Intuition

Neural Networks are great for estimating behaviour of unknown input on base of training data. They can be thought of as Linear (or Logistic) Regression with adaptiv feature selection, what makes them more suitable to describe the problem. Structure and adaptivness of Neural Networks resemble the human brain, therefore they are often compared to processes running in our brains.

This flyer will introduce you to working with Neural Networks and coding the fundamentals in Octave. With this you will be able to run more sophisticated algorithms using e.g. Octave's Neural Network Package [1]. Besides the following topics are covered:

- Understanding the network architecture
- Calculating results with Forward Propagation
- Setting up cost functions for multiclass networks
- Calculating weights with Back Propagation
- Initialization and suggested defaults

Neural Networks are a way of supervised learning (i.e. learning from training data with known result), as is Linear (or Logistic) Regression. To get the whole picture we encourage you to also read the Linear Regression Flyer. Choosing one over the other is mainly a question of computational cost and accuracy. While Regression is cheaper, Networks are often more precise. For further reading on Neural Networks you can use Google or one of these sites as a starting point:

- Wikipedia [2]
- Machine Learning @ The Standford OpenClassroom [3]
- NN FAQ from comp.ai.neural-nets [4]

fsadmin@flyered-science.org

[1] http://www.plexso.com/61_octave/index.html
[2] http://en.wikipedia.org/wiki/Artificial_neural_network
[3] http://openclassroom.stanford.edu
[4] ftp://ftp.sas.com/pub/neural/FAQ.html

Numerical Solution (in Octave)

Like in Linear Regression we will use a numerical solver to compute $\Theta^{(l)}$. We already defined the cost function and the gradients needed, to pass it along with an inital guess of parameters to a suitable linear solver.

The procedure needs the cost function parameters to be vectors. Therefore the weight matrices $Theta_{ji}^{(l)}$ have to be "unrolled" to a vector, to pass it along, and then "reshaped" again before calculating cost and gradient:

```
ThetaVec = [Theta1(:); Theta2(:); ...

function [J, grad] = neuralCF(ThetaVec...
  Theta1 = reshape(ThetaVec(1:sizeL1* ...
           (sizeL0+1),sizeL1,sizeL0+1)
  (...)
endfunction
```

With an initial $Theta_{ji}^{(l)}$ you calculate the forward propagation $a_l = sigmoid(a_{l-1}\Theta_l')$. (With optimized $\Theta^{(l)}$ this is how the network is used later on.

Cost function and gradients are then summed over all test samples, as explained before. The gradient sum in Octave can be written as follows:

```
%compute backpropagation
delta_output = a2(counter,:)-yvec;
delta_hidden = (delta_output*Theta2) ...
       .* ((a1 .* (1-a1))(counter,:));

%sum gradients
T1grad = T1grad+(delta_hidden(2:end)' ...
             .* X(counter,:));
T2grad = T2grad+(delta_output' ...
             .* a1(counter,:));
```

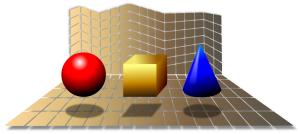
One can the optimize Θ using the cost function with fminunc as depicted in Linear Regression flyer (Math.1).

Flyered Science
Knowledge factsheets for the everyday scientist

Neural Networks

An introduction to representation and algorithm, with implementation in Octave

The Flyered Science Series provides condensed factsheets on scientific topics. Persons new to the subject will find the intuitive introduction with links to further reading quite useful, while the experienced user benefits from the structured representation, formulary and implementation examples. Fits into every pocket, serves perfectly as bookmark in your textbooks.



Theory

Forward Propagation

Neural networks are used to estimate the result $y = h_(\Theta)(x)$ of unknown x on base of trainig data $x^{(i)}$ with known result $y^{(i)}$. The problem is comparable to linear (or logistic) regression. Instead of calculating the result y directly from features x, the neural network introduces (hidden) layers with adaptiv features (activations) $a_j^{(l)}$, that are calculated from the preceeding layer using the weight matrix $\Theta^{(l-1)}$.

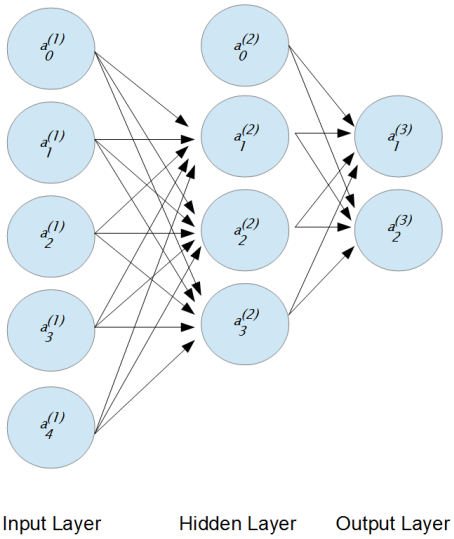


Figure 1: Neural network layout with one hidden layer.

Input and hidden layer(s) can contain a constant offset (bias term) $a_0^{(l)} = 1$. Given L different layers (i.e. $L-2$ hidden layers) the output layer hypothesis $h_(\Theta) = a^{(L)}$ is calculated from input activations $a_i^{(0)} = x_i$ considering the weight matrix Θ and the activation formular $g(\Theta^T a)$. This is called "Forward Propagation".

In this flyer we use the sigmoid function as activation formular, i.e. $g(\Theta^T a) = \frac{1}{1+exp(-\Theta^T a)}$. This is calles "Logistic Unit". Using a "Linear Unit" (i.e. weighted sum), a step function or a linear combination is straight forward.

Weight Matrices

We now discuss how to derive the weight matrix Θ from a training set of labeled data (known result) $(x^{(i)}, y^{(i)})$. Suppose that activation $a_i^{(l-1)}$ in layer $l-1$ contributes to activation $a_j^{(l)}$ in layer l with the weight $\Theta_{ji}^{(l-1)}$. This yields $a_j^{(l)} = g((\Theta_j^{(l-1)})^T a^{(l-1)})$, as shown below.

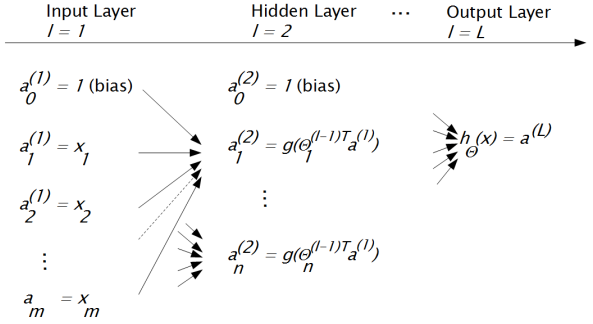


Figure 2: Calculating activations in neural networks.

Like in logistic regression (see also flyer Math.1) a cost function is calculated for the training set, that considers results $y^{(i)}$ and hypothesis $h_(\Theta)(x^{(i)})$ from Forward Propagation. With m training sets and K classes in the output layer the cost function J then becomes

$$\begin{aligned} J(\Theta) = &-\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(h_(\Theta)(x^{(i)}))_k \\ &+ (1-y_k^{(i)}) log(1-h_(\Theta)(x^{(i)})) \\ &+ \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{n_i}\sum_{j=1}^{n_j+1}(\Theta_{ji}^{(l)})^2 \end{aligned}$$

The first term applies for results $y^{(i)} = 1$ and is zero, when the hypothesis matches the result. The second term applies for results $y^{(i)} = 0$ in the same manner. The third term is called the regularization term, which prevents overfitting (i.e. small Θs are preferred, see also flyer Math.1).

The objectiv is to solve $\max_{\Theta} J(\Theta)$ for an optimal set of weight matrices $\Theta^{(l)}$. This is done by "Backpropagation" of the deviations between result and hypothesis.

Backpropagation

The idea of Backpropagation is to consider deviations of the trained network hypothesis from the real result. To find the optimal set of $\Theta^{(l)}$, the deviations have to be minimized. In the output layer L the deviation is calculated as

$$\delta_j^{(L)} = a_j^{(L)} - y_j$$

The deviation can be propagated back through the network using the following equation:

$$\delta^{(L-1)} = (\Theta^{(L-1)})^T \delta^{(L)} .* g'(\Theta^{(L-1)})^T a^{(L-1)})$$

The $.*$ operator means elementwise multiplication. Without considering regularization (i.e. $\lambda = 0$) the deviation of the activation function g can be written as

$$g'(\Theta^{(L-1)})^T a^{(L-1)}) = a^{(L-1)} .* (1 - a^{(L-1)})$$

The task is to compute all $\delta^{(l)}$. Remember: The input parameters in the input layer have zero deviation: $\delta^{(0)} = 0$. Considering only one test example the gradients can be computed from this:

$$\frac{\partial}{\partial \Theta_{ji}^{(l)}} J(\Theta) = a_j^{(l)} \delta_j^{(l+1)}$$

Considering the whole test set, the deviations are then the sum of all test set contributions:

$$\Delta_{ji}^{(l)} = \sum_{i=1}^{m} a_i^{(l)} \delta_j^{(l+1)}$$

The gradient calculates accordingly:

$$\begin{aligned} \frac{\partial}{\partial \Theta_{ji}^{(l)}} J(\Theta) &= \frac{1}{m}\Delta_{ji}^{(l)} + \lambda \Theta_{ji}^{(l)} \qquad for j \neq 0 \\ &= \frac{1}{m}\Delta_{ji}^{(l)} \qquad for j = 0 \end{aligned}$$

(Remember: The bias terms $j = 0$ are not included in regularization). Following the gradients using an optimization algorithm then yields the optimal set of $\Theta^{(l)}$ for the given training set.