Homework3 / Back propagation for functions approximation

李豪韋 (HW-Lee) ID 103061527

Overview

In concepts of linear filtering for line fitting, we know that linear filtering can general learn a model linearly mapping input sets to desired output sets, but cannot deal with non-linear relation very well without lifting functions. (e.g. we cannot make the model infinitely approach the quadratic function without adding x^2 as a feature.) Therefore, an idea of cascading multiple stages of neural layers which contains several neurons as a learning system is suggested. Theoretically, it will work because the model complexity is increased as we apply more neurons or more layers, and this project is aimed at verifying its feasibility and discussing some issues of implementation during this work.

For purposes of easily usability and highly flexibility for programming/simulation, I constructed a simple framework consisting of some objects needed in a neural network. (e.g. Neurons, Neural Layers, Neural Nets, and Learning System) Because of limited pages of the report, only the relatively important(or the most important) part will be described in the report, and the more detailed information of the framework is publicly available on https : //github.com/HW-Lee/2015-NN-Homeworks/tree/master/HW03.

Implementation

1. **Objects and Functions**

   - **Node** is a point with a certain value, used for calling value with reference.
   - **Neuron** is an unit in a neural net with some arithmetic functions and several I/O **Nodes**.
   - **NeuralLayer/NeuralNet** is a container consisting of a couple of **Neurons/NeuralLayers**.
   - **TrainingSystem** is a learning system controlling how to change the weights of a net when training instances are given.

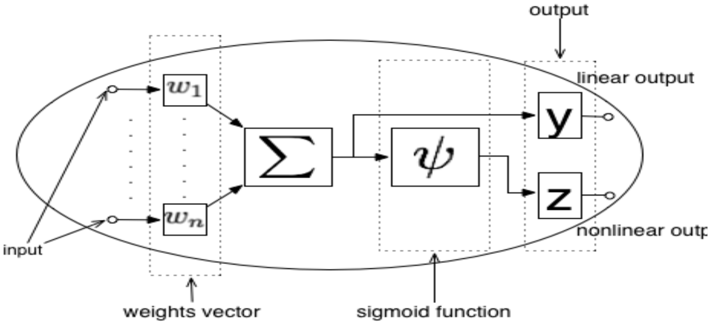

Figure 1: Structure of a Neuron

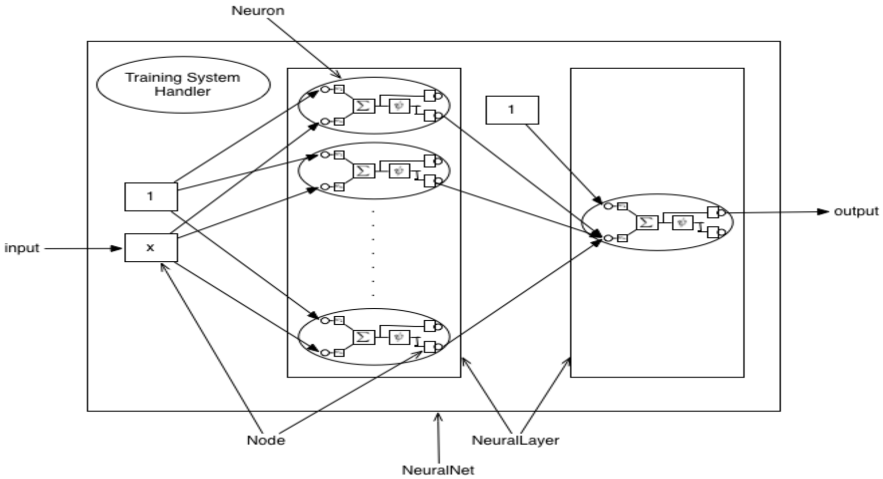

Figure 2: Structure of a NeuralNet with a hidden layer

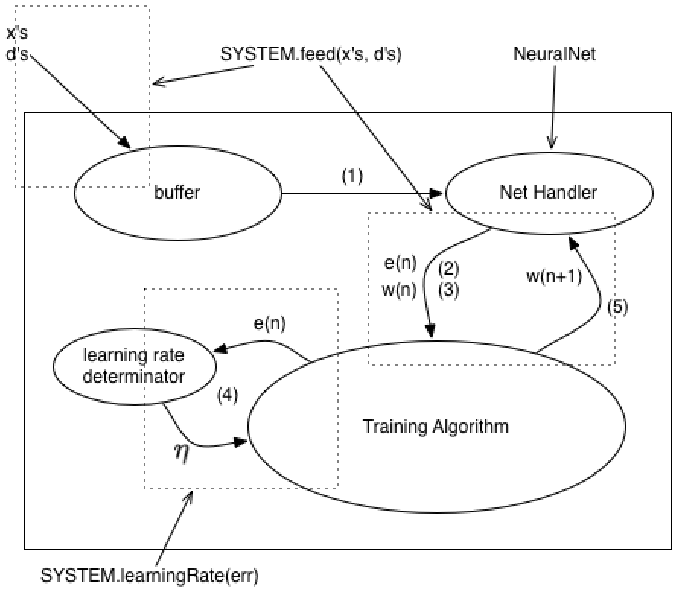

Figure 3: The operation flow between a **Neural Net** and a **Training System**: 1) a number of instances will be stored in the buffer and sent one-by-one, 2) the difference between output value and desired value will be generated, 3) the **Neural Net** and the difference will be sent to algorithm processor, 4) algorithm will get an appropriate learning rate from learning rate determining function with the error, 5) update the new weights of the **Neural Net**, and then 6) repeat the process until the buffer is empty.

2. **Process**

   - **Feed an instance and compute the error**
   - **Compute all local gradients δ's**
   - **Update all weights based on outputs of each stage and local gradients**

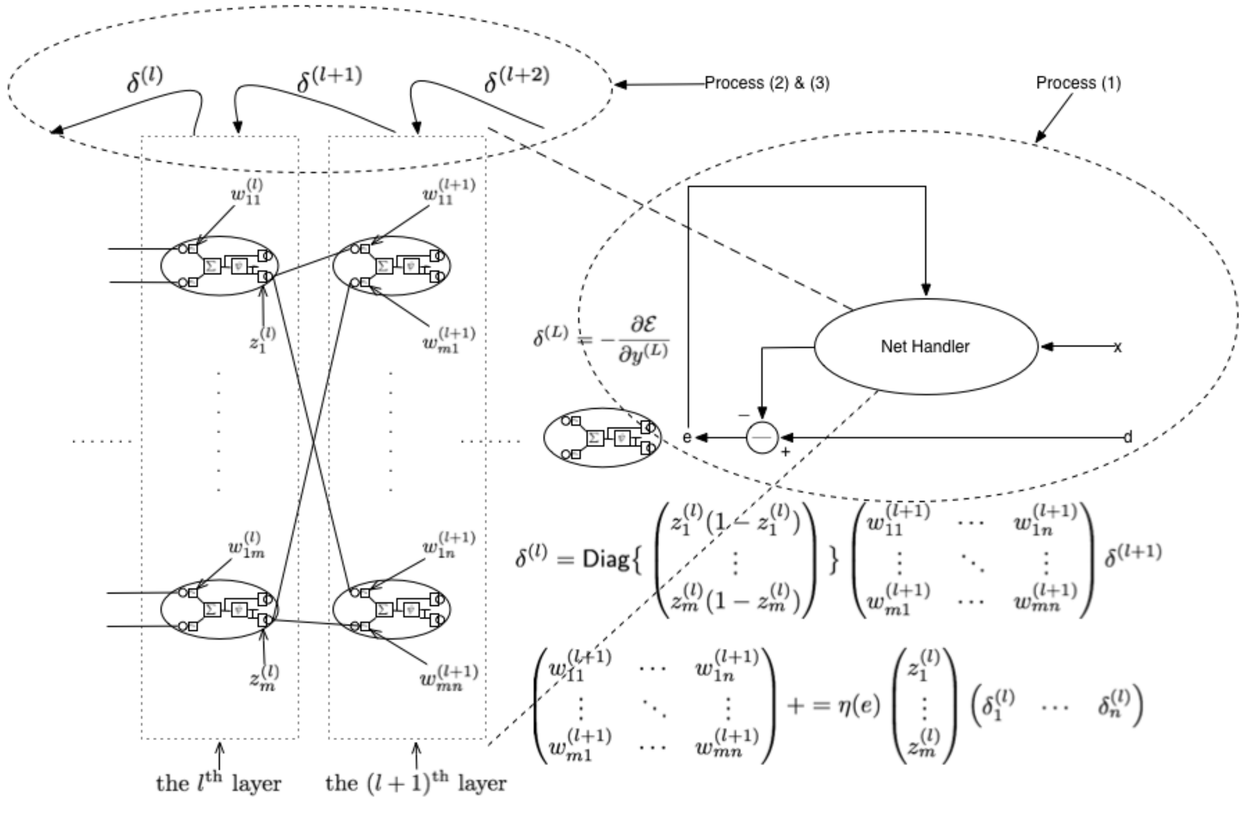
$\delta^{(l)}$
$\delta^{(l+1)}$
$\delta^{(l+2)}$
Process (2) & (3)
Process (1)
$w_{11}^{(l)}$
$w_{11}^{(l+1)}$
$z_1^{(l)}$
$w_{m1}^{(l+1)}$
$\delta^{(L)} = -\frac{\partial \mathcal{E}}{\partial y^{(L)}}$
Net Handler
x
e
−
+
d
$w_{1m}^{(l)}$
$w_{1n}^{(l+1)}$
$z_m^{(l)}$
$w_{mn}^{(l+1)}$
$\delta^{(l)} = \mathsf{Diag}\{\begin{pmatrix} z_1^{(l)}(1-z_1^{(l)}) \\ \vdots \\ z_m^{(l)}(1-z_m^{(l)}) \end{pmatrix}\}\begin{pmatrix} w_{11}^{(l+1)} & \cdots & w_{1n}^{(l+1)} \\ \vdots & \ddots & \vdots \\ w_{m1}^{(l+1)} & \cdots & w_{mn}^{(l+1)} \end{pmatrix}\delta^{(l+1)}$
$\begin{pmatrix} w_{11}^{(l+1)} & \cdots & w_{1n}^{(l+1)} \\ \vdots & \ddots & \vdots \\ w_{m1}^{(l+1)} & \cdots & w_{mn}^{(l+1)} \end{pmatrix} += \eta(e)\begin{pmatrix} z_1^{(l)} \\ \vdots \\ z_m^{(l)} \end{pmatrix}\begin{pmatrix} \delta_1^{(l)} & \cdots & \delta_n^{(l)} \end{pmatrix}$
the l^{th} layer
the $(l+1)^{\text{th}}$ layer

Figure 4: Process of back propagation algorithm: a more detailed demo of "Training Algorithm" block in the Figure 3

Results

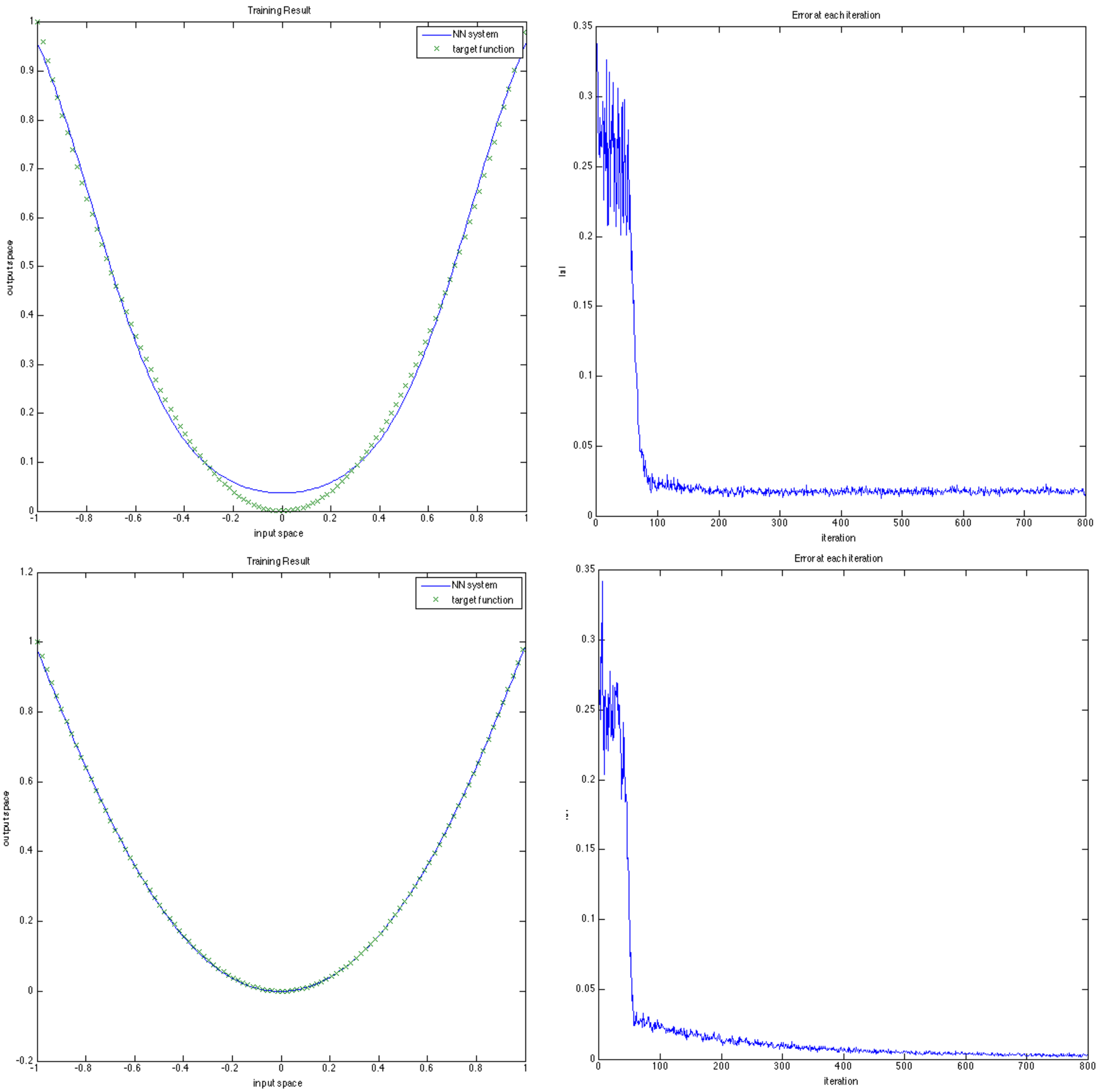

Figure 5: Training results and absolute error at each iteration (50 instances/iteration) with two different number of hidden neurons: 1) 2 hidden neurons upper, and 2) 5 hidden neurons lower

Discussion

1. Does initialization matter?

   In practice, yes. Note that this algorithm CANNOT use zero initialization because the local gradients will be always zero. (see Figure 4)

2. Does randomization matter?

   The randomly selected samples are relatively irrelevant to the result after a sufficiently large iterations. Instead of it, the range of samples is a key reason determining training efficiency.

3. How does the learning rate affect the system's behavior?

   It affects whether the system will finally get converged, how fast it gets converged, and how many iterations it requires to get converged.

4. Does the system always converge to the same set of weights or does the system sometimes 'get stuck' in a sub-optimal region in the weight space?

	$n_1^{(1)}$	$n_2^{(1)}$	$n_3^{(1)}$	$n_4^{(1)}$	$n_5^{(1)}$
b	-1.55 ± 0.90	-1.65 ± 0.90	-0.81 ± 1.81	-0.06 ± 2.67	-1.20 ± 0.78
x	1.16 ± 1.08	1.43 ± 1.11	-0.07 ± 1.92	0.04 ± 2.56	-0.76 ± 0.86
	$w_{11}^{(2)}$	$w_{21}^{(2)}$	$w_{31}^{(2)}$	$w_{41}^{(2)}$	$w_{51}^{(2)}$
	0.63 ± 1.85	1.42 ± 1.22	1.21 ± 1.58	1.64 ± 1.45	0.48 ± 1.47

	$n_1^{(1)}$	$n_2^{(1)}$
b	-4.19 ± 0.01	-4.19 ± 0.01
x	5.38 ± 0.02	-5.39 ± 0.02
	$w_{11}^{(2)}$	$w_{21}^{(2)}$
	1.24 ± 0.02	1.24 ± 0.01

   The above table shows 10 sets of final weights after 800 × 50 instances learning the quadratic function with 5/2 hidden neurons. According to the table, the answer is NOT REALLY.

5. How well does your system learn a different function, say $y = \cos^2(\frac{\pi}{2}x)$, $y = \sin(kx)$, $y = |x|$, and $y = \sqrt{|x|}$?

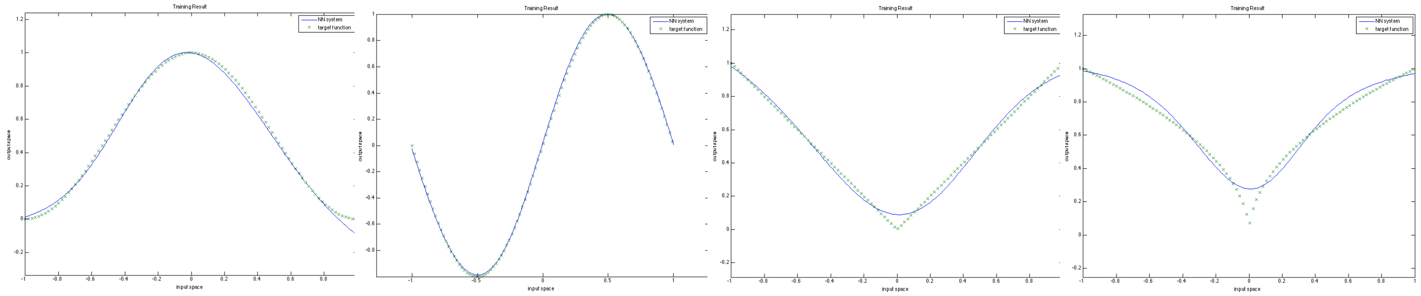

Figure 6: NN system training results of different target functions: $y = \cos^2(\frac{\pi}{2}x)$, $y = \sin(\pi x)$, $y = |x|$, $y = \sqrt{|x|}$ (from left to right)

6. Continuing from Q3, does it help to increase the number of hidden neurons if the system suggested from the assignment sheet does not learn a function well?

   In general, yes. (see Additional Discussion section)

Additional Discussion: Why does it (multi-layer) work?

1. Describe the sigmoid function, says generally **logistic function**, in another way.

$$\phi(x) = \frac{1}{1+e^{-x}} = \lim_{n\to\infty}\sum_{i=0}^{n}\frac{\phi^{(i)}(x_0)}{i!}(x-x_0)^i = \frac{1}{2} + \lim_{n\to\infty}\sum_{i=1}^{n} a_i(x-x_0)^i$$

$$\text{, where } a_i = \frac{\phi^{(i)}(x_0)}{i!},\ \phi^{(n)}(x) = \phi^{(n-1)}(x)(1-2\phi(x)) - \sum_{i=1}^{n-2} C_i^{n-1}\phi^{(n-1-i)}(x)\phi^{(i)}(x)\ \forall\, n > 2$$

And we can also obtain that:

$$\phi^{(n)}(x) = \phi^{(n-1)}(x)(1-2\phi(x)) - \sum_{i=1}^{n-2} C_i^{n-1}\phi^{(n-1-i)}(x)\phi^{(i)}(x) < 2^{n-1} < n! \text{ when } n \text{ is greater than } 2$$

$\Longrightarrow$ If n is sufficiently large, the expansion can even perfectly fit the closed form curve for any x and any x_0. Therefore, we can simply assume $x_0 = 0$ without loss of generality.

2. Under the structure constructed with a hidden layer with m hidden neurons.

For every hidden neuron:

$$z_i^{(1)} = \phi(y_i^{(1)}) = \phi(w_{1i}^{(1)} + w_{2i}^{(1)}x) \approx \frac{1}{2} + \sum_{k=1}^{N} a_k(w_{1i}^{(1)} + w_{2i}^{(1)}x)^k, \text{ for some } N < \infty, \text{ and } x_0 = 0$$

$$= b_{i0} + \sum_{k=1}^{N} b_{ik}x^k = \begin{pmatrix} b_{i0} & \cdots & b_{iN}\end{pmatrix}\begin{pmatrix}1\\ \vdots \\ x^N\end{pmatrix} \Longrightarrow z^{(1)} = \begin{pmatrix} b_{10} & \cdots & b_{1N}\\ \vdots & \ddots & \vdots \\ b_{m0} & \cdots & b_{mN}\end{pmatrix}\begin{pmatrix}1\\ \vdots \\ x^N\end{pmatrix}$$

$$y^{(2)} = z^{(1)T}\begin{pmatrix} w_{11}^{(2)} \\ \vdots \\ w_{m1}^{(2)}\end{pmatrix} \approx \begin{pmatrix}1 & \cdots & x^N\end{pmatrix}\begin{pmatrix}c_0\\ \vdots \\ c_N\end{pmatrix} = \sum_{k=0}^{N} c_k x^k \approx f(x)$$

$$\Longrightarrow \begin{pmatrix} b_{10} & \cdots & b_{m0}\\ \vdots & \ddots & \vdots \\ b_{1N} & \cdots & b_{mN}\end{pmatrix}\begin{pmatrix} w_{11}^{(2)} \\ \vdots \\ w_{m1}^{(2)}\end{pmatrix} \approx \begin{pmatrix}c_0\\ \vdots \\ c_N\end{pmatrix}$$

3. Conclusions and Facts with reasons

- Multi-layer neural net does work because it works as an infinite order polynomial approximation with finite order polynomials.
- Theoretically, the more number of hidden neurons m, the better training performance we can get without considering computation consumption and convergence issues. Because increasing m is equivalently increasing N and reducing the approximation error of the expansion.
- The bias node is redundant in the second stage, so the bias node in the second stage has been removed in this project.
- The bias node is essential in the first stage because the net will only cover odd part (x^{2n-1}) and constant part if $b = 0$. (i.e. $a_i\big|_{x_0=0} = 0$, $\forall\, i$ is even.)
- In Result section, the error seems not to approach to zero because the chosen $N < \infty$. In addition, 5-hidden-neurons net also has lower static error, which proves that increasing m can equivalently increase N and than reduce the static error.