More about Neural Network Training

Part I
Implementing BP algorithm

Implementing BP Algorithm

To implement BP algorithm, we need to create variables for each layer of perceptron:

- the weight matrix $w_{ji}^{(1)}$ and $w_{kj}^{(2)}$,
- the net input vectors: $\text{net}_j^{(1)}$ and $\text{net}_k^{(2)}$
- the output vectors of perceptron,

  $z_j = f(\text{net}_j^{(1)})$ and $y_k = f(\text{net}_k^{(2)})$
- the "error" vectors, δ_{1j} and δ_{2k}

3

(Cont'd)

Step 1: Forward Propagation

Compute the activation for each hidden note, z_j, $i=1,\ldots,M$:

$$net_j^{(1)} = \sum_{i=0}^{d} w_{ji}^{(1)} x_i \quad \text{and} \quad z_j = f_1(net_j^{(1)})$$

Compute the activation for each output node, y_k, $k=1,\ldots,c$:

$$net_k^{(2)} = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \quad \text{and} \quad y_k = f_2(net_k^{(2)})$$

4

(Cont'd)

Step 2: Backward Propagation

Compute error signal for each output node, δ_{2k}, $k=1,\ldots,c$:

$$\delta_{2k} = (d_k - y_k) f_2'(net_k^{(2)})$$

Compute error signal for each hidden node, δ_{1j}, $j=1,\ldots,M$:

$$\delta_{1j} = f_1'(net_j^{(1)}) \sum_{k=1}^{c} w_{kj}^{(2)} \delta_{2k}$$

(Cont'd)

Step 3: Accumulate gradients over the input patterns

$$\frac{\partial E}{\partial w_{kj}^{(2)}} = -\sum_{t=1}^{N} \delta_{2k}(t) z_j(t)$$

$$\frac{\partial E}{\partial w_{ji}^{(1)}} = -\sum_{t=1}^{N} \delta_{1j}(t) x_i(t)$$

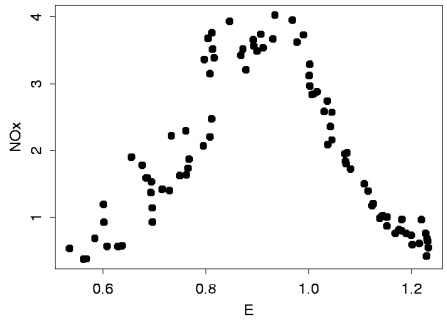
Step 4: After repeat Step 1 to 3 for all patterns, update the weights:

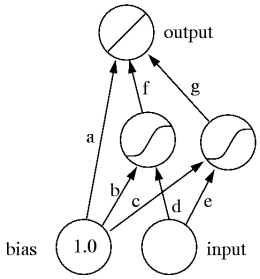
$$w_{kj}^{(2)} \leftarrow w_{kj}^{(2)} - \mu \frac{\partial E}{\partial w_{kj}^{(2)}}$$

$$w_{ji}^{(1)} \leftarrow w_{ji}^{(1)} - \mu \frac{\partial E}{\partial w_{ji}^{(1)}}$$

Backpropagation of Error: An Example

We will now show an example of a backprop network as it learns to model the highly nonlinear data.





The left hand panel shows the data to be modeled. The right hand panel shows a network with two hidden units, each with a tanh nonlinear activation function.

7

(Cont'd)

The output unit computes a linear combination of the two functions

$$y_0 = f \cdot \tanh_1(x) + g \cdot \tanh_2(x) + a \qquad (1)$$

where

$$\tanh_1(x) = \tanh(dx + b) \qquad (2)$$

and

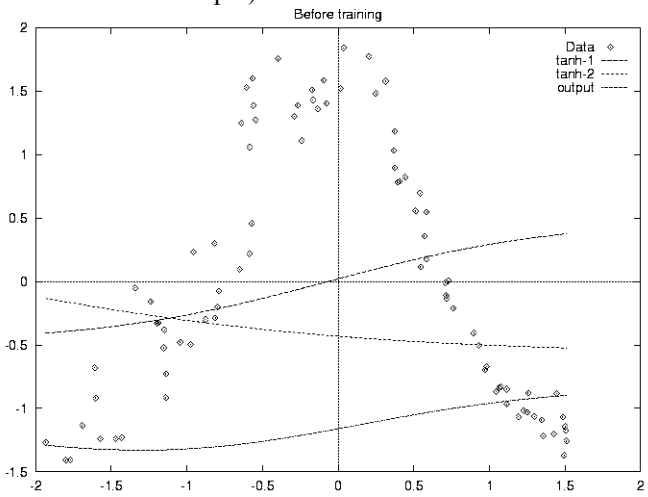
$$\tanh_2(x) = \tanh(ex + c) \qquad (3)$$

To begin with, we set the weights, a,b,c,d,e,g, to random initial values in the range [-1,1]. Each hidden unit is thus computing a random *tanh* function.

8

(Cont'd)

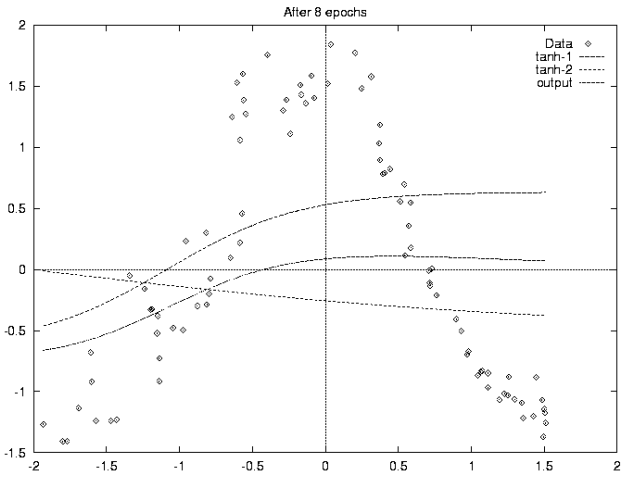
The next figure shows the initial two activation functions and the output of the network, which is their sum plus a negative constant. (If you have difficulty making out the line types, the top two curves are the *tanh* functions, the one at the bottom is the network output).



9

(Cont'd)

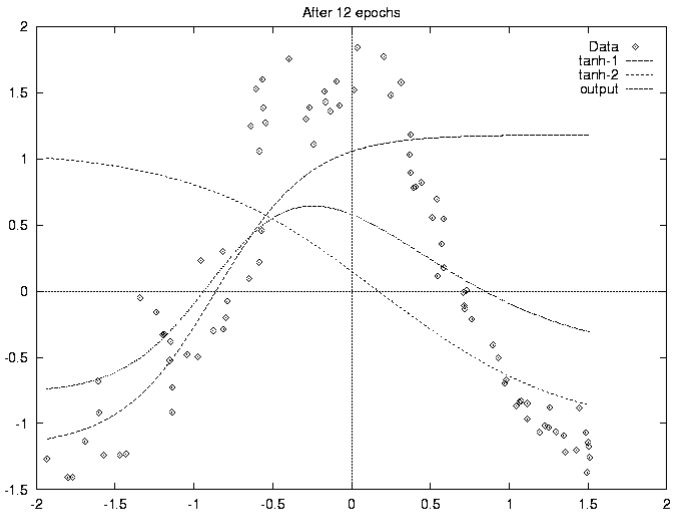
We now train the network (learning rate 0.3), updating the weights after each pattern (online learning). After we have been through the entire dataset 10 times (10 training epochs), the functions computed look like this (the output is the middle curve):



10

(Cont'd)

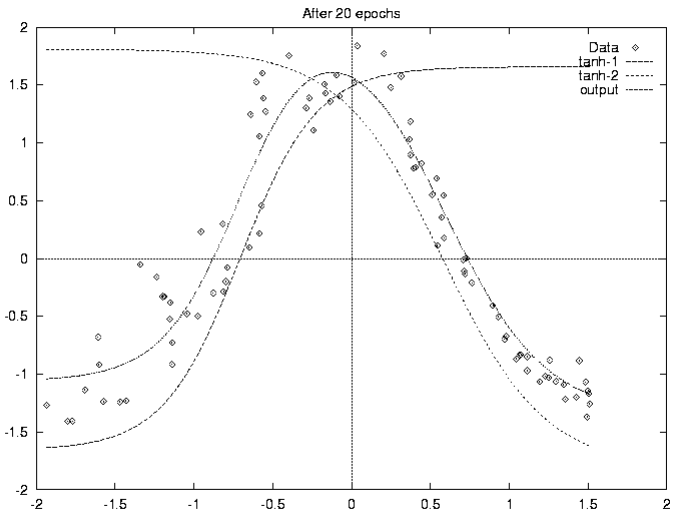
After 20 epochs, we have (output is the humpbacked curve):



11

(Cont'd)

and after 27 epochs we have a pretty good fit to the data:

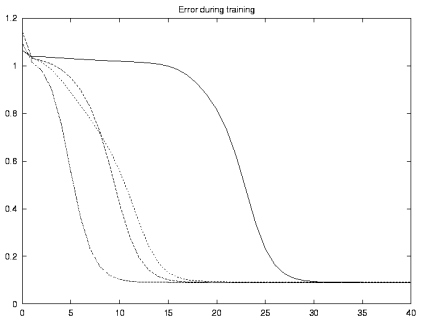


12

(Cont'd)

As the activation functions are stretched, scaled and shifted by the changing weights, we hope that the error of the model is dropping. In the next figure we plot the total sum squared error over all 88 patterns of the data as a function of training epoch. Four training runs are shown, with different weight initialization each time:

You can see that the path to the solution differs each time, both because we start from a different point in weight space, and because the order in which patterns are presented is random. Nonetheless, all training curves go down monotonically, and all reach about the same level of overall error.



13

Demonstrations

- Network function (nnd11nf)
- Backpropagation calculation (nnd11bc)
- Function approximation (nnd11fa)

14

Part II
More on implementation of BP algorithm

Cost function

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} loss(y^{(i)}, f(x^{(i)};\theta)$$

We need to find θ that minimizes the cost function:

$$\theta = \arg\min_{\theta}\{J(\theta\}$$

Neural Network Regression

Neural Network regression has no activation function at the output layer.

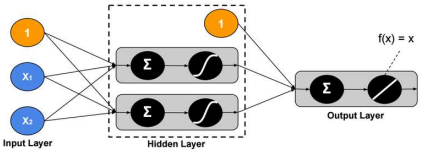
- L1 Loss function

$$loss(y, \hat{y}) = | y - \hat{y} |$$

- L2 Loss function

$$loss(y, \hat{y}) = (y - \hat{y})^2$$

- Hinge loss function



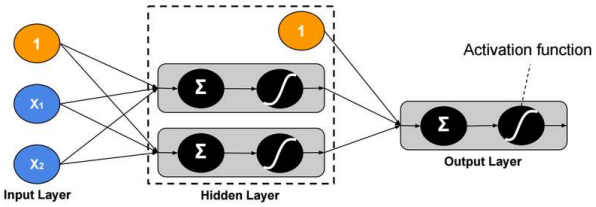
Hinge loss function is recommended when there are some outliers in the data.

$$loss(y, \hat{y}) = \max(0, | y - \hat{y} | - m)$$

(1) (2) (3)

Which corresponds to L1 loss ? L2 loss, and Hinge loss?

Two-Class Neural Network



- Binary Cross Entropy Loss function

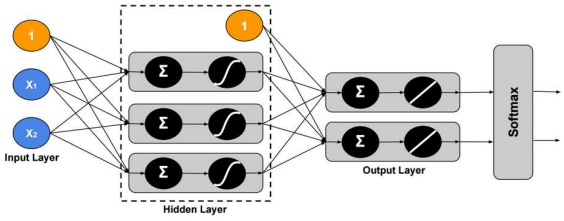
$$loss(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

- Hinge loss (SVM)

$$\ell(y) = \max(0, 1 - t \cdot y)$$

Multi-Class Neural Network – One-Task

Using Softmax, the output ŷ is modeled as a probability distribution, therefore we can assign only one label to each example.



- Binary Cross Entropy Loss function

$$loss(Y,\hat{Y}) = -\sum_{j=1}^{c} Y_j \log(\hat{Y}_j)$$

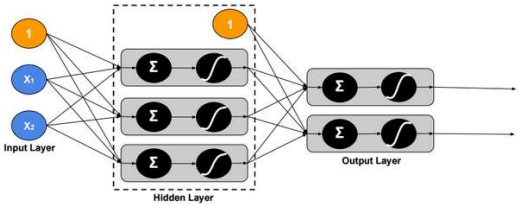
- Hinge Loss (SVM) function

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \hat{y} = \begin{bmatrix} 2 \\ -5 \\ 3 \end{bmatrix}$$

$$loss(y,\hat{y}) = \sum_{c \neq 1} \max(0, \hat{y}_c - \hat{y} + m)$$

Multi-Class Neural Network – Multi-Task

In this version, we assign multiple labels to each example.



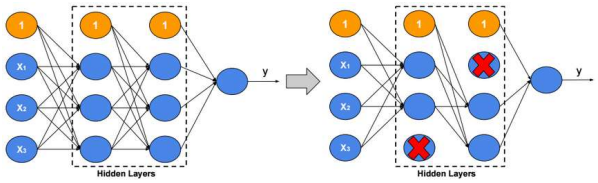
- Loss function

$$loss(Y,\hat{Y}) = \sum_{j=1}^{c} \left(-Y_j \log(\hat{Y}_j) - (1 - Y_j) \log(1 - \hat{Y}_j) \right)$$

Regularization

Regularization is a very important technique to prevent overfitting.

- Dropout



For each training example, ignore randomly p x 100% activation nodes of each hidden layer. p is called dropout rate ($p \in [0,1]$). When testing, scale activations by the dropout rate p.

Why dropout works ?

- Inverted Dropout

With inverted dropout, scaling is applied at the training time. First, dropout all activations by dropout factor p, and second, scale them by inverse dropout factor 1/(1-p)=1/q. Nothing needs to be applied at test time.

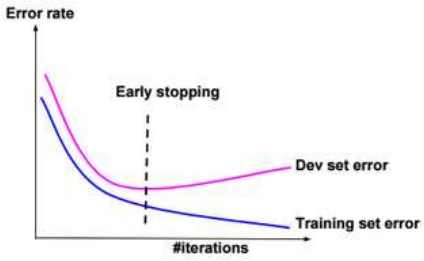
$$\textbf{Train phase: } O_i = \frac{1}{q} X_i a\left(\sum_{k=1}^{d_i} w_k x_k + b\right)$$

$$\textbf{Test phase: } O_i = a\left(\sum_{k=1}^{d_i} w_k x_k + b\right)$$

- Data Augmentation

As a regularization technique, we can apply random transformations on input images when training a model.

- Early Stopping



Stop when error rates decreases on training data while it increases on dev (cross-validation) data.

- L1 Regularization

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} loss(\mathrm{y}^{(i)}, f(x^{(i)};\theta) + \lambda \sum_{j} |\theta_j|$$

λ is called regularization parameter

- L2 Regularization

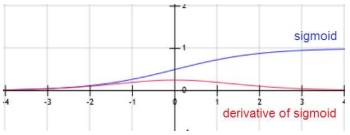
$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} loss(\mathrm{y}^{(i)}, f(x^{(i)};\theta) + \lambda \sum_{j} \theta_j^2$$

- Lp Regularization

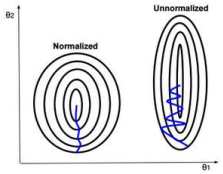
$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} loss(\mathrm{y}^{(i)}, f(x^{(i)};\theta) + \lambda \sum_{j} \theta_j^p$$

Normalization

Gradient descent converges quickly when data is normalized Xi ∈ [-1,1]. If features have different scales, then the update of parameters will not be in the same scale (zig-zag).



For example, if the activation function g is the sigmoid function, then when W.x+b is large g(W.x+b) is around 1, but the derivative of the sigmoid function is around zero. For this reason the gradient converges slowly when the W.x+b is large.



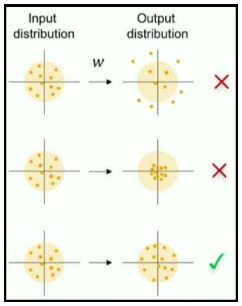
Below some normalization functions

- ZScore $X := \frac{X - \mu}{\sigma}$
- MinMax $X := \frac{X - \min}{\max - \min}$
- Logistic $X := \frac{1}{1 + \exp(-X)}$
- LogNormal $X := \frac{1}{\sigma\sqrt{2\pi}} \int_0^x \frac{\exp(\frac{-(\ln(t) - \mu)^2}{2\sigma^2})}{t} dt$
- Tanh $X := \frac{1 - \exp(-X)}{1 + \exp(-X)}$

Weight Initialization

Weight initialization is important because if weights are too big then activations explode. If weights are too small then gradients will be around zero (no learning).

When we normalize input data, we make the mean of the input features equals to zero, and the variance equals to one. To keep the activation units normalized too, we can initialize the weights $W^{(1)}$ so $Var(g(W^{(1)}j.x+b^{(1)}j))$ is equals to one.



Hyperparameters

- Learning rate (α) (e.g. 0.1, 0.01, 0.001,…)
- Number of hidden units
- Number of layers
- Mini-batch size
- Momentum rate (e.g. 0.9)
- Adam optimization parameters (e.g. β 1=0.9, β 2=0.999, ϵ=0.00000001)
- Learning rate decay

Local Minimum

The probability that gradient descent gets stuck in a local minimum in a high dimensional space is extremely low. We could have a saddle point, but it's rare to have a local minimum.

Transfer Learning

Transfer Learning consists in the use of parameters of a trained model when training new hidden layers of an extended version of that model.