ICS273A: Machine Learning **Winter 2008**

Lecture 8 — February 4

Scribe: Carlos Agell (Student) *Lecturer: Deva Ramanan*

8.1 Neural Nets

8.1.1 Logistic Regression

Recall the logistic function:

$$g(x) = \frac{1}{1 + e^{-\theta^T x}} \tag{8.1}$$

where $x \in \{0, 1\}^n$. Assume that we are trying to solve problems such as Digit Classification (from a grid of yes/no or 1/0 values, as in figure 8.1)

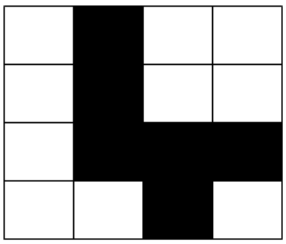

Figure 8.1. Example of digit recognition, kind of problem we are trying to solve.

But, what is the space of representable hypotheses?
Consider $x \in \{0, 1\}^2$ and an arbitrary boolean function y

$$y(x) = (x_1 \wedge x_2 \wedge \overline{x_3}) \cup (x_2 \wedge \overline{x_4} \ldots) \ldots \tag{8.2}$$

We can examine the AND ($\wedge$) and OR ($\cup$) operations, and consider whether they can be represented by a linear decision boundary. Following 8.2, we can see they can be. By similar inspection, we can determine that the XOR ($\oplus$) cannot be represented by a linear decision boundary.

However, let us think of a hierarchy of classifiers, where the output of classifiers are used as features (inputs) of other classifiers. For example, one can represent an XOR as $z_1 \cup z_2$ where $z_1 = x_1 \wedge \bar{x_2}$ and $z_2 = \bar{x_1} \wedge x_2$. This naturally leads to **Neural Nets**

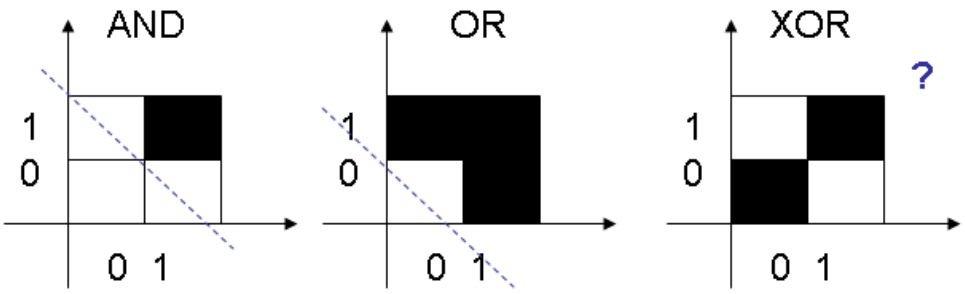

Figure 8.2. AND ($\wedge$) and OR ($\cup$) are linearly separable while XOR ($\oplus$) is not.

8.1.2 Definition of Neural Net

A logistic regression model can be interpreted as a single layer Neural Net. We will write the final output as a nonlinearity g applied to a linear combination of input features.

$$h(x) = g(\sum_i w_i x_i) \tag{8.3}$$

where $g(x)$ is the logistic function defined in equation 8.1. Its graphical representation is shown in figure 8.3

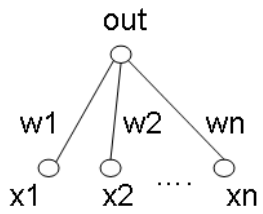

Figure 8.3. Single layer Neural Network described in equation 8.3.

We can also consider multiple layers of this network, which allows us to deal with non-linearities.

Let us consider a 2-layer Neural Net (see figure 8.4), where, by convention w_{ji} corresponds to the weight between the higher node j and a lower node i. The output will be given by:

$$out = g\left(\sum_{j=1}^{3} w_j z_j\right) = g\left(\sum_{j=1}^{3} w_j g\left(\sum_{i=1}^{2} w_{ji} x_i\right)\right) \tag{8.4}$$

because

$$z_j = g\left(\sum_{i=1}^{2} w_{ji} x_i\right) \tag{8.5}$$

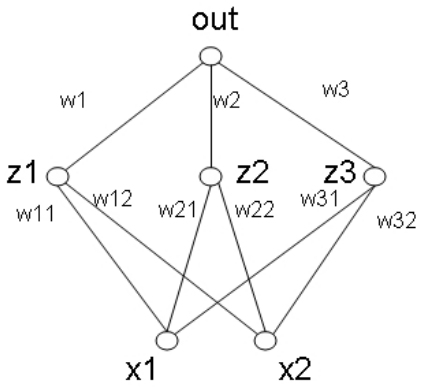

Figure 8.4. Example of 2-layer Neural Net.

8.1.3 Training a Neural Net

Let us define the following *squared error* cost function.

$$J(w) = \frac{1}{2}\sum_i \left(y^{(i)} - out(x^{(i)})\right)^2 \tag{8.6}$$

Although it has no formal probabilistic interpretation, we treat it as a objective function that we wish to minimize.

One option is to use regular gradient descent to train w:

$$w := w - \alpha \frac{\partial J(w)}{\partial w} \tag{8.7}$$

We can use the **backpropogation** algorithm to efficiently compute $\frac{\partial J(w)}{\partial w}$.

We can also perform a regular Newton-Raphson update, following the standard formula:

$$w := w - \left[\frac{\partial^2 J(w)}{\partial w^2}\right] .^2 \frac{\partial J(w)}{\partial w} \tag{8.8}$$

8.1.4 Advantages and Disadvantages

Advantages

- It can represent nonlinear boundaries
- Fast "feed forward" architecture

Fact: Any boolean function can be represented with a 2-layer neural net. [But it can have an arbitrarily large number of nodes.]

Disadvantages

- Gradient descent is not guaranteed to reach a global optimum.

- We do not know the optimal number and size of layers.

Computing the gradient update requires roughly $O(n)$ operations, where n is the number of weights in the neural net. Furthermore, it can also be implemented in an stochastic online fashion by updating w after examining each training point.

Computing the Hessian roughly takes $O(n^2)$ operations (and inverting an $n \times n$ matrix requires $O(n^3)$ operations), so it can get expensive with a large number of weights. In practice, Newton-Raphson updates work best when we are in the *quadratic bowl* around a local minimum (rather than a plateau).

There are many combinations of these two approaches:

- Line search: changes the step size of the gradient seeking for the minimum of the function among the steps.
- Conjugate Gradient

Consult your favorite optimization textbook for more information - but in most cases, these algorithms can be seen as approximations to the full Newton-Raphson update.

8.1.5 Backpropogation

We will derive the backpropogation equations for (8.6). We will consider the gradient due to a particular training example (for say, the online gradient descent setting) and drop the $^{(i)}$ notation.

$$\frac{\partial J}{\partial w} = (y - out(x)) \left[\frac{\partial out(x)}{\partial w} \right] \tag{8.9}$$

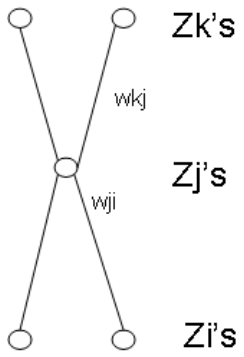

Figure 8.5. Part of the Neural Network for Notation explanation.

Notation: We define the *activation* a_j of a node j to the the linear combination of input features before the nonlinearity is applied:

$$a_j = \sum_i w_{ji} z_i \tag{8.10}$$

$$z_j = g(a_j) \tag{8.11}$$

We call $g(a)$ the *activation function*. For our purposes, we will generally use the logistic function (eq 8.1) as $g(x)$ but we could use other "squashing functions". Another common choice is $g(x) = tanh(x)$.

Using the chain rule, and defining $\delta_j \equiv \frac{\partial out}{\partial a_j}$ we have:

$$\frac{\partial out}{\partial w_{ji}} = \frac{\partial out}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i \tag{8.12}$$

Therefore we will need to:

- Calculate the z_i's with a *forward* step. This is equivalent to applying the neural net to the current x input, with the current weights $\{w\}$ and computing all the hidden node values, from the bottom layer up.
- Calculate the δ_j's. We will show below that one can do this recursively from the top layers down to the bottom layers - the *backpropogation* step.

Recall the chain rule for partial derivatives:

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial w} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial w} \tag{8.13}$$

This can be verified with a simple example: $f(x, y) = xy; x = 2w; y = 3w$

Then:

$$\delta_j = \frac{\partial out}{\partial a_j} = \sum_k \frac{\partial out}{\partial a_k}\frac{\partial a_k}{\partial a_j} = \sum_k \delta_k w_{kj}\frac{\partial z_j}{\partial a_j} = g'(a_j)\sum_k \delta_k w_{kj} \tag{8.14}$$

where $g'(a) = g(a)(1 - g(a))$ for the sigmoid activation function. To initialize this recursion, we can compute

$$\delta_{out} = \frac{\partial out}{\partial a_{out}} = g'(a_{out}) \tag{8.15}$$

8.1.6 Comments on Neural Nets

- What if $g(x) = x$? The final output will be linear!
- Why not make $g(x) = I\{x > 0\}$ (a step function)? The output is not differentiable, and so gradient descent or Newton Raphson become difficult to apply.

- We can apply other loss functions besides squared error. Recall that logistic regression was derived with the cross-entropy loss (eq 8.16). This might be a more reasonable cost function when $y^{(i)} \in \{0, 1\}$. For multiclass classification, one can also use a multi-class cross-entropy loss (8.17) when the output of the neural net is a softmax function(8.18).

$$J(w) = -\sum_i \log\left(out(x^{(i)})^{y^{(i)}}(1 - out(x^{(i)}))^{1-y^{(i)}}\right) \tag{8.16}$$

$$J(w) = -\sum_{i=1}^{m}\sum_{k=1}^{9} y_k^{(i)} log\left(out_k(x^{(i)})\right) \tag{8.17}$$

$$out_k(x^{(i)}) = \frac{\exp\left(a_{out,k}\right)}{\sum_j \exp(a_{out,k})} \tag{8.18}$$