Stanford Machine Learning - Week V

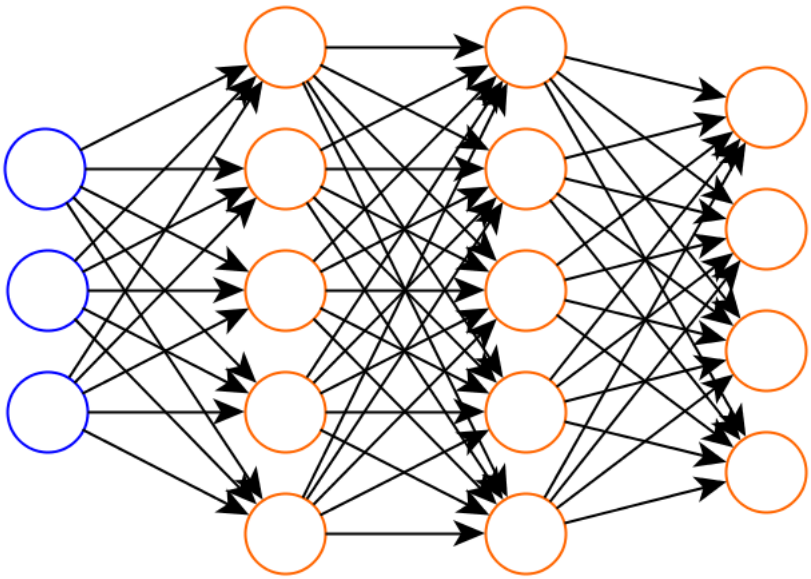
Eric N Johnson

August 13, 2016

1 Neural Networks: Learning

What learning algorithm is used by a neural network to produce parameters for a model?

Suppose we have a neural network that we will use for classification.



where

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

are training examples, L denotes the total number of layers in the network, and s_l denotes the number of units (not including bias units) in layer l. Here $L = 4$, $s_1 = 3$, $s_2 = s_3 = 5$, and $s_L = s_4 = 3$.

For binary classification, $y \in \{0, 1\}$, and there is 1 output unit $h_\Theta(x) \in \mathbb{R}$. $s_L = 1$, and $K = 1$.

For multi-class classification with K classes, the response variable y is a vector with a single 1 and otherwise zeros. E.g.,

$$y \in \mathbb{R}^k$$
$$= \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \text{ or } \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \text{ etc.}$$

and so $h_\Theta(x) \in \mathbb{R}^K$, $s_L = K$.

1.1 Cost Functions

We will generalize our cost function used for regularized logistic regression.

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

For multi-class classification, $h_\Theta(x) \in \mathbb{R}^K$. Let $(h_\Theta(x))_i$ be the i^{th} element of the vector $h_\Theta(x)$. The new cost function is then

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

Note that $k = 1, 2, ..., K$ denotes the each of the output units. The second term is the regularization term. The regularization term just sums over each of the Θ_{ji} terms.

2 Backpropagation Algorithm for Optimizing the Cost Function

To minimize $J(\Theta)$ as a function of Θ using one of the optimization methods (*fminunc, conjugate gradient, BFGS, L-BFGS*), we need to calculate both $J(\Theta)$ and each of the partial derivative terms $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ for all i, j, l.

Suppose we only have one training example (x, y). First we apply forward propagation:

$$\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \text{ and add } a_0^{(2)} \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g(z^{(3)}) \text{ and add } a_0^{(3)} \\
z^{(4)} &= \Theta^{(3)} a^{(3)} \\
a^{(4)} &= h_\Theta(x) = g(z^{(4)})
\end{aligned}$$

to obtain activations for each layer with bias terms.

Then to obtain the gradient calculation, we use back propogation:

$$\delta_j^{(l)} = \text{ error of node } j \text{ in layer } l$$

Recall that $a_j^{(l)}$ is the j^{th} activation for layer l. For each output layer (here $L =$), we compute:

$$\begin{aligned}
\delta_j^{(4)} &= a_j^{(4)} - y_j \\
&= (h_\Theta(x))_j - y_j
\end{aligned}$$

We may vectorize this code in Octave by simply performing vector subtraction

```
delta[4] = a[4] - j
```

Then we compute the calculate the error for the other terms (pseudocode):

$$\begin{aligned}
\delta^{(3)} &= (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)}) \\
\delta^{(2)} &= (\Theta^{(2)})^T \delta^{(4)} .* g'(z^{(2)})
\end{aligned}$$

where $g'(z^{(3)})$ is the derivative of the sigmoid function evaluated at $z^{(3)}$. This can be done in Octave using

```
a[3].*(1-a[3])
```

and

```
a[2].*(1-a[2])
```

There is no $\delta^{(1)}$ term since we don't have any error there.

Finally, if we ignore regularization, we have

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

2.1 The Backpropagation Algorithm

Training set: $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$

Set: $\Delta_{ij}^{(l)} = 0$ for all i, j, l (used to compute the gradients)

FOR $i = 1$ to m

- Set: $a^{(1)} = x^{(i)}$
- Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, ..., L$
- Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
- Compute $\delta^{(L-1)}, \delta^{(L-2)}, ..., \delta^{(2)}$ (no $\delta^{(1)}$)
- Let $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
- I.e., we calculate... $\Delta^{(L)} := \Delta^{(L)} + \delta^{(l+1)} (a^{(l)})^T$

Then outside the FOR loop we find

$$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}, \text{ for } j \neq 0 \text{ and}$$
$$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} \text{ for } j = 0$$

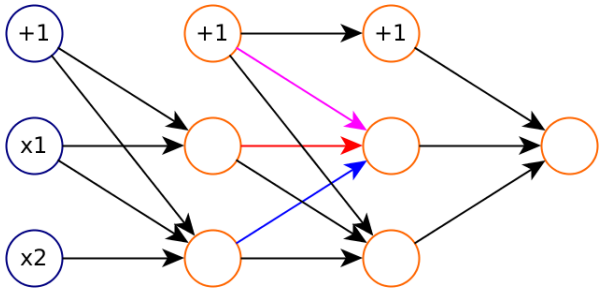
The $j = 0$ term is the bias term. Doing this finds

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

Ewwww. :/.

3 Backpropagation Intuition

The algorithm certainly looks complicated! The exercises will walk through the steps so don't fret. Consider *forward propagation*.



This neural network has two input units, two hidden layers each with two units (not including the bias unit), and a single output unit. In forward propagation, we have...

- some input $(x^{(i)}, y^{(i)})$ that is used to calculate $z_1^{(2)}$ and $z_2^{(2)}$ in the first hidden layer.
- These are used to calculate $a_1^{(2)}$ and $a_2^{(2)}$ (using the sigmoid function).
- These values are forward propagated to the second hidden layer to calculate $z_1^{(3)}$ and $z_2^{(3)}$.
- These values are used to calculate $a_1^{(3)}$ and $a_2^{(3)}$ (using the sigmoid function).
- Finally, the output layer calculates $z_1^{(4)}$, and then $a_1^{(4)}$.

Note that each of these $z_i^{(j)}$ calculations require both units and the bias unit. The actual calculation uses the Θ matrices. In the case of $z_1^{(3)}$, the actual calculation completed is:

$$z_1^{(3)} = \Theta_{10}^{(2)} \cdot 1 + \Theta_{11}^{(2)} \cdot a_1^{(2)} + \Theta_{12}^{(2)} \cdot a_1^{(2)}$$

where

- $\Theta_{10}^{(2)} \times 1$ (pink)
- $\Theta_{11}^{(2)} \times a_1^{(2)}$ (red)
- $\Theta_{12}^{(2)} \times a_1^{(2)}$ (blue)

So what is backpropagation doing? Our cost function is still:

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

Focusing on a single example $(x^{(i)}, y^{(i)})$, with 1 output unit and ignoring regularization (by setting $\lambda = 0$), the cost is

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

How well is the network doing on this example? The cost is similar to the residual of the result. I.e.,

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

Which is essentially the error of cost for calculating $a_j^{(l)}$. We previously defined these to be 'little delta' terms given by

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \cdot \text{cost}(i)$$

Clearly, we may change the $z_j^{(l)}$ terms which will effect the cost. These are the *weight* terms that are used by the algorithm.

We perform *back propagation* to calculate these error terms as follows:

- $\delta_1^{(4)}$ is the last error term (on the output later).
- The $\delta_1^{(4)}$ term is used to back calculate the error terms $\delta_1^{(3)}$ and $\delta_2^{(3)}$.
- $\delta_1^{(3)}$ and $\delta_2^{(3)}$ are used to back calculate $\delta_1^{(2)}$ and $\delta_2^{(2)}$

In the case of $\delta_2^{(2)}$, we have to find these weighted sums:

$$\delta_2^{(2)} = \Theta_{12}^{(1)} \cdot \delta_1^{(3)} + \Theta_{12}^{(2)} \cdot \delta_2^{(3)}$$

in the case of $\delta_2^{(3)}$, we have to find these weighted sum:

$$\delta_2^{(3)} = \Theta_{12}^{(3)} \cdot \delta_1^{(4)}$$

The error terms for the bias units are sometimes calculated as well.

4 Backpropagation in Practice: "Unrolling Parameters"

We will need to unroll parameters from a matrix into vectors. Say we are going to implement a function in Octave to take 'theta' and return the cost function value 'jVal' and the gradients. Then these values are passed into an advanced optimization function like 'fminunc', etc. We have the matrices $\Theta^{(1)}, \Theta^{(2)}$, and $\Theta^{(3)}$, called `Theta1, Theta2`, and `Theta3`, as follows:

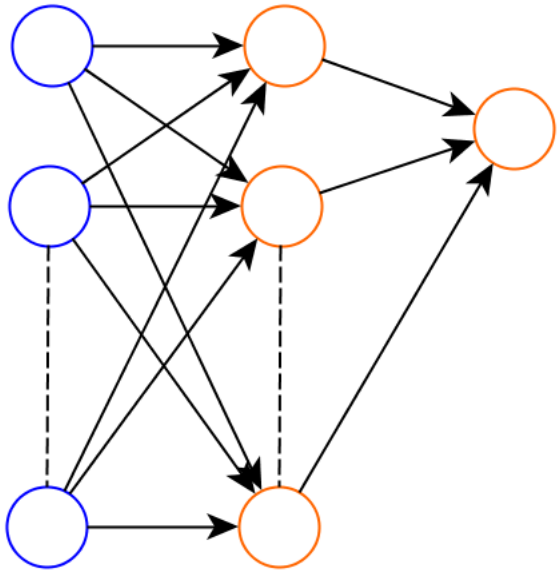
```
function [jVal, gradient] = costFunction(theta)
...

optTheta = fminunc(@costFunction, initialTheta, options)
```

The assumption here is that `theta` and `gradient` is an $n+1$ vector. This works fine for what we did previously, but now `Theta` are each matrices.

Here is an example:

- Suppose we have $s_1 = 10$ units for layer 1, $s_2 = 10$ units for layer 2, and $s_3 = 1$ unit for the output layer.



Our matrices will be of these dimensions:

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

- In Octave, we may take each of these and use the following code to turn them into vectors:

```
thetaVec = [ Theta1(:); Theta2(:); Theta3(:) ];
DVec = [ D1(:); D2(:); D3(:) ];
```

which will put the matrices into a few long vectors. Then use the `reshape` command to put them back:

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

- Here is an example you can run in Octave:

```
Theta1 = ones(10,11);
Theta2 = 2*ones(10,11);
Theta3 = 3*ones(1,11);
ThetaVec = [Theta1(:); Theta2(:); Theta3(:)];
reshape(ThetaVec(1:110), 10,11);
```

- We will have to do this in the programming assignment:
  1. Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
  2. Unroll these to get `initialTheta` to pass to
  3. `fminunc(@costFunction, initialTheta, options)`
  4. Then implement the cost function:

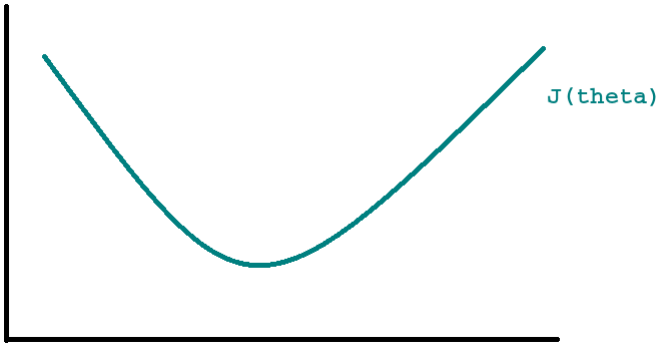
     ```
     function [jVal, gradientVec] = costFunction(thetaVec)
     ```

     (a) From `thetaVec`, get $\Theta^{(1)}, \Theta^{(2)}$, and $\Theta^{(3)}$.
     (b) Use forward propagation / back propagation to compute `D1, D2, D3` and $J(\Theta)$.
     (c) Unroll `D1, D2, D3` to get `gradientVec`

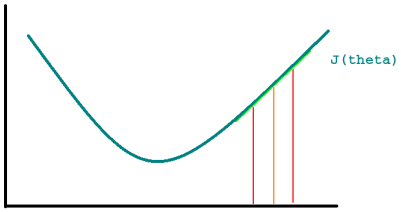
There are advantages to storing all this as vectors or as matrices, respectively.

5 Backpropagation in Practice: "Numerical Gradient Checking"

The algorithm we discussed is complicated and may have hidden errors in the implementation. We will use *Gradient Checking* to ensure the algorithm is working correctly. Here is an example:



Suppose this is a plot of $J(\Theta)$ where $\Theta \in \mathbb{R}$. We can get a numerical approximation of the derivative using a secant line.



We approximate the secant line with the centered difference

$$\frac{d}{d\Theta} \approx \frac{J(\Theta + \varepsilon) - J(\Theta - \varepsilon)}{2\varepsilon}$$

We set $\varepsilon \approx 10^{-4}$. We can easily implement this in Octave using

```
gradApprox = (J(theta + epsilon) - J(theta - epsilon))/2*epsilon
```

In a more general case, say $\theta \in \mathbb{R}^n$.

$$\theta = [\theta_1, \theta_2, ..., \theta_n]$$

The partial derviatives are

$$\begin{aligned}
\frac{\partial}{\partial \theta_1} J(\theta) &\approx \frac{J(\theta_1 + \varepsilon, \theta_2, ..., \theta_n) - J(\theta_1 - \varepsilon, \theta_2, ..., \theta_n)}{2\varepsilon} \\
\frac{\partial}{\partial \theta_2} J(\theta) &\approx \frac{J(\theta_1, \theta_2 + \varepsilon, ..., \theta_n) - J(\theta_1, \theta_2 - \varepsilon, ..., \theta_n)}{2\varepsilon} \\
&\vdots \\
\frac{\partial}{\partial \theta_1} J(\theta) &\approx \frac{J(\theta_1, \theta_2, ..., \theta_n + \varepsilon) - J(\theta_1, \theta_2, ..., \theta_n - \varepsilon)}{2\varepsilon}
\end{aligned}$$

In Octave, we use the unrolled parameter vector θ:

```
for i = 1:n,
        thetaPlus = theta;
        thetaPlus(i) = thetaPlus(i) + epsilon;
        thetaMinus = theta;
        thetaMinus(i) = thetaMinus(i) - epsilon;
        gradApprox(i) = (J(thetaPlus)-J(thetaMinus))/2*epsilon
end;
```

which is used to

- Check that `gradApprox` $\approx$ `DVec`

where `DVec` is calculated using backward propagation.

Implementation:

- Implement backpropagation to compute `DVec`, which is the unrolled vector containing $D^{(1)}, D^{(2)}$, and $D^{(3)}$.
- Implement a numerical gradient check to compute `gradApprox`
- Compare these and obtain `gradApprox` $\approx$ `DVec`
- Turn off gradient checking. Use the confirmed backpropagation code to do the learning.

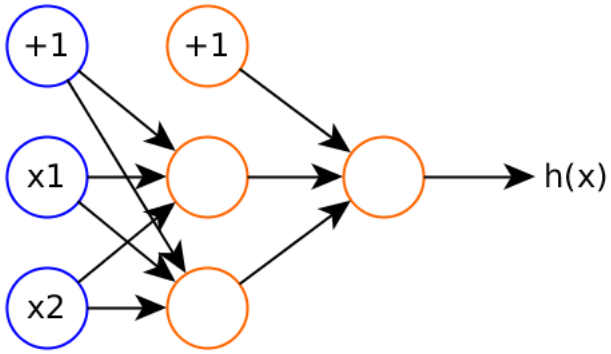
It is important to disable your gradient checking code before training your classifier! If you don't the code will run painfully slow. Backpropagation is efficient - but this checking step is computationally expensive.

6 Backpropagation in Practice: "Random Initialization"

For gradient descent and many advanced optimization methods we need initial values for Θ. I.e.,

```
optTheta = fminunc(@costFunction, initialTheta, options)
```

Should we set `initialTheta = zeros(n,1)`? This worked when we were doing logistic regression. What happens, however, when we do this with a neural network?



If we set $\Theta_{ij}^{(l)} = 0$ for all i, j, l, then we will necessarily have $a_1^{(2)} = a_2^{(1)}$, $\delta_1^{(2)} = \delta_2^{(2)}$, and $\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$. Even after one pass of gradient descent, the weights $\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$. Thus parameters corresponding to inputs going into each of the two hidden weights will stay the same - and the NN cannot calculate the anything interesting. This kind of implementation is performing redundant calculations rather than calculating separate inputs.

A random initialization is *symmetry breaking* in that it won't treat inputs as the same parallel input. We initialize:

$$\Theta_{ij}^{(l)} \in \{-\varepsilon, \varepsilon\}$$
$$\Leftrightarrow$$
$$-\varepsilon \leq \Theta_{ij}^{(l)} \leq \varepsilon$$

This may be done in Octave using

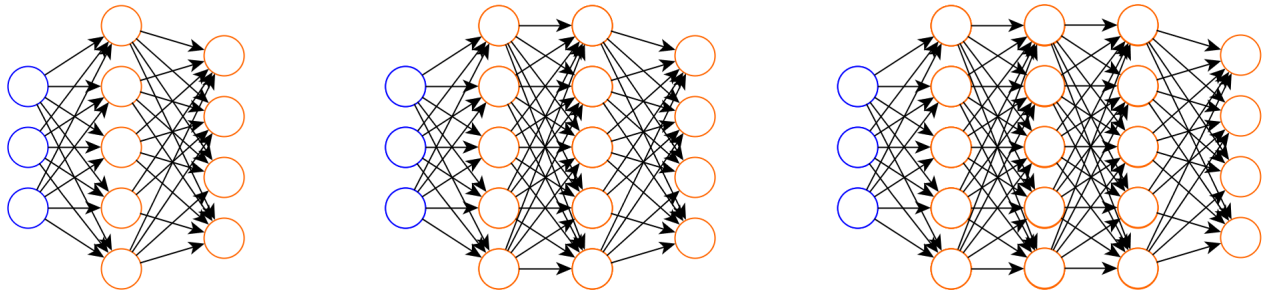
```
Theta1 = rand(10,11)*(2*initEpsilon) - initEpsilon;
Theta2 = rand(1,11)*(2*initEpsilon) - initEpsilon;
```

`rand(10,11)` calculates a 10×11 matrix where each element is in $[0, 1]$.

7 Backpropagation in Practice: "Putting it Together"

Here's a summary of the neural network learning algorithm.

1. Pick a network architecture.



- The number of input units (usually fixed) is the dimension of the features $x^{(i)}$.
- The number of output units is the number of classes.
- The number of hidden layers should be at least 1. There should be the same number of hidden units in every hidden layer.

2. Train the neural network.

(i) Randomly initialize weights

(ii) Implement forward propagation to get $h_\Theta\left(x^{(i)}\right)$ for any $x^{(i)}$.

(iii) Implement code to compute the cost function $J(\Theta)$.

(iv) Implement backward propagation to compute the partial derivatives

$$\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$$

We usually do this with a FOR loop:

```
for i = 1:m
```

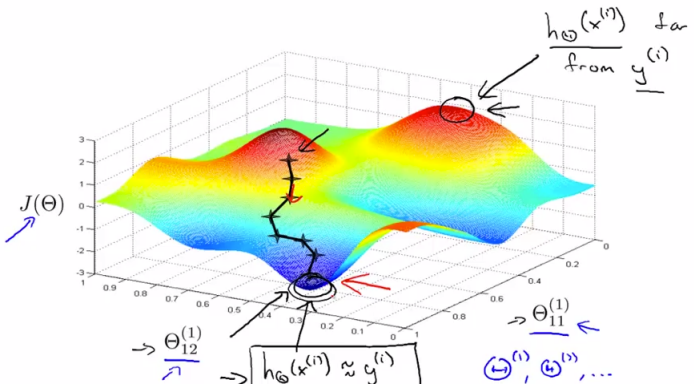
{
- Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$
- This will get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, ..., L$
}

(v) Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}}$ computed using backpropagation vs. the numerical gradient estimates of the gradient of $J(\Theta)$.

Then *disable the gradient checking code.*

(vi) Use gradient descent of advanced optimization with backpropagation (calculates partial derivatives) to try to minimize the cost $J(\Theta)$ as a function of the parameters Θ.

Note: This $J(\Theta)$ is non-convex – but this does not often create problems in practice.



8 Application of Neural Network Learning: Automomous Driving

Self-driving cars use Neural Network Learning to learn from humans how to drive.

- First, the NN learns to stear by watching a human driver turn. The NN learns to have the same stearing direction as the human. The same procedure is used on different types of roadways.
- Several times per second the computer makes a stearing decision and outputs a measure of it's confidence in the stearing direction it chose.
- The computer is then allowed to drive...