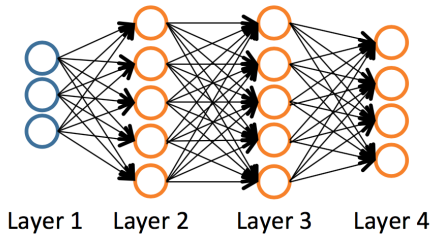
5. Neural Networks: Learning

The Neural Network is one of the most powerful learning algorithms (when a linear classifier doesn't work, this is what I usually turn to), this lecture investigates the *'backpropagation'* algorithm for training these models. First, we begin with a learning algorithm for fitting the parameters of a neural network given a training set. As with the discussion of most of our learning algorithms, we're going to begin by talking about the *cost function* for fitting the parameters of the network.

test

Example 5.1 : *Neural Network (classification)* Focusing on a Neural Network for *classification problems*, suppose we have a network like that shown on the left.



$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$$

$L =$ Total no. of layers in network

$s_\ell =$ No. of units (not counting bias unit) in layerℓ.

There are two major types of classification problems:

Binary Classification:
Here, the labels are binary and there is one output unit:

$$y = 0 \text{ or } 1$$
$$h_\Theta(x) \in \mathbb{R}$$

Then,

$$s_\ell = 1 \text{ and } k = 1$$

Multi-class Classification:
Now we have K distinct classes. So our early example had this representation for y if we have 4 classes. We also have K output units.

$$y \in \mathbb{R}^K$$

with:

$$s_\ell = K \text{ and } k =\geq 3$$

for example:

$$y^{(i)} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

5.1. Cost Function. Now, looking at the cost function, we notice that it is simply a generalization of the *Logistic regression* cost function:

Definition 1. ***Logistic Regression***

$$J(\theta) = -\frac{1}{m}\sum_{j=1}^{n} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

Which includes the cost function as well as the regularization term. For our neural network, we now have K output terms:

$$h_\Theta(x) \in \mathbb{R}^K \qquad (h_\Theta(x))_i = i^{th} \text{ output}$$

Definition 2. ***Neural Network***

$$J(\Theta) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log h_\Theta(x^{(i)})_k + (1-y_k^{(i)}) \log(1-h_\Theta(x_k^{(i)})) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

Now, we have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple θ matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

Note:

- 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

5.2. Backpropagation Algorithm.

Now, lets find a way to minimize the cost function. In particular, we'll talk about the backpropagation algorithm. "Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute: $\min_\Theta J(\Theta_0)$

That is, we want to minimize our cost function J using an optimal set of parameters in Θ. In this section we'll look at the equations we use to compute the partial derivative of $J(\Theta)$: $\frac{\partial}{\partial \Theta_{ij}^{(\ell)}}$

Example 5.2 : ***Gradient Computation:*** *One training example*

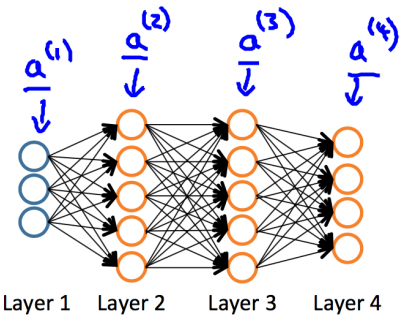
First, we perform a forward propagation, which is then followed by backpropagation. Now, given one training example (x, y):

$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$

$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$

$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$



'activation values'

Now, we have a vectorized implementation of the forward propagation. Next, in order to compute the derivatives, we will use *'backpropagation.*

Backpropagation:

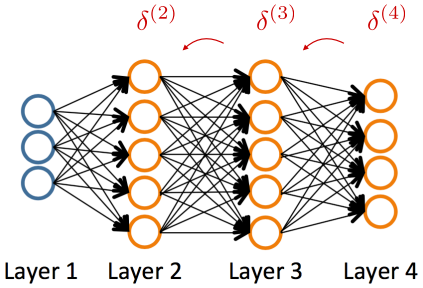
Intuition: $\delta_j^{(\ell)}$ = 'error' of node j in layer ℓ. So this delta term is going to capture (in some sense) the error in each node.

For each output unit (Layer $L = 4$), we calculate $\delta_j^{(\ell)}$:

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

or, written in vector-form:

$$\delta^{(4)} = a^{(4)} - y$$
$$\delta^{(3)} = (\Theta^{(3)})^\intercal \delta^{(4)} .* g'(z^{(3)})$$
$$\delta^{(2)} = (\Theta^{(2)})^\intercal \delta^{(3)} .* g'(z^{(2)})$$
$$- \ (\text{ No } \delta^{(1)}) -$$



Note: It is possible to prove (ignoring λ-regularization), that:

$$\frac{\partial}{\partial \Theta_{ij}^{(\ell)}} J(\Theta) = a_j^{(\ell)} \delta_i^{(\ell+1)}$$

Backpropagation Algorithm:

Now, putting everything together, we implement *backpropagation* to compute derivatives with respect to your parameters for a general (potentially large) training set.

Suppose we have a training set of m examples:

Algorithm 1: Back Propagation

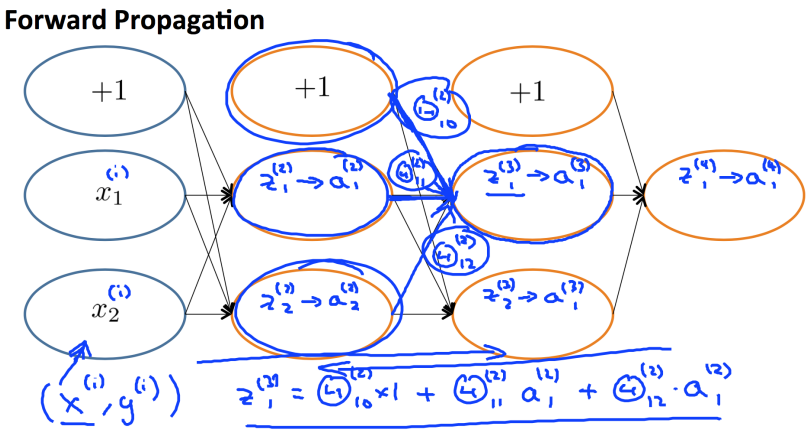
training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$
Set $\Delta_{ij}^{(\ell)} = 0$ (for all ℓ, i, j)
for $i = 1$ **to** m **do**
  Set $a^{(1)} = x^{(i)}$
  Perform forward propagation to compute $a^{(\ell)}$ for $\ell = 2, 3, \ldots, L$
  Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
  Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
  $\Delta_{ij}^{(\ell)} := \Delta_{ij}^{(\ell)} + a_j^{(\ell)} \delta_i^{(\ell+1)}$
end
$D_{ij}^{(\ell)} := \frac{1}{m} D_{ij}^{(\ell)} + \lambda \Theta_{ij}^{(\ell)}$ if $j \neq 0$
$D_{ij}^{(\ell)} := \frac{1}{m} D_{ij}^{(\ell)}$ if $j = 0$

Finally, once we have the D-terms, one can show that this becomes:

$$\frac{\partial}{\partial \Theta_{ij}^{(\ell)}} J(\Theta) = D_{ij}^{(\ell)}$$

Which is what we're looking for.

5.3. Backpropagation: Intuition.



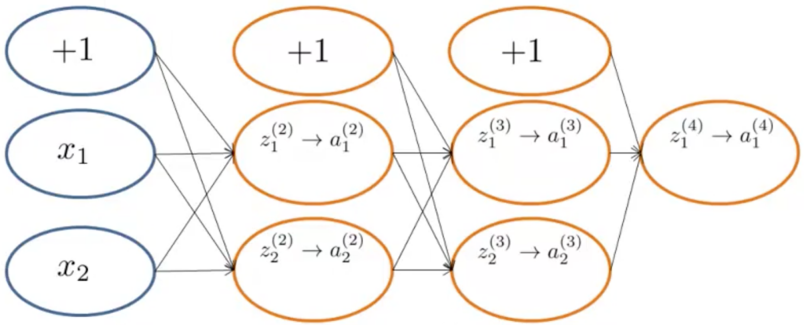
Now, focusing on one single example, $x^{(i)}, y^{(i)}$, the case of 1 output unit, and ignoring regularization $\lambda = 0$, our cost function becomes:

$$cost(i) = y_k^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\Theta(x^{(i)}))$$

or, more simple yet, think of:

$$cost(i) = \left(h_\Theta(x^{(i)}) - y^{(i)}\right)^2$$

This defines the error at each point, as defined by the sum of squares approach.

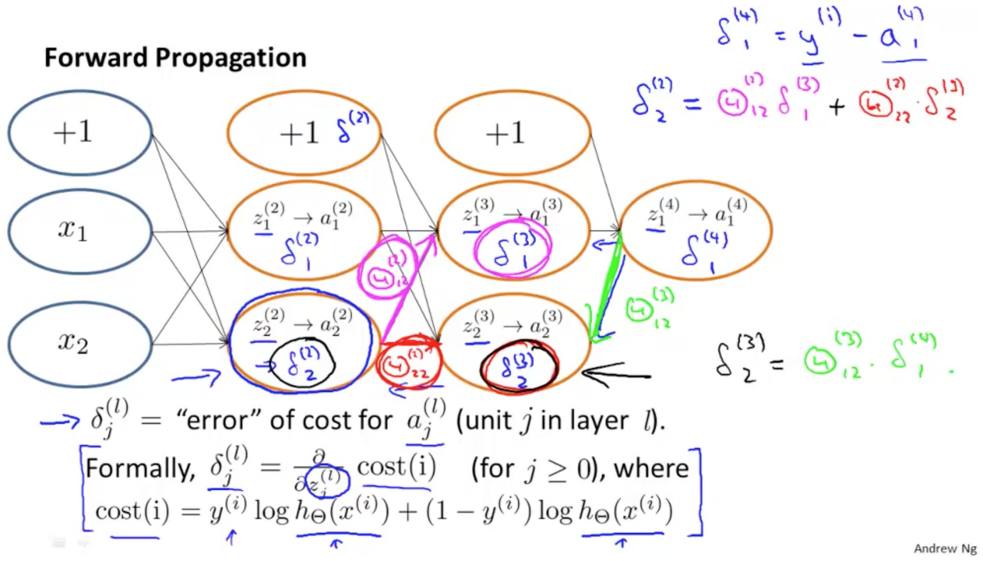


$$\delta_j^{(\ell)} = \text{ 'Error' of cost for } a_j^{(\ell)} \text{ (unit } j \text{ in layer } \ell)$$

formally, $\delta_j^{(\ell)} = \frac{\partial}{\partial z_j^{(\ell)}} cost(i)$, for $(j \geq 0)$, where

$$cost(i) = y_k^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\Theta(x^{(i)}))$$

How to calculate the delta values:



Here, we can say that $z_j^{(\ell)}$ = 'weighted sum of inputs;' then δ is the partial derivative of the cost function w.r.t the weighted sum of inputs. It is also a measure of 'How much would we like to change the network weights' in order to affect the intermediate values, and subsequently, the overall cost function.

5.4. Implementation.

Now, in order to use some of the advanced optimization routines, we need to reformat our data structures to something that they're expecting. That is, unrolling your parameters from matrices into vectors.

Concretely, let's say you've implemented a cost function:

```
% define cost function
function [jVal, gradient] = costFunction(theta)
...
% run optimization routine
optTheta = fminunc(@costFunction, initialTheta, options)
```

In the functions above, the expected data types are as follows:

- `theta` = $\mathbb{R}^{n+1}$ (vectors)
- `gradient` = $\mathbb{R}^{n+1}$ (vectors)

However, our neural network is composed of multiple matrices:

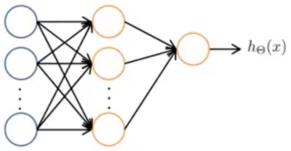
- $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$ - (matrices), `(Theta1, Theta2, Theta3)`
- $D^{(1)}$,$D^{(2)}$,$D^{(3)}$ - (matrices), `(D1, D2, D3)`

In order to use these in the advanced optimization schemes above, they need to be 'unrolled' into vectors.

Example 5.3 : *Unrolling vectors* Assume we have the following scenario:

$$s_1 = 10, s_2 = 10, s_3 = 1$$

$$\Theta^{(1)} \in \mathbb{R}^{10\times 11}, \Theta^{(2)} \in \mathbb{R}^{10\times 11}, \Theta^{(3)} \in \mathbb{R}^{1\times 11}$$
$$D^{(1)} \in \mathbb{R}^{10\times 11}, D^{(2)} \in \mathbb{R}^{10\times 11}, D^{(3)} \in \mathbb{R}^{1\times 11}$$



Here we have three layers: *one input layer (10 units)*, *one hidden layer (10 units)*, and *one output layer (one unit)*. The dimension of our matrices are given above.

Converting from a matrix to a vector may be done as follows:

```
% convert from matrix to vector
thetaVec = [ Theta1(:); Theta2(:); Theta3(:) ];
DVec = [ D1(:); D2(:); D3(:) ];
```

Reverting back may be done as follows:

```
Theta1 = reshape ( thetaVec(1:110), 10, 11);
Theta2 = reshape ( thetaVec(111:220), 10, 11);
Theta3 = reshape ( thetaVec(221:231), 1, 11);
```

Learning Algorithm:

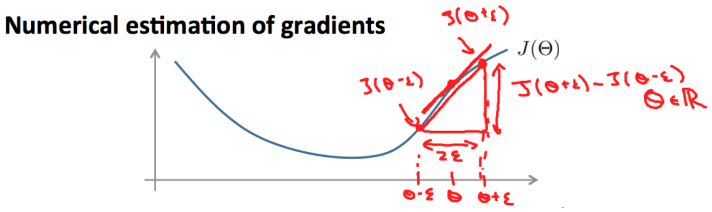
- Have initial parameters: $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$.
- Unroll to get `initialTheta` to pass to `fminunc(@costFunction, initialTheta, options)`

Then:

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

5.5. Gradient Checking.

Gradient checking will assure that our backpropagation works as intended. We can approximate the derivative of our cost function as follows:



In which the central difference is written as:

$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

With multiple theta matrices, we can approximate the derivative with respect to Θ_j as follows:

$$\frac{\partial}{\partial \Theta_j} J(\Theta) \approx \frac{J(\Theta_1, \ldots, \Theta_j + \epsilon, \ldots, \Theta_n) - J(\Theta_1, \ldots, \Theta_j - \epsilon, \ldots, \Theta_n)}{2\epsilon}$$

A small value for ϵ such as $\epsilon = 10 - 4$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems.

Hence, we are only adding or subtracting epsilon to the Θ_j matrix. In octave we can do it as follows:

```
epsilon = 1e-4;
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) += epsilon;
```

```
  thetaMinus = theta;
  thetaMinus(i) -= epsilon;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;
```

We previously saw how to calculate the `deltaVector`. So once we compute our `gradApprox` vector, we can check that `gradApprox` $\approx$ `deltaVector`.

Once you have verified once that your backpropagation algorithm is correct, you don't need to compute `gradApprox` again. The code to compute `gradApprox` can be very slow.

5.6. Random Initialization.

Initializing all Θ weights to zero does not work with neural networks - this is true for the gradient descent and other advanced optimization algorithms. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our Θ matrices, that is:

- Initialized each $\Theta_{ij}^{(\ell)}$ to a random value in $[-\epsilon, \epsilon]$, that is: $(-\epsilon \leq \Theta_{ij}^{(\ell)} \leq \epsilon)$

Written in Matlab code:

```
    Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

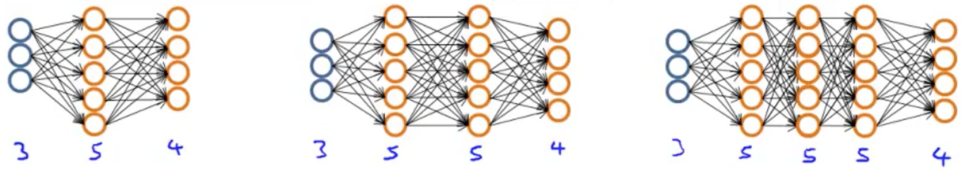
Hence, we initialize each $\Theta_{ij}^{(\ell)}$ to a random value between $[-\epsilon, \epsilon]$. Using the above formula guarantees that we get the desired bound. The same procedure applies to all the Θ's. Below is some working code you could use to experiment.

```
% If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11.
Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta3 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

`rand(x,y)` is just a function in octave that will initialize a matrix of random real numbers between 0 and 1.

5.7. Summary: Putting it all together.

First, ***choose network architecture***:



Choose the layout of your neural network, including:

- Number of input units = dimension of features x(i)
- Number of output units = number of classes
- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)

Note: *Defaults:* 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

Secondly, ***train the network:***

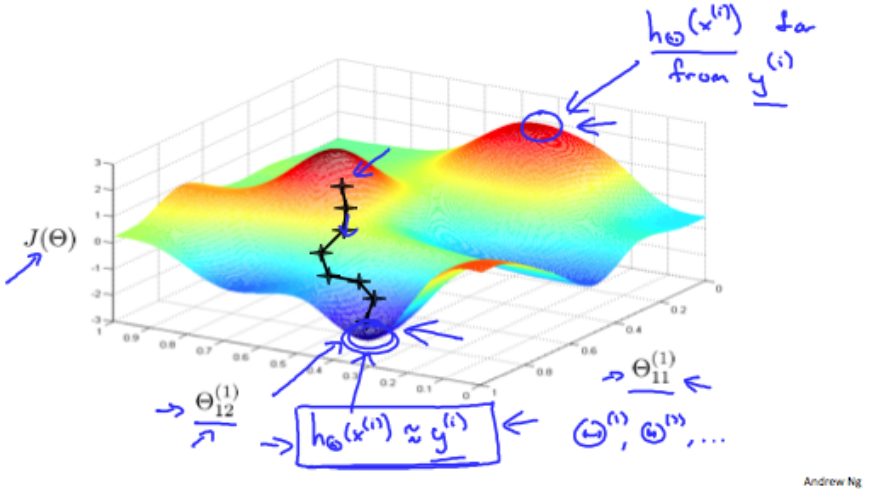
- Randomly initialize the weights
- Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x(i)$

- Implement the cost function
- Implement backpropagation to compute partial derivatives
- Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
- Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.
-

When we perform forward and back propagation, we loop on every training example:

```
for i = 1:m,
   Perform forward propagation and backpropagation using example (x(i),y(i))
   (Get activations a(l) and delta terms d(l) for l = 2,...,L
```

The following image gives us an intuition of what is happening as we are implementing our neural network:



Ideally, you want $h_\Theta(x^{(i)}) \approx y(i)$. This will minimize our cost function. However, keep in mind that $J(\Theta)$ is not convex and thus we can end up in a local minimum instead.