Lecture 36 - Backpropagation continued; Best Practices for Training ANNs

Error Backpropagation

- The learning procedure involves the presentation of a set of pairs of input and output patterns, $X = \{x_i\}_{i=1}^N$ and $Y = \{y_j\}_{j=1}^M$. The system uses the input vector to produce its own output vector and then compares this with the \emph{desire output}, or \emph{target output} $t = \{t_j\}_{j=1}^M$. If there is no difference, no learning takes place. Otherwise, the weights are changed to reduce the difference. This procedure is basically the perceptron learning algorithm.

- This procedure can be *automated* by the machine itself, without any outside help, if we provide some **feedback** to the machine on how it is doing. The feedback comes in the form of the definition of an *error criterion* or *objective function* that must be *minimized* (e.g. Mean Squared Error). For each training pattern we can define an error (ϵ_k) between the desired response (d_k) and the actual output (y_k). Note that when the error is zero, the machine output is equal to the desired response. This learning mechanism is called **(error) backpropagation** (or **BP**).

- The backpropagation algorithm consists of two phases:

  - **Forward phase:** computes the *functional signal*, feed-forward propagation of input pattern signals through the network.
    - **Backward phase:** computes the *error signal*, propagates the error backwards through the network starting at the output units (where the error is the difference between desired and predicted output values).

- **Objective function/Error Criterion:** there are many possible definitions of the error, but commonly in neuro-computing one uses the error variance (or power):

$$J(w) = \frac{1}{2}\sum_{k=1}^{N} \epsilon^2 = \frac{1}{2}\sum_{k=1}^{N}(d_k - y_k)^2 = \frac{1}{2}\sum_{k=1}^{N}(d_k - w^T x_k)^2$$

- Now we need to define an **adaptive learning** algorithm. Backpropagation commonly uses the gradient descent as the adaptive learning algorithm.

- **Adaptive Learning Algorithm:** there are many learning algorithms, the most common is the method of Gradient/Steepest Descent.

  - Move in direction opposite to the gradient, $\nabla J(\mathbf{w})$, vector (**gradient descent**):

$$w^{(n+1)} = w^{(n)} + \Delta w^{(n)}$$

This is known as the **error correction rule**. We define:

$$\Delta w^{(n)} = w^{(n)} - w^{(n-1)}$$
$$\Delta w^{(n)} = -\eta \nabla J(w^{(n)})$$

where η is the learning rate.

Backpropagation of the Error for the Output Layer

There are many approaches to train a neural network. One of the most commonly used is the **Error Backpropagation Algorithm**.

Let's first consider the output layer:

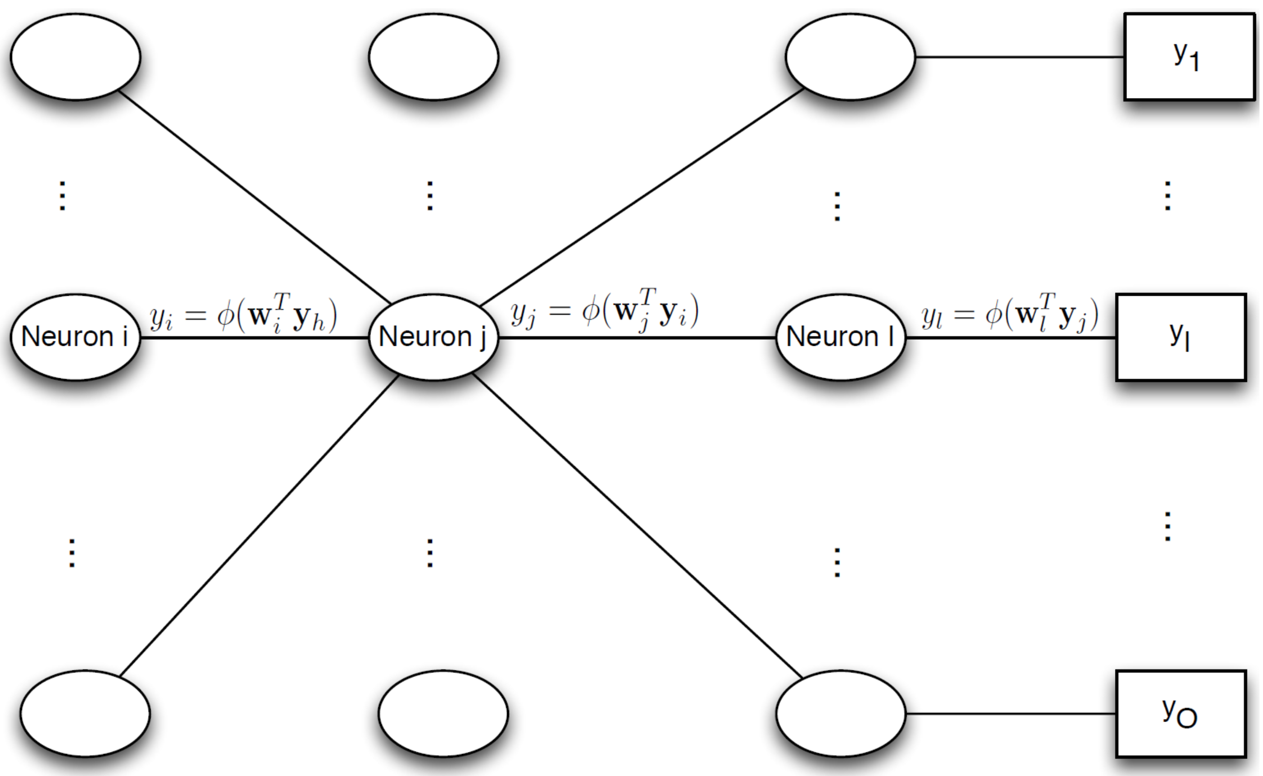
- Given a training set, $\{x_n, d_n\}_{n=1}^{N}$, we want to find the parameters of our network that minimizes the squared error:

$$J(w) = \frac{1}{2}\sum_{l=1}^{N}(d_l - y_l)^2$$

- In order to use gradient descent, we need to compute the analytic form of the gradient, $\frac{\partial J}{\partial w_{lj}}$.

In [4]:
```
Image('figures/HiddenLayer.png',width=700)
```

Out[4]:



Chain Rule Given a labelled training set, $\{x_n, d_n\}_{n=1}^{N}$, consider the objetive function

$$J(w) = \frac{1}{2}\sum_{l=1}^{N} e_l^2$$

where w are the parameters to be estimated and $\forall l$:

$$\begin{aligned}
&e_l = d_l - y_l \\
&y_l = \phi(v_l),\ \phi(\bullet) \text{ is an activation function} \\
&v_l = w^T x_j \text{ (note that } x_j \in \mathbb{R}^{D+1}\text{)}
\end{aligned}$$

Using the Chain Rule, we find:

$$\frac{\partial J}{\partial w_{lj}} = \frac{\partial J}{\partial e_l}\frac{\partial e_l}{\partial y_l}\frac{\partial y_l}{\partial v_l}\frac{\partial v_l}{\partial w_{lj}}$$

where

$$\frac{\partial J}{\partial e_l} = \frac{1}{2}2e_l = e_l = d_l - y_l$$

$$\frac{\partial e_l}{\partial y_l} = -1$$

$$\frac{\partial y_l}{\partial v_l} = \frac{\partial \phi(v_l)}{\partial v_l} = \phi'(v_l)$$

$$\frac{\partial v_l}{\partial w_{lj}} = x_j$$

Therefore

$$\frac{\partial J}{\partial w_{lj}} = e_l(-1)\phi'(v_l)x_j$$

- If activation function is the sigmoid, $\phi(x) = \frac{1}{1+e^{-x}}$, then $\phi'(x) = \phi(x)(1-\phi(x))$
- If activation function is the hyperbolic tangent (tanh), $\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, then $\phi'(x) = 1 - \phi(x)^2$
- If activation function is the ReLU, $\phi(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$, then $\phi'(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$

Now that we have the gradient, how do we use this to update the output layer weights in our MLP?

$$w_{lj}^{(t+1)} = w_{lj}^{(t)} - \eta\frac{\partial J}{\partial w_{lj}} = w_{lj}^{(t)} + \eta e_i \phi'(v_l)x_j$$

- How will this update equation (for the output layer) change if the network is a multilayer perceptron with hidden units?
- Can you write this in vector form to update all weights simultaneously?
- Next, the hidden layers...

Backpropagation of the Error for the Hidden Layers

- In a neural network, we can only define an error at the output layer! Therefore, we need to backward propagate the error obtain at the output layer, hence *backpropagation*.

Suppose we want to update w_{ji} where j is the hidden layer. (Let's follow the labeling in the figure below.)

The error objective function overall N data points is

$$J(w) = \frac{1}{2}\sum_{l=1}^{N} e_l^2 = \frac{1}{2}\sum_{l=1}^{N}(d_l - y_l)^2 = \frac{1}{2}\sum_{l=1}^{N}(d_l - \phi_l(v_l))^2$$

As we have seen earlier,

$$\begin{aligned}\frac{\partial J}{\partial w_{lj}} &= \frac{\partial J}{\partial e_l}\frac{\partial e_l}{\partial y_l}\frac{\partial y_l}{\partial v_l}\frac{\partial v_l}{\partial w_{lj}} \\ &= e_l(-1)\phi'(v_l)y_{jl}\end{aligned}$$

Let's define the *local gradient* δ_l:

$$\begin{aligned}\delta_l &= -\frac{\partial J}{\partial v_l} \\ &= e_l\phi'(v_l)\end{aligned}$$

Similarly,

$$\begin{aligned}\delta_j &= -\frac{\partial J}{\partial v_j} \\ &= -\frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial v_j} \\ &= -\frac{\partial J}{\partial y_j}\phi'(v_j)\end{aligned}$$

Note that,

$$\begin{aligned}\frac{\partial J}{\partial y_j} &= \sum_l \frac{\partial J}{\partial e_l}\frac{\partial e_l}{\partial y_l}\frac{\partial y_l}{\partial v_l}\frac{\partial v_l}{\partial y_j} \\ &= \sum_l e_l(-1)\phi'(v_l)w_{lj}\end{aligned}$$

So,

$$\begin{aligned}\delta_j &= -\frac{\partial J}{\partial y_j}\phi'(v_j) \\ &= -\left[\sum_l e_l(-1)\phi'(v_l)w_{lj}\right]\phi'(v_j) \\ &= \phi'(v_j)\sum_l \delta_l w_{lj}\end{aligned}$$

- We can write the gradient at a hidden neuron in terms of the local gradient and the connect neurons in the next layer:

$$\Delta w_{ij} = \eta \delta_j x_i$$

And so,

$$w_{ij}^{t+1} \leftarrow w_{ij}^{t} + \Delta w_{ij}^{t}$$

Best Practices for Training ANNs

1. Defining Network Architecture

Suppose you have a set of data $\{x_i\}_{i=1}^{N} \in \mathbb{R}^D$.

- What size network should you choose? How many layers? How many units per layer?

Input Layer

- Regardless of whether you are utilizing processing the **input space** or **feature space** of a given data set, the number of neurons in the input layer is the same as the dimensionality of the space.

Output Layer

- Suppose you are trying to do classification, then your output layer will represent the class labels.
- You can have different types of output encoding which directly impact performance.

Output Layer Encoding (also called Feature Engineering):

- Common encoding methods for classification:
    1. Integer encoding label for each class
    2. One-hot encoding (binary vectors with one indicator for each class)
    3. Binary (or other base) encoding

Hidden Layer

- We don't really know how many neurons to add in the hidden layer or how many hidden layers to use.
- **Rule of Thumb**: the amount of training data you need for a *well* performing model is 10x the number of parameters in the model.
    - Data will directly impact model choice...

Example: A network is said to have architecture 10-100-50-5 if its input layer has 10 units, 1st hidden layer 100 units, 2nd hidden layer 50 units and output layer 5 units.

to be continued...