Neural networks

Virgil Pavlu March 24, 2014

1 The perceptron

Lets suppose we are (as with regression regression) with $(\mathbf{x}_i, y_i); i = 1, .., m$ the data points and labels. This is a classification problem with two classes $y \in \{-1, 1\}$

Like with regression we are looking for a linear predictor (classifier)

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{x}\mathbf{w} = \sum_{d=0}^{D} x^d w^d$$

(we added the $x^0 = 1$ component so we can get the free term w^0) such that $h_{\mathbf{w}}(\mathbf{x}) \geq 0$ when $y = 1$ and $h_{\mathbf{w}}(\mathbf{x}) \leq 0$ when $y = -1$.

On $y = -1$ data points: given that all $\mathbf{x}$ and y are numerical, we will make the following transformation: when $y = -1$, we will reverse the sign of the input; that is replace $\mathbf{x}$ with **-x** and $y = -y$. Then the condition $h_{\mathbf{w}}(\mathbf{x}) \leq 0$ becomes $h_{\mathbf{w}}(\mathbf{x}) \geq 0$ for all data points.

The perceptron objective function is a combination of the number of miss-classification points and how bad the miss-classification is

$$J(\mathbf{w}) = \sum_{\mathbf{x} \in M} -h_{\mathbf{w}}(\mathbf{x}) = \sum_{\mathbf{x} \in M} -\mathbf{x}\mathbf{w}$$

where M is the set of miss-classified data points. Note that each term of the sum is positive, since miss-classified implies $\mathbf{w}\mathbf{x} < 0$. Using gradient descent, we first differentiate J

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \sum_{\mathbf{x} \in M} -\mathbf{x}^T$$

then we write down the gradient descent update rule

$$\mathbf{w} := \mathbf{w} + \lambda \sum_{\mathbf{x} \in M} \mathbf{x}^T$$

(λ is the learning rate). The batch version looks like

```
1. init $\mathbf{w}$
2. LOOP
3.         get $M$ = set of missclassified data points
4.         $\mathbf{w} = \mathbf{w} + \lambda \sum_{\mathbf{x} \in M} \mathbf{x}^T$
5. UNTIL $|\lambda \sum_{\mathbf{x} \in M} \mathbf{x}| < \epsilon$
```

Assume the instances are linearly separable. Then we can modify the algorithm

```
1. init $\mathbf{w}$
2. LOOP
3.         get $M$ = set of missclassified data points
4.         for each $\mathbf{x} \in M$ do $\mathbf{w} = \mathbf{w} + \lambda \mathbf{x}^T$
5. UNTIL $M$ is empty
```

Proof of perceptron convergence Assuming data is linearly separable , or there is a solution $\bar{\mathbf{w}}$ such that $\mathbf{x}\bar{\mathbf{w}} > 0$ for all $\mathbf{x}$.
Lets call $\mathbf{w}_k$ the $\mathbf{w}$ obtained at the k-th iteration (update). Fix an $\alpha > 0$. Then

$$\mathbf{w}_{k+1} - \alpha\bar{\mathbf{w}} = (\mathbf{w}_k - \alpha\bar{\mathbf{w}}) + \mathbf{x}_k^T$$

where $\mathbf{x}_k$ is the datapoint that updated $\mathbf{w}$ at iteration k. Then

$$||\mathbf{w}_{k+1} - \alpha\bar{\mathbf{w}}||^2 = ||\mathbf{w}_k - \alpha\bar{\mathbf{w}}||^2 + 2\mathbf{x}_k(\mathbf{w}_k - \alpha\bar{\mathbf{w}}) + ||\mathbf{x}_k||^2 \leq ||\mathbf{w}_k - \alpha\bar{\mathbf{w}}||^2 - 2\mathbf{x}_k\alpha\bar{\mathbf{w}} + ||\mathbf{x}_k||^2$$

Since $\mathbf{x}_k\bar{\mathbf{w}} > 0$ all we need is an α sufficiently large to show that this update process cannot go on forever. When it stops, all datapoints must be classified correctly.

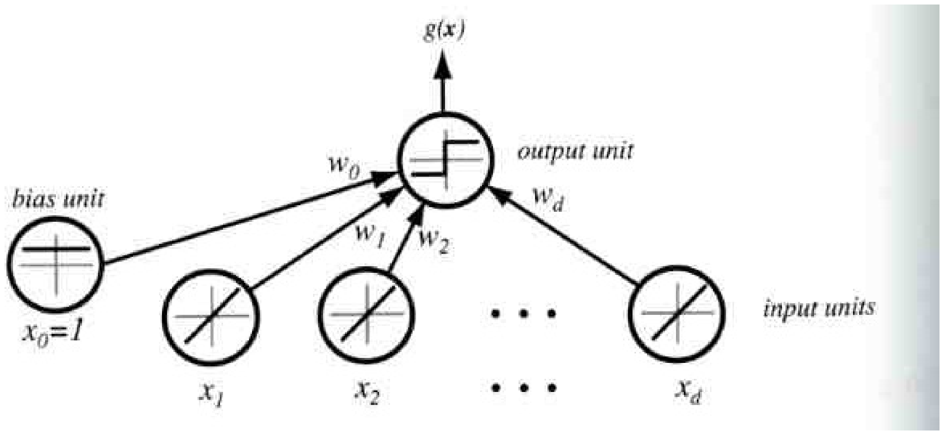

Figure 1: bias unit

2 Multilayer perceptrons

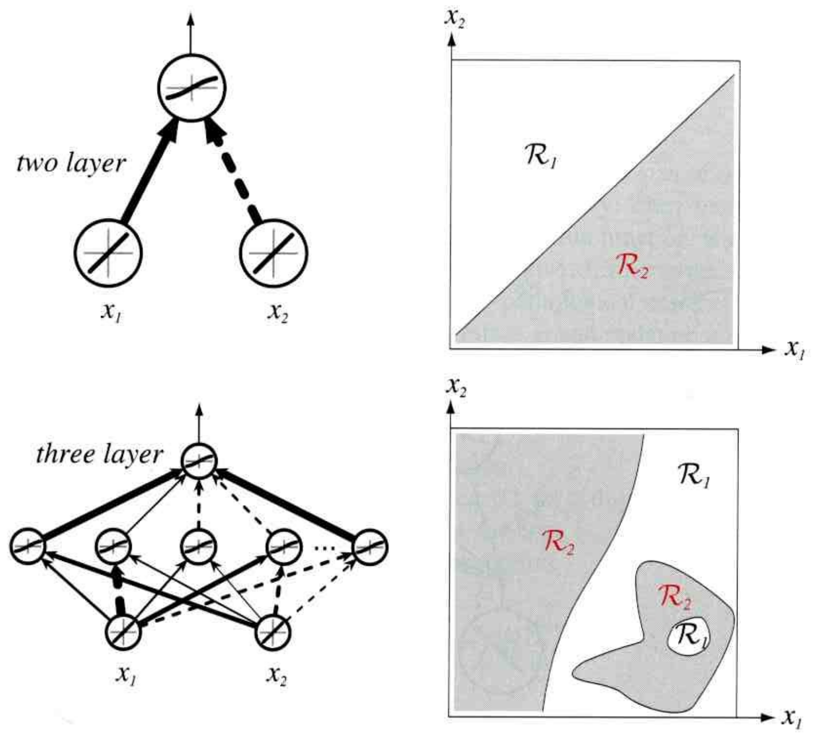

Figure 2: multilayer perceptron

2.1 More than linear functions, example: XOR

Perceptrons have been shown to have limited processing power. The decision boundary of a perceptron is a line (hyper plane), which means perceptrons can only classify objects that are linearly separable. Early researches discovered that perceptrons are not able to learn a XOR function. As can be seen in Figure 3, there is no single line that can separate the red points from the black points. Since XOR is an important function in circuit design, this shows the limited abilities of perceptron in practice.

However, perceptrons can implement **AND**, **OR** and **NOT** gates fairly easily, since the corresponding problems are linearly separable. We know **XOR** gates can be constructed using **AND**, **OR** and **NOT** gates:

$$XOR(x,y) = OR(x,y)\ AND\ (NOT(AND(x,y)))$$

This gives us the hint that by composing perceptrons together, we can get greater processing power. This leads to multi-layer neural networks (also called multi-layer perceptrons).

2.2 Construction and structure of NNets

Typical neural networks have the following structure. This is a 3-layer neural network. It consists of one input layer, one hidden layer and one output layer. Each node in the input layer represent a component of the feature vector. Each hidden unit performs the weighted sum of inputs to form a net activation:

$$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} = \mathbf{w}_j^t \mathbf{x}$$

where w_{ji} denotes the weight between input node i and hidden node j. Each hidden unit emits an output

$$y_j = f(net_j)$$

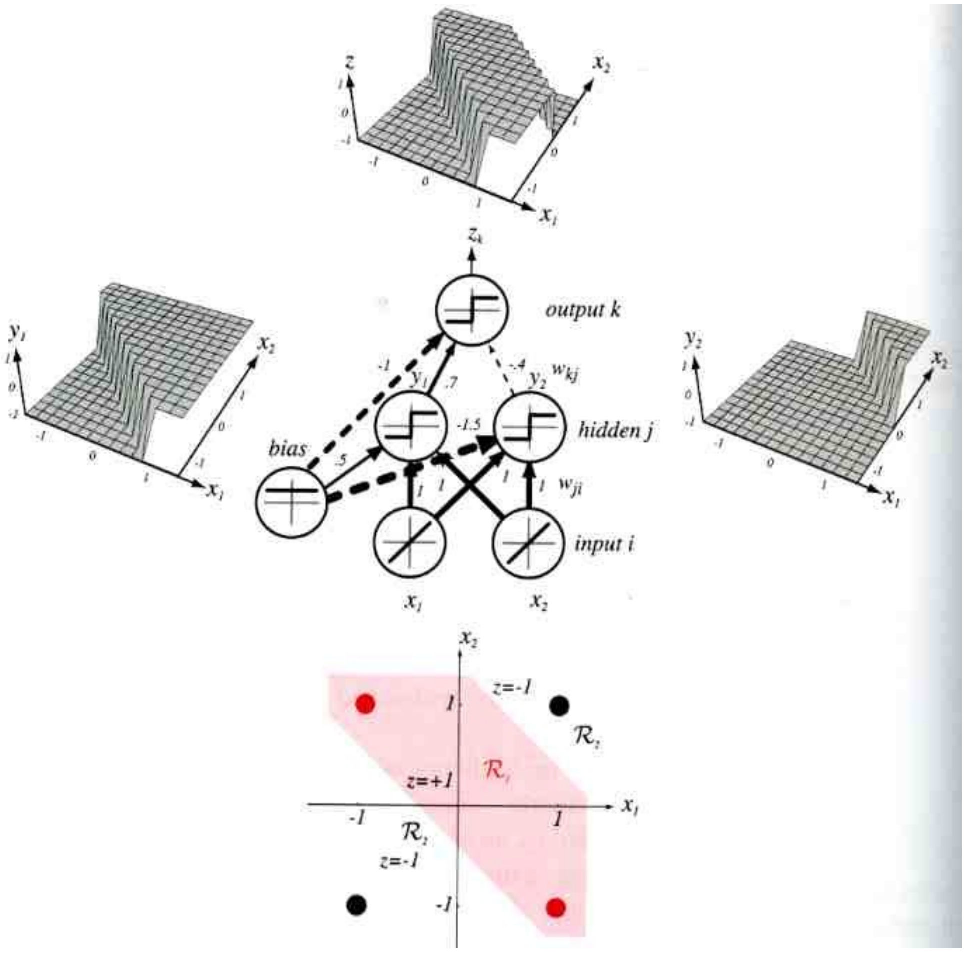

Figure 3: XOR NNet

Each output unit computes a net activation based on hidden unit outputs

$$net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = \mathbf{w}_k^t \mathbf{y},$$

and emits an output

$$z_k = f(net_k)$$

For a 3-layer neural network, the outputs can be written as

$$g_k(\mathbf{x}) = z_k = f\left(\sum_j w_{kj} f\left(\sum_i w_{ij} x_i + w_{j0}\right) + w_{k0}\right) = F\left(F(\mathbf{x}\mathbf{w}_j)\mathbf{w}_k\right)$$

Given these discriminant functions, we can predict each datapoint's label as

$$\arg\max_k g_k(x)$$

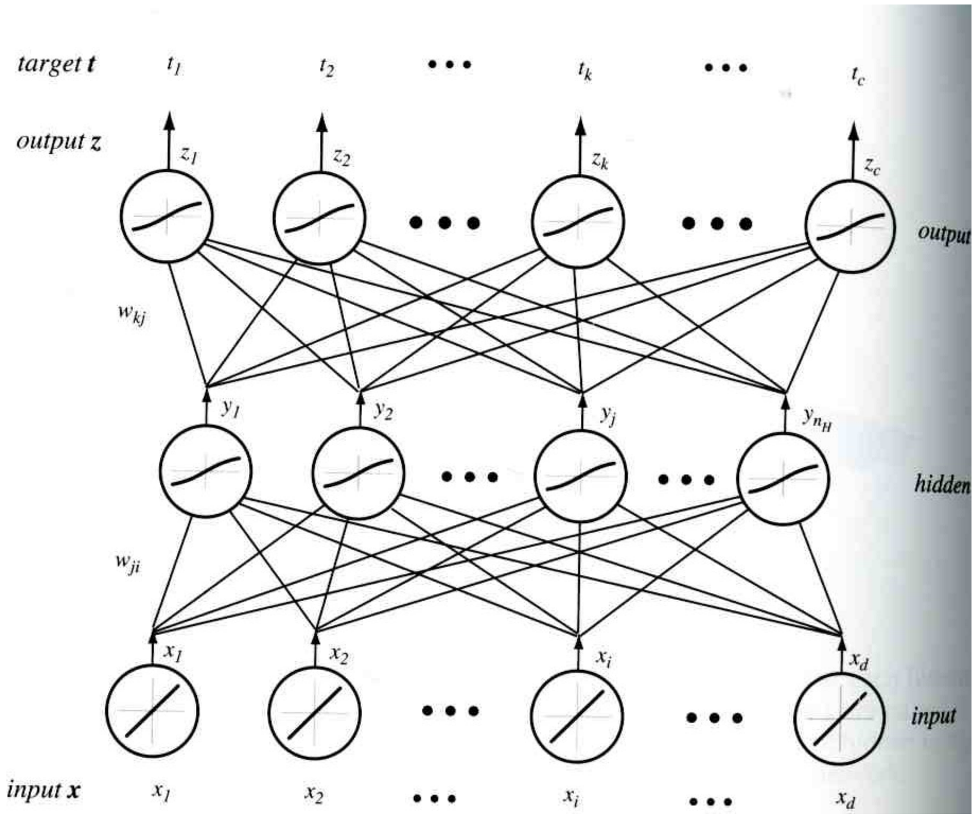

Figure 4: NNet fully connected

2.3 Kolmogorov theorem, expressive power of NNet

Any function g can be written

$$g(\mathbf{x}) = \sum_j \Xi_j \left(\sum_d \Psi_{dj}(x^d) \right)$$

but there is no practical way to use this theorem in practice. Usually Ξ and Ψ are very complex and not smooth.

It can be shown that neural networks are universal approximators. A feed-forward network with a single hidden layer containing a finite number of neurons, can approximate any continuous functions, under mild assumptions.

3 Training, Error backpropagation

In this section, we'll discuss how to learn a multi-layer neural network using a special kind of gradient descent algorithm – back propagation algorithm.

Similarly to linear regression, we try to minimize the squared error between the true labels and the predicted values. To make gradient descent easier, we often choose sigmoid(logistic) function as the activation function, because sigmoid function is differentiable.

Let's consider a 3-layer neural network:

- error

For one datapoint x:

$$J(w) = \frac{1}{2}\sum_k (t_k - z_k)^2$$

$$\Delta w_{pq} = -\lambda \frac{\partial J}{\partial w_{pq}},$$

where λ is the learning rate.

- propagation to last set of weights (close to output)

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = -\delta_k \frac{\partial net_k}{\partial w_{kj}}$$

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = (t_k - z_k)f'(net_k)$$

$$\frac{\partial net_k}{\partial w_{kj}} = y_j$$

So we get the update rule for hidden-to-output weights:

$$w_{kj} = w_{kj} + \lambda(t_k - z_k)f'(net_k)y_j$$

- propagation to first set of weights (close to input)

Using chain rule, we get

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}$$

$$\begin{aligned}\frac{\partial J}{\partial y_j} &= \frac{\partial[\frac{1}{2}\sum_k (t_k - z_k)^2]}{\partial y_j} \\ &= -\sum_k (t_k - z_k)\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial y_j}\end{aligned}$$

$$\frac{\partial h_j}{\partial net_j} = f'(net_j)$$

$$\frac{\partial net_j}{\partial w_{ji}} = x_i$$

$$w_{ji} \leftarrow w_{ji} - \lambda[\sum_k (t_k - z_k)f'(net_k)w_{kj}]f'(net_j)x_i$$

- stochastic VS batch

Based on the above derivation, we get the following two algorithms:

```
STOCHASTIC TRAINING
Select $x_t$ (randomly chosen)
    $w_{ij} = w_{ij} + \lambda\delta_j x_i$
    $w_{jk} = w_{jk} + \lambda\delta_k y_j$
until $| \bigtriangledown_w J| < \epsilon$
```

```
BATCH TRAINING
for each iteration:
    for each $x_t$
        $\delta w_{ij} = \delta w_{ij} + \lambda \delta_j x_i$
        $\delta w_{jk} = \delta w_{jk} + \lambda \delta_k y_j$
    $w_{ij} \leftarrow w_{ij} + \delta w_{ij}$
    $w_{jk} \leftarrow w_{jk} + \delta w_{jk}$
until $|| \bigtriangledown_w J|| < \epsilon$
```