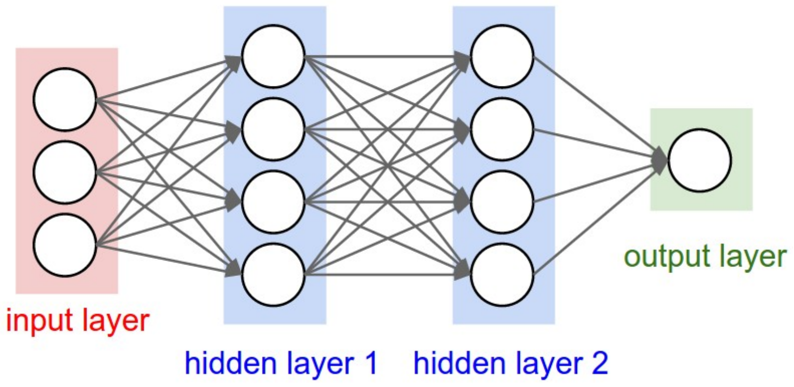
Neural Network

Why do we need Neural Network? Consider an image we like to recognize objects in it, if we try to use regression methods or the rest of methods we learned, we end up with huge matrix need to compute since we have a huge feature set.

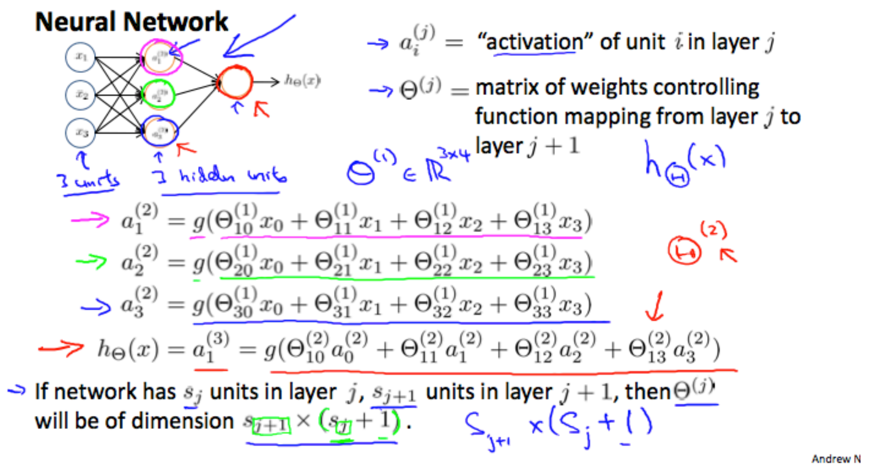
Basically, neural network is mimicking the brain. We teach a kid how to talk, how to recognize the object. With neural network, we like to be able to do the same with machine. There was an experiment performed on a mouse brain. Scientists cut the wiring between ear and part of brain called Auditory cortex (this part of brain hears), and rewired it to the eye. That part of brain learned to see!

Model Representation

Neural network takes input (dendrite) similar to brain and channel output (axons). The input features are like $x_1, x_2, \dots, x_n$. We add $x_0$as bias node and it's always equal to 1. In neural network, we use similar function to logistic regression, but we call it activation function, and we call theta, weights.



The intermediate layers between the input and output layer is called hidden layers. Here are the notation we use in our model; a_i^j: activation of unit I in layer j, and Θ^j: matrix of weight controlling the map function from layer j to layer j+1. Here is an example for neural network with 1-hidden layer.


x_1
x_2
x_3
$h_\Theta(x)$
3 units
3 hidden units

Neural Network

$\rightarrow a_i^{(j)} =$ "activation" of unit i in layer j

$\rightarrow \Theta^{(j)} =$ matrix of weights controlling function mapping from layer j to layer $j+1$

$\Theta^{(1)} \in \mathbb{R}^{3\times4}$ $\qquad h_\Theta(x)$

$$\rightarrow a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$
$$\rightarrow a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$
$$\rightarrow a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$
$$\rightarrow h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

$\Theta^{(2)}$

$\rightarrow$ If network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$. $\qquad s_{j+1} \times (s_j + 1)$

Andrew N

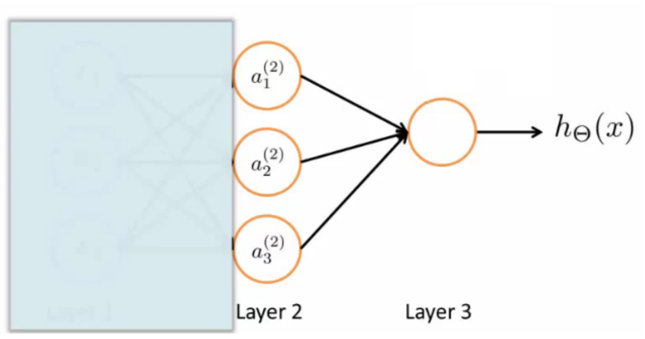
Neural Network

As an example, if layer 1 has 2 input nodes and layer 2 has 4 activation nodes. Dimension of theta1 is going to be 4×3.

Vectorize Implementation

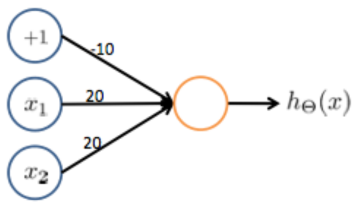
Let's define a new variable called z that encompress the parameter inside of the g function. $a_i^j = g(z_i^j)$. For example, $z_k^2 = \Theta_{k,0}^1 x_0 + \Theta_{k,1}^1 x_1 + \cdots + \Theta_{k,n}^1 x_n$, where $x = \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix}$, z=$\begin{bmatrix} z_1^j \\ z_2^j \\ \cdots \\ z_n^j \end{bmatrix}$, and x=$a^1$. We get $z^j = \Theta^{(j-1)} a^{(j-1)}$. Finally add bias term (matrix of 1) to layer j after computing a^j. To wrap it up, vectorize equation is:

$$\boxed{h_\Theta(x) = a^{(j+1)} = g(z^{(j+1)})}$$



Notice that in this last step, between layer j and layer j+1, we are doing exactly the same thing as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses. The only difference is, instead of input a feature vector, the features are just values calculated by the hidden layer. *In neural network learns from its own features*. We can define OR, NOR, and etc. with neural network easily.

Example: OR function



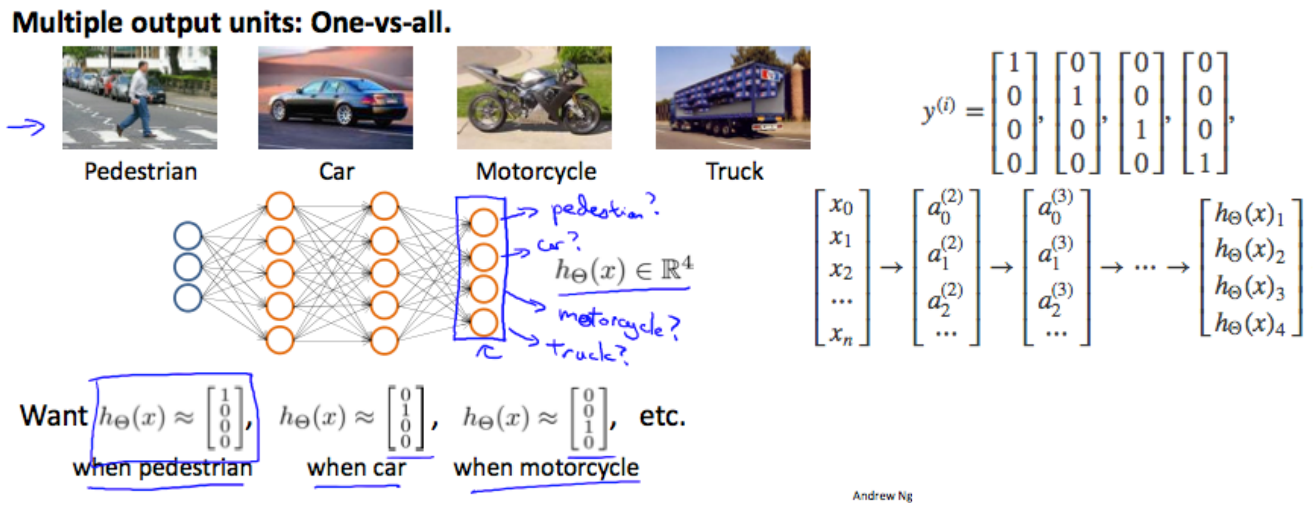
$g(-10 + 20x_1 + 20x_2)$

x_1	x_2	$h_\Theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	≈ 1
1	1	≈ 1

Neural Network

Multiclass Classification

When we have more than two classes to distinguish between, we use multiclass classification. Suppose in an image we are trying to distinguish between cars, pedestrians, motorcycle, and trucks. The output vector will be 1 by 4 array.

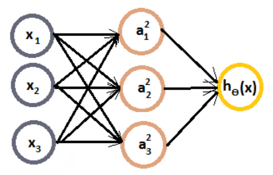

Multiple output units: One-vs-all.
Pedestrian
Car
Motorcycle
Truck
pedestrian?
car?
$h_\Theta(x) \in \mathbb{R}^4$
motorcycle?
truck?
$y^{(i)} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix},$
$\begin{bmatrix}x_0\\x_1\\x_2\\\cdots\\x_n\end{bmatrix} \to \begin{bmatrix}a_0^{(2)}\\a_1^{(2)}\\a_2^{(2)}\\\cdots\end{bmatrix} \to \begin{bmatrix}a_0^{(3)}\\a_1^{(3)}\\a_2^{(3)}\\\cdots\end{bmatrix} \to \cdots \to \begin{bmatrix}h_\Theta(x)_1\\h_\Theta(x)_2\\h_\Theta(x)_3\\h_\Theta(x)_4\end{bmatrix}$
Want $h_\Theta(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, etc.
when pedestrian
when car
when motorcycle
Andrew Ng

So instead of integer like logistic regression, we have arrays as an output for classifying the outputs.

Forward Propagation

We talked about the model representation of neural network. NN is one of the most powerful learning algorithm in machine learning. The process of forward propagation is:

- Start off with activation of input unit
- Forward propagate and calculate the activation layer by layer



$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

Cost Function

Cost function for NN is similar to logistic regression, but abit more complex:

$$J(\Theta) == \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[\, {y^i}_k log\, h_\Theta(x^i) + (1 - y^i)\log\left(1 - h_\Theta(x^i)\right)] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_j}\sum_{j=1}^{s_{j+1}}(\Theta_{j,i}^{(l)})^2$$

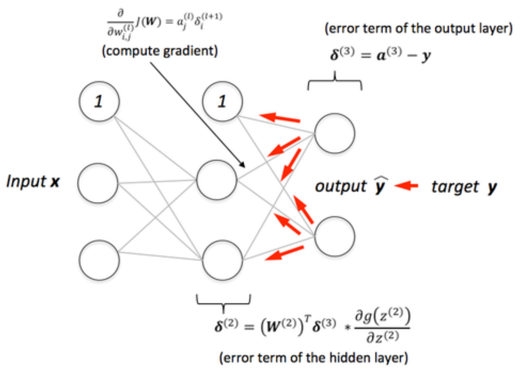
Where L is total number of layers, s_j is number of units (not containing bias unit) in layer l, and K is the number of output classes. The nested summation is for considering multiple output nodes. Similar to logit, we square up the last term. To wrap up:

- the double sum simply adds up the logistic regression costs calculated for each cell in the output layer
- the triple sum simply adds up the squares of all the individual Θs in the entire network.
- the i in the triple sum does not refer to training example i

Backpropagation Learning, Error Calculation, and Weight Adjustment

We've already described forward propagation. This is the algorithm which takes your neural network and the initial input into that network and pushes the input through the network. It leads to the generation of an output hypothesis, which may be a single real number, but can also be a vector. Remember there is a theta matrix for every single layer.

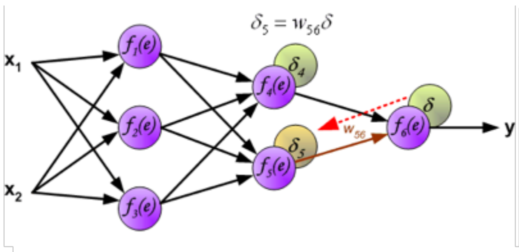
In backpropagation, we take the output we got from our network, compares it to the real value (y) and calculates how wrong the network was. Then we back-calculates the error associated with each unit from the preceding layer (i.e. layer L - 1). This goes on until you reach the input layer (*where obviously there is no error, as the activation is the input*). These "error" measurements for each unit can be used to calculate the partial derivatives, we may use gradient descent to minimize the cost function and update vale of Θ. We repeat it until the descent converges (similar to the previous time).


$\frac{\partial}{\partial w_{i,j}^{(l)}} J(W) = a_j^{(l)} \delta_i^{(l+1)}$
(compute gradient)
(error term of the output layer)
$\delta^{(3)} = a^{(3)} - y$
1
1
Input x
output $\hat{y}$
target y
$\delta^{(2)} = (W^{(2)})^T \delta^{(3)} * \frac{\partial g(z^{(2)})}{\partial z^{(2)}}$
(error term of the hidden layer)

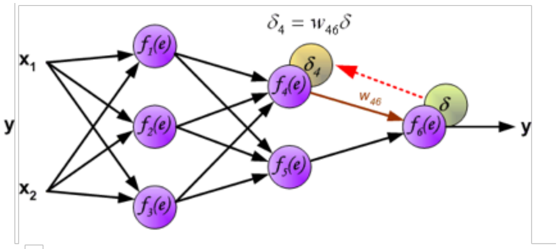
To teach the neural network, we need training set. The training set has assigned corresponded target. The network training is an iterative process. *In each process weights coefficients of nodes are modified using new dataset from the training dataset*. Pictures below demonstrate how propagation through networks. Initially random numbers are assigned to the weights. Starting from forward propagation, values in each hidden layer is calculated till get the output y. Next step is comparing the value to target value and calculating the error d of output layer neuron. It is impossible to compute error signal for internal neurons directly, because output values of these neurons are unknown. For many years the effective method for training multiplayer networks has been unknown. Only in the middle eighties the backpropagation algorithm has been worked out. The idea is to propagate error signal d (computed in single teaching step) back to all neurons.

After the error signal is calculated, the weights may be modified.

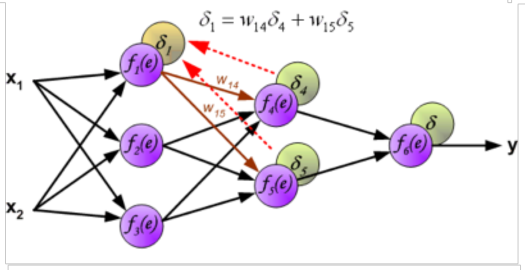
Neural Network



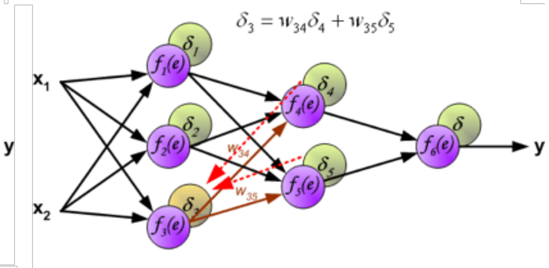
$$\delta_5 = w_{56}\delta$$



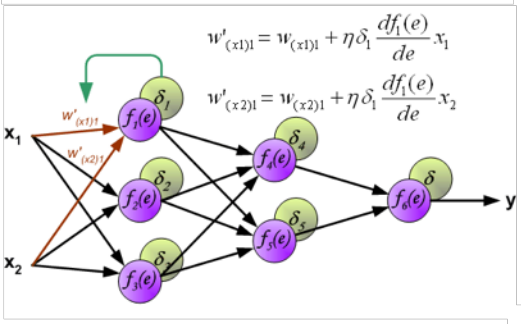
$$\delta_4 = w_{46}\delta$$



$$\delta_1 = w_{14}\delta_4 + w_{15}\delta_5$$

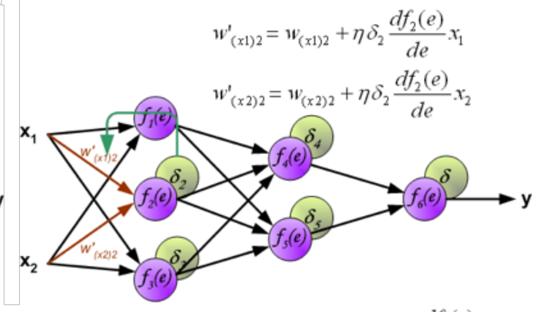


$$\delta_3 = w_{34}\delta_4 + w_{35}\delta_5$$



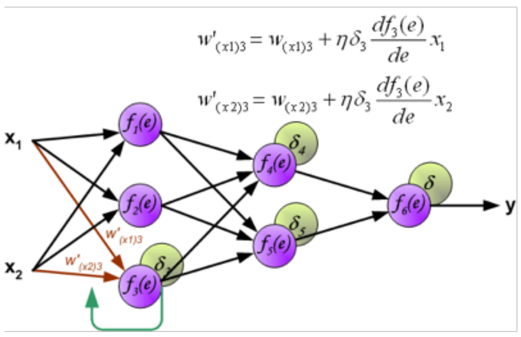
$$w'_{(x1)1} = w_{(x1)1} + \eta\delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta\delta_1 \frac{df_1(e)}{de} x_2$$



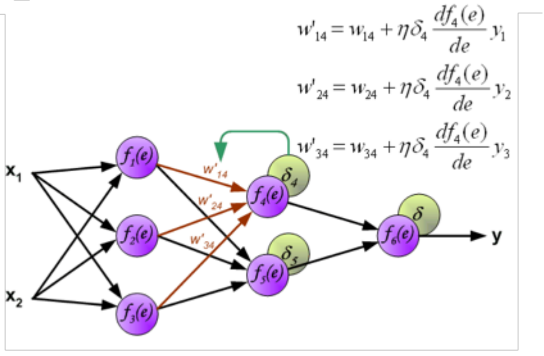
$$w'_{(x1)2} = w_{(x1)2} + \eta\delta_2 \frac{df_2(e)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} + \eta\delta_2 \frac{df_2(e)}{de} x_2$$



$$w'_{(x1)3} = w_{(x1)3} + \eta\delta_3 \frac{df_3(e)}{de} x_1$$

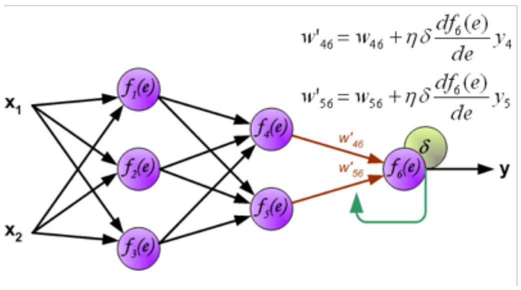
$$w'_{(x2)3} = w_{(x2)3} + \eta\delta_3 \frac{df_3(e)}{de} x_2$$



$$w'_{14} = w_{14} + \eta\delta_4 \frac{df_4(e)}{de} y_1$$

$$w'_{24} = w_{24} + \eta\delta_4 \frac{df_4(e)}{de} y_2$$

$$w'_{34} = w_{34} + \eta\delta_4 \frac{df_4(e)}{de} y_3$$



$$w'_{46} = w_{46} + \eta\delta \frac{df_6(e)}{de} y_4$$

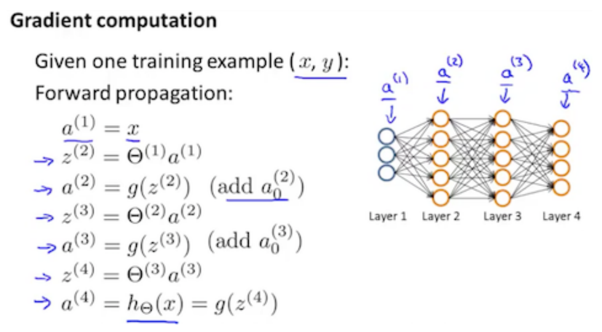
$$w'_{56} = w_{56} + \eta\delta \frac{df_6(e)}{de} y_5$$

Pictures taken from home.agh.edu.pl

Neural Network

In summary, here is how we update the parameter in backpropagation learning. Initialize all weight with random numbers (typically values between -1 and 1). If we give value of 0 to the weight, all nodes will end up having a same value.

- Given a training set (x^1, y^1), (x^2, y^2), …, (x^m, y^m)
- Set $\Delta^l_{i,j}$=0 for all (i, j, l)
- Set $a^1 = x^1$
- Perform forward propagation to compute a^l for l=2, 3, …, L



Gradient computation

Given one training example $(\underline{x, y})$:

Forward propagation:

$$\underline{a^{(1)}} = \underline{x}$$
$$\rightarrow z^{(2)} = \Theta^{(1)}a^{(1)}$$
$$\rightarrow a^{(2)} = g(z^{(2)}) \quad (\text{add } \underline{a_0^{(2)}})$$
$$\rightarrow z^{(3)} = \Theta^{(2)}a^{(2)}$$
$$\rightarrow a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$\rightarrow z^{(4)} = \Theta^{(3)}a^{(3)}$$
$$\rightarrow a^{(4)} = \underline{h_\Theta(x)} = g(z^{(4)})$$

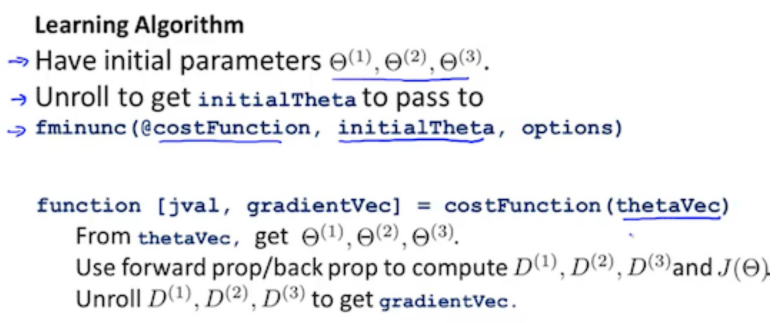
- Using y^t, compute $\delta^L = a^L - y^t$ (where L is total number of layers and a^L is the vector of outputs pf the activation units for the last layer. So, our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y.
- Compute δ^{L-1}, δ^{L-2}, …, δ^2, using $\delta^L = [(\Theta^l)^T\delta^{l+1}].*\, a^l.*\,(1 - a^l)$. The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values. The g prime derivative terms is: $g'(z^l) = a^l.*\,(1 - a^l)$
- $\Delta^l_{i,j} := \Delta^l_{i,j} + a^l_j\delta^{l+1}_i$ or vectorization form $\Delta^l := \Delta^l + \delta^{l+1}(a^l)^T$, hence matrix delta gets updated:
    - $D^l_{i,j} = \frac{1}{m}(\Delta^l_{i,j} + \lambda\Theta^l_{i,j})$ if j≠ 0
    - $D^l_{i,j} = \frac{1}{m}(\Delta^l_{i,j})$ if j=0

  The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. $D^l_{i,j} = \frac{\partial}{\partial\Theta^l_{i,j}} J(\Theta)$

Neural Network

Unrolling parameter

In order to use optimizing functions we will want to "unroll" all the elements and put them into one long vector.



Learning Algorithm

→ Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

→ Unroll to get `initialTheta` to pass to

→ `fminunc(@costFunction, initialTheta, options)`

```
function [jval, gradientVec] = costFunction(thetaVec)
```

From `thetaVec`, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

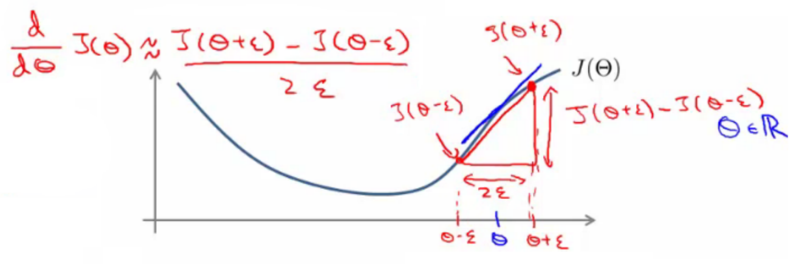
Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$.

Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get `gradientVec`.

Gradient Checking

One way of debugging a neural network is to do gradient checking, it helps make sure the algorithm works properly.

In this method, we calculate the derivative of J. when epsilon is so small, it gives derivative in a plot on the right.


$\frac{d}{d\Theta} J(\Theta) \approx \frac{J(\Theta+\varepsilon) - J(\Theta-\varepsilon)}{2\varepsilon}$
$J(\Theta+\varepsilon)$
$J(\Theta)$
$J(\Theta-\varepsilon)$
$J(\Theta+\varepsilon) - J(\Theta-\varepsilon)$
$\Theta \in \mathbb{R}$
2ε
$\Theta-\varepsilon$
Θ
$\Theta+\varepsilon$

Use gradient checking to compare the partial derivatives computed using the above algorithm and numerical estimation of gradient of J. We disable the gradient checking after we start running the code in later.

The cost function is non-convex, so it can end up to the local minimum, but even local minimum gives us pretty good values for our model so it's not a big concern.

Application

One of the application of neural network is self-driving car. At first, it learns how to drive with training set, then it will drive by itself and adapt to the new environments. If there is a pattern or situation, it has not seen before, the car would not behave correctly.