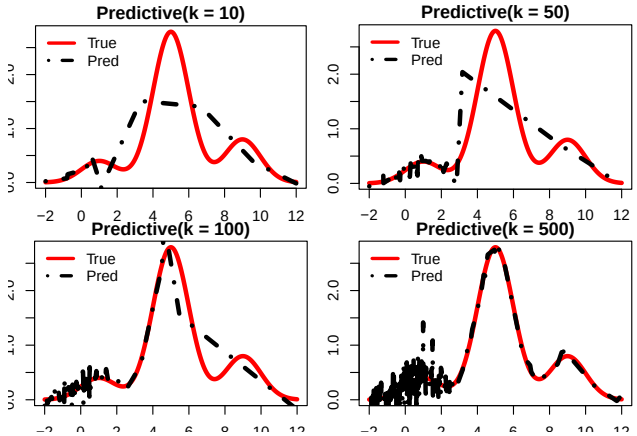

Figure 6: Normally Generating(ReLU Activation)

Here, we can see that there are not many differences whether we generate the weight and intercept by uniform distribution or normal distribution. However, when the number of hidden units are very small, like 10, the model performs poorly. However, if we increase the number of the unit to a very large value, it will have the overfitting problem like k = 500. We can see from the case when $k = 500$, the function becomes more noisy than the others although it fits the train set well. Therefore, setting a proper number of hidden nodes is very important. Another important index to compare them is the sum of the predictive error $\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$, which is shown in table 1 and 2. However, an obvious advantage of sigmoid comparing with ReLU is that sigmoid function is more

Table 1: Prediction Error(Uniform)

	Sigmoid	ReLU
k=10	36.2243230	625.01487
k=50	0.7233566	193.08840
k=100	0.5639920	57.96391
k=500	0.5995596	18.48554

Table 2: Prediction Error(Normal)

	Sigmoid	ReLU
k=10	50.2567256	375.88930
k=50	1.0217606	306.06212
k=100	0.7179009	114.96665
k=500	1.1749112	16.52577

stable and smoother, when predicting smooth curves, it performs better than ReLU function because of a smaller of prediction error.

4.2 Two-dimensional Case

Here, I will consider another two-dimensional function mentioned above. Similarly, I will use $k = 10$, $k = 50$,$k = 100$ and $k = 200$ as the number of the hidden nodes. Also, both uniformly and normally generating are considered as follows. The reason I use 200 is that when k equals 500, my computer would run out of memory.

4.2.1 Uniformly Generating Hidden Layer Slopes and Interceptss

Similarly, this is the case that we use $U[-1,1]$ to generate both slope and intercept. And the predictive plot of this case is in figure 7 with sigmoid activation function and in figure 8 with ReLU function.

4.2.2 Normally Generating Hidden Layer Slopes and Intercepts

Also, here is the predictive performance for the normally generated methods in figure 9 and figure 10. The predictive squared error are shown in table 3 and table 4.

5.Model Comparisons

From the sum of squared prediction error, we can see that for the 1-dimensional case, the sigmoid function performs better tan the ReLU function especially when the target function is a smooth curve instead of straight line. Furthermore, either uniformly or normally generating the

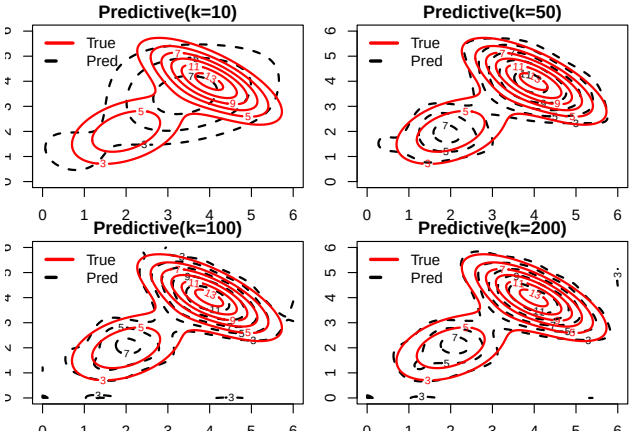

Figure 7: Uniformly Generating(Sigmoid Activation)

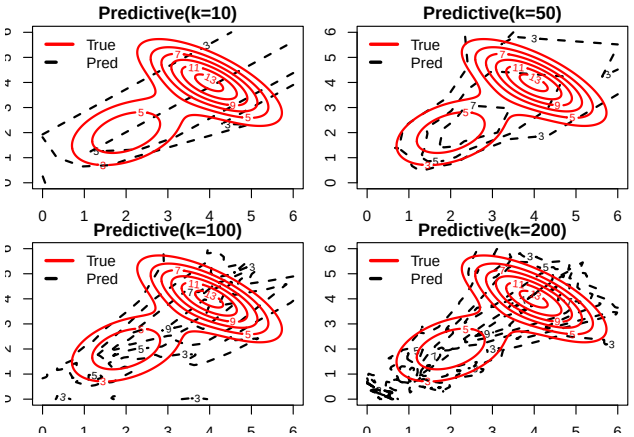

Figure 8: Uniformly Generating(ReLU Activation)

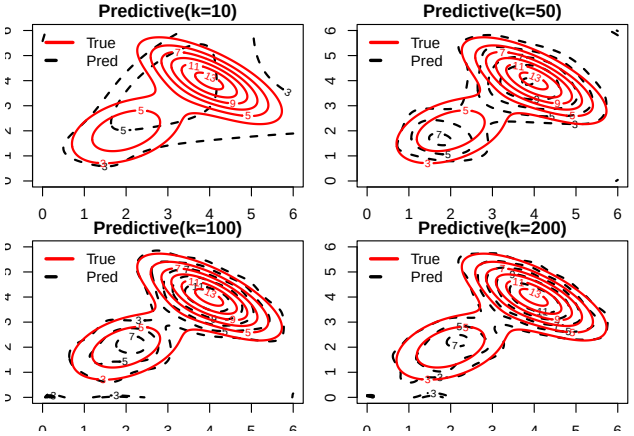

Figure 9: Normally Generating(Sigmoid Activation)

Table 3: 2D Prediction Error(Uniform)

	Sigmoid	ReLU
k=10	1438155.7	2109331
k=50	239359.4	1628295
k=100	135938.8	1332774
k=200	121702.2	2528379

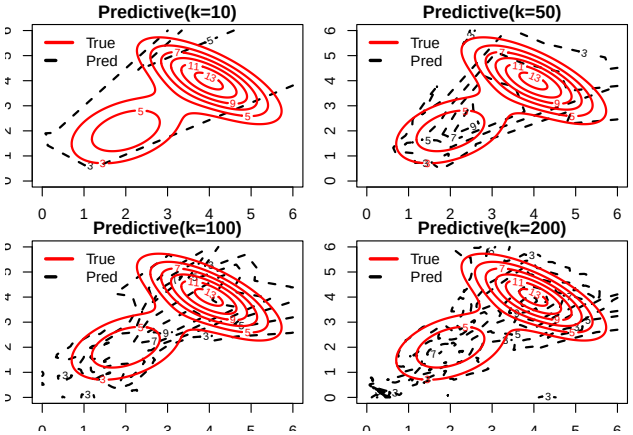

Figure 10: Normally Generating(ReLU Activation)

Table 4: 2D Prediction Error(Normal)

	Sigmoid	ReLU
k=10	1694830.7	2395380.4
k=50	379485.8	1469807.0
k=100	154295.6	1264944.1
k=200	126758.7	943259.2

weights will not affect the prediction result much. But the only difference there is that normal distribution can avoid being overfitted, and uniform distribution can explore the properties of the data better.

Also, the number of hidden nodes is a factor of affecting the prediction performance, too. If too many hidden nodes are included, the function will fall into an overfitting problem, like the case of the ReLU activation function with nodes 500 in the 1d case. For the 2-d case, the sigmoid function also has a better prediction performance based on the prediction error. Sigmoid function is easier to be overfitted in our case since it has performed well when k=50. ReLU function tends to learn the data slower than sigmoid function since it generates too many useless information, like all the columns are generated as 0.

6.Discussion

In my model, I used a *ginv*() function to calculate the generalized inverse of the H matrix, I was thinking of using the *lm*() function to get the regression coefficient to make the prediction. However, some of the coefficient was shown as NA, which indicates they are not informative at all. For example, for a ReLU function, if we randomly generated W and c, cases are possible that we got an extreme value so all the correspondingly hidden node value are same, like 0. That's why sometimes the ReLU function does not perform well. In the future research, I am thinking of using a x^i as hidden

nodes, so we are making a polynomial approximation of the true function (feels like a Taylor's Expansion) of a function, which may makes it easier to handle. For the two-dimensional case, I may choose $x^i y^j$ as hidden nodes to avoid a activation function. However, it could not be called "Neuro Networking" since it does not only include a linear function and a activation function.