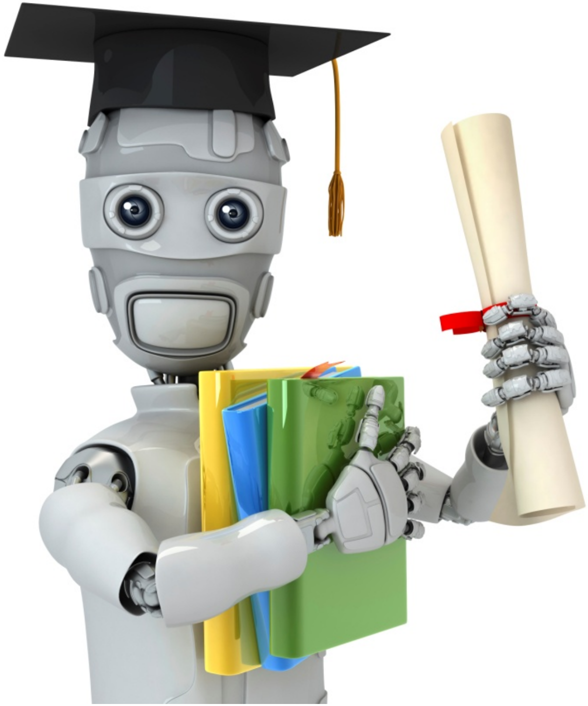


Machine Learning

The most powerful learning algorithm we have today!!!

# Neural Networks: Learning

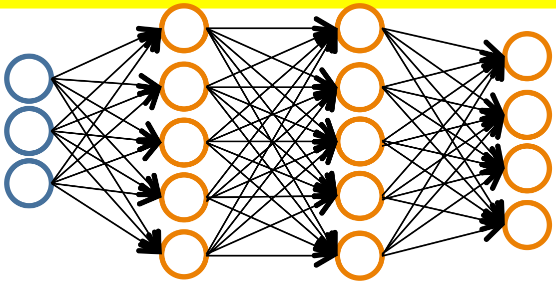
## Cost function

We focus on NN to the application of classification problems!

# Neural Network (Classification)

m training examples!

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



Layer 1 Layer 2 Layer 3 Layer 4

L = 4

$L =$ total no. of layers in network

s1 = 3, s2 = 5

$s_l =$ no. of units (not counting bias unit) in layer $l$

We consider two types of classification problems

## Binary classification

$y = 0 \text{ or } 1$

1 output unit

either 0 or 1

K = 1, K denotes number of units in output layer

## Multi-class classification (K classes)

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

K output units

K = 4 in this case

# Cost function

## Logistic regression:

regularization term

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

## Neural network:

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

training data set

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right]$$

Have K output units, similar to logistic regression, where we only have 1 output

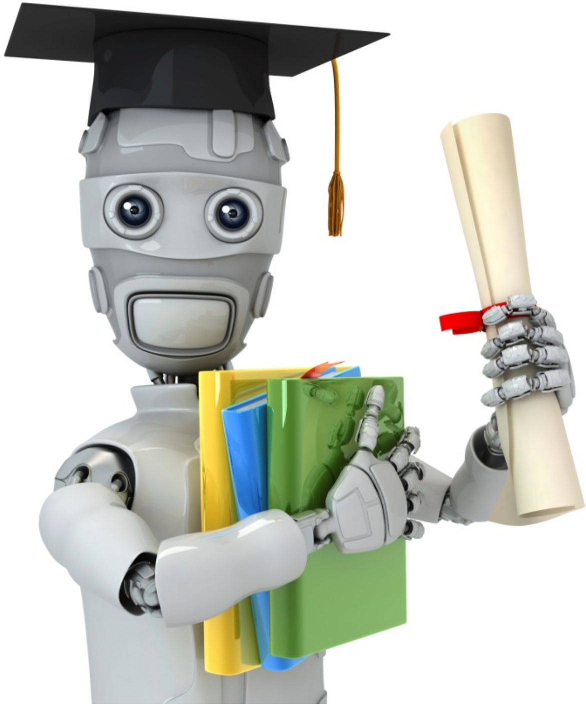
outputs in layer l+1

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

we do not sum the i=0 term!

Layers

nodes in each layer, we exclude the bias node as we did in logistic regression



Machine Learning

# Neural Networks: Learning

## Backpropagation algorithm

The algorithm for trying to minimize the cost function!

# Gradient computation

our previously defined cost function

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

$$\min_\Theta J(\Theta)$$

objective

Need code to compute:

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

we need these two terms to run the optimization programs

$\Theta_{ij}^{(l)} \in \mathbb{R}$

we will focus on how to compute these partial derivative terms

# Gradient computation

consider only one training example

Given one training example $(x, y)$:

## Forward propagation:

first we do is to apply forward propagation

$$a^{(1)} = x$$

also add bias term here!

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

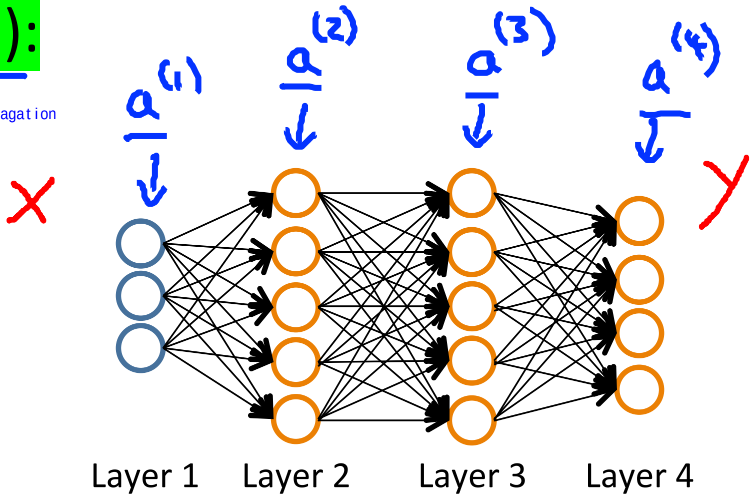
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$

$$z^{(3)} = \Theta^{(2)} a^{(2)}$$

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

$$z^{(4)} = \Theta^{(3)} a^{(3)}$$

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



we use backpropagation to compute the derivatives!

backpropagation refers to here
we compute the error term backwards

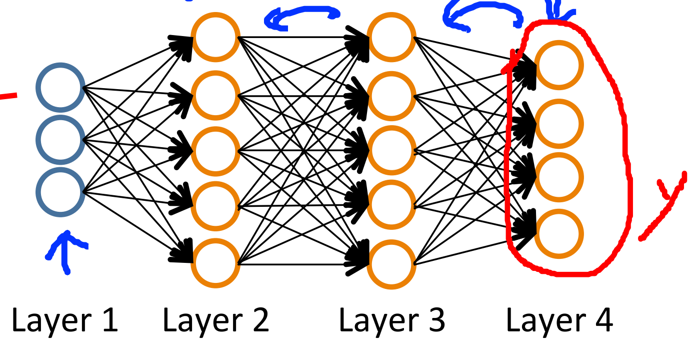
# Gradient computation: Backpropagation algorithm

Intuition: $\delta_j^{(l)} =$ "error" of node $j$ in layer $l$.

somehow represent the error term!

error term in 4th layer

$\delta^{(2)}$ $a_j^{(l)}$ $\delta^{(3)}$ $\delta^{(4)}$




4 th layer

For each output unit (layer L = 4)

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

$(h_\Theta(x))_j$  $\delta^{(4)} = a^{(4)} - y$

vector implementation

derivative

4x5  4x1  5x1

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

$a^{(3)} .* (1 - a^{(3)})$

$a^{(2)} .* (1 - a^{(2)})$

(No $\delta^{(1)}$)

1st layer is the input layer,
it does not have any error associated with it

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

delta*a' in vector form

(ignoring $\lambda$; if $\lambda = 0$)

ignore the regularization term

we can quickly compute the partial derivatives

we can prove this!

procedures

# Backpropagation algorithm

A simple version of backpropagation algorithm derivation can be found in the note!

now we have large training set!

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

Set $\triangle_{ij}^{(l)} = 0$ (for all $l, i, j$).

(use to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$)

For $i = 1$ to $m$

$(x^{(i)}, y^{(i)})$

move through our training set

Set $a^{(1)} = x^{(i)}$

Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

Using $y^{(i)}$, compute $\boxed{\delta^{(L)} = a^{(L)} - y^{(i)}}$

error in the output layer

Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ ~~$\delta^{(1)}$~~

$\boxed{\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}}$

vector form

$\boxed{\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T}$

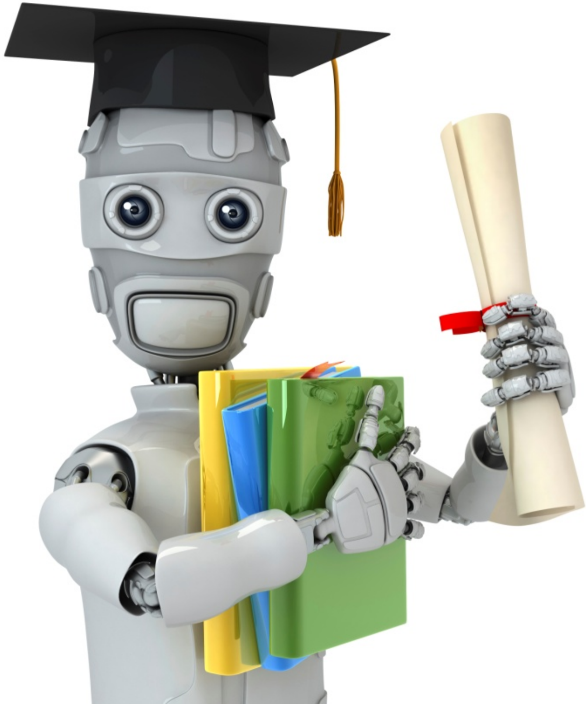
★!

$$\boxed{\begin{aligned} D_{ij}^{(l)} &:= \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \quad \text{if } j \neq 0 \\ D_{ij}^{(l)} &:= \frac{1}{m} \triangle_{ij}^{(l)} \quad \text{if } j = 0 \end{aligned}}$$

with regularization
this computation is outside of the for loop!

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

once we have the partial derivatives, we can use them in the gradient descent or other optimization algorithms



Machine Learning

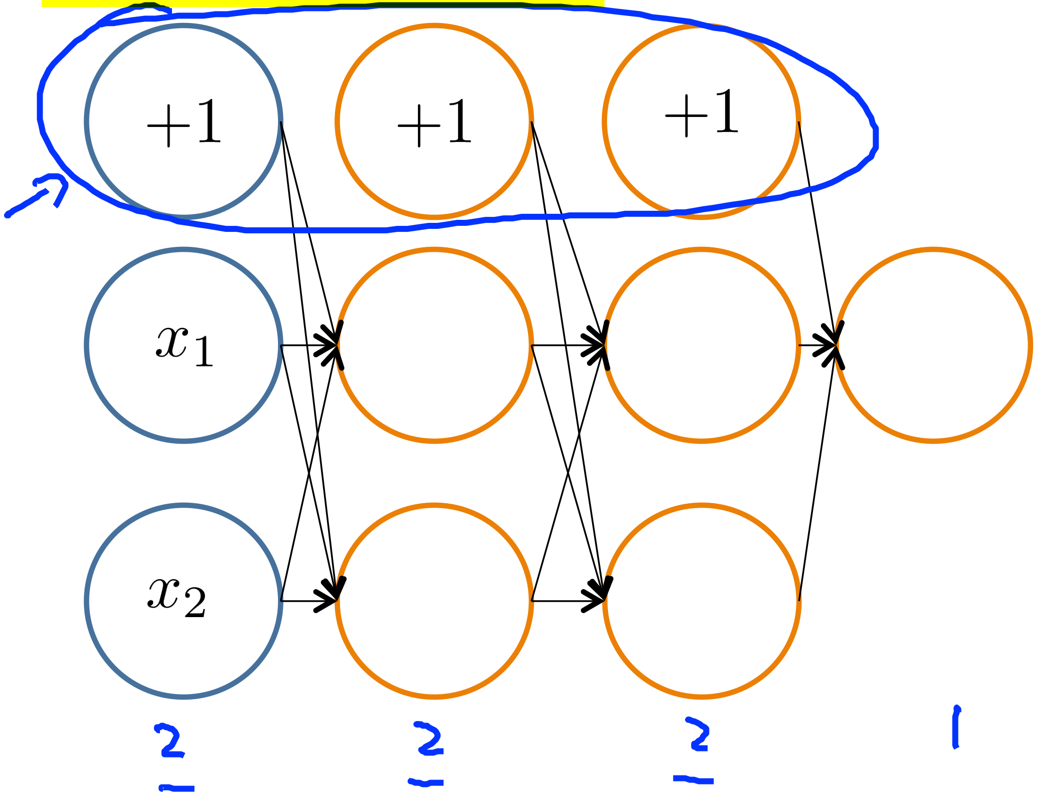
# Neural Networks: Learning

## Backpropagation intuition

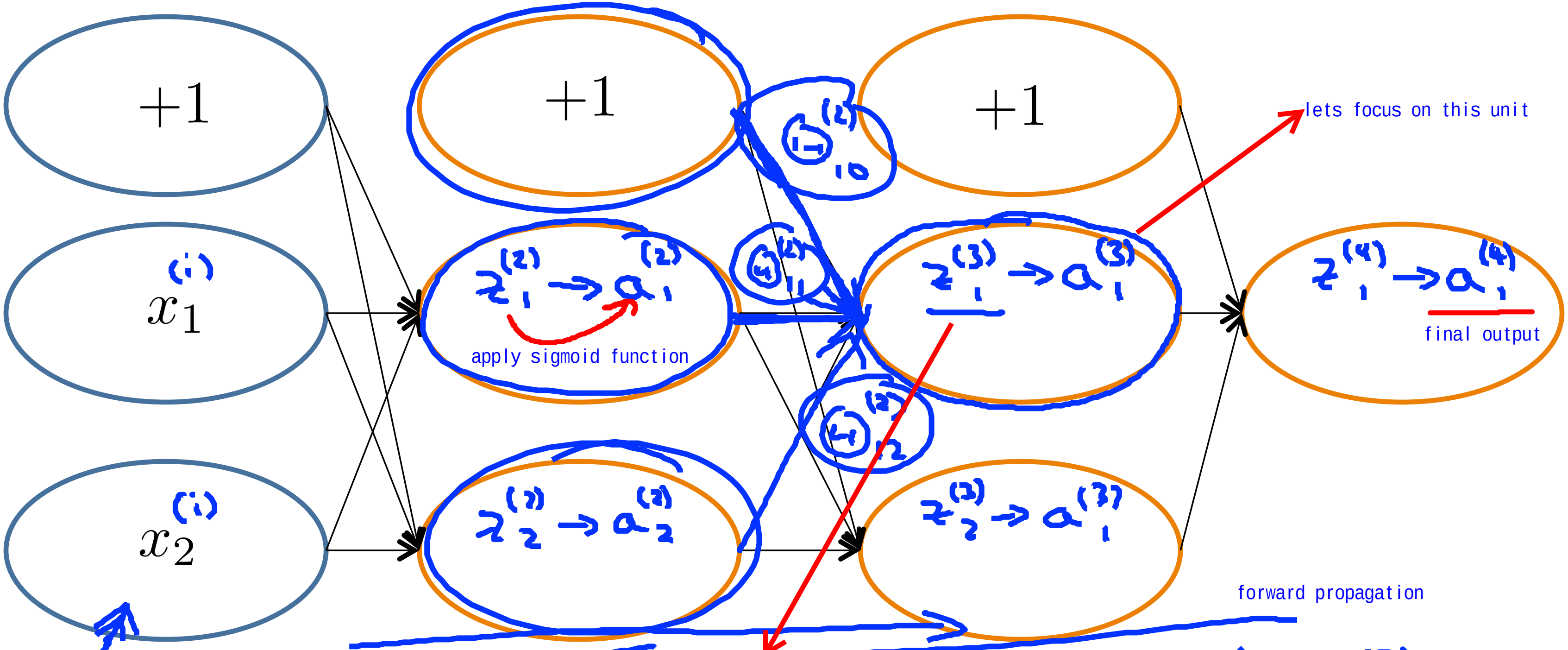
In this section, we will look at the mechanical steps of backpropagation!

# Forward Propagation

not counting the bias unit



# Forward Propagation




$(x^{(i)}, y^{(i)})$

$$z^{(3)}_1 = \Theta^{(2)}_{10} \times 1 + \Theta^{(2)}_{11} a^{(2)}_1 + \Theta^{(2)}_{12} \cdot a^{(2)}_1$$

# What is backpropagation doing?

to better understand what backpropagation is doing!

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} \underbrace{y^{(i)}\log(h_\Theta(x^{(i)})) + (1-y^{(i)})\log(1-(h_\Theta(x^{(i)})))}_{(x^{(i)},\, y^{(i)})}\right]$$

$$\cancel{+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2}$$

Focusing on a single example $x^{(i)}$, $y^{(i)}$, the case of 1 output unit, and ignoring regularization ($\lambda = 0$),

think in this way!

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

$y^{(i)}$

(Think of $\text{cost}(i) \approx (\boxed{h_\Theta(x^{(i)})} - \boxed{y^{(i)}})^2$ )

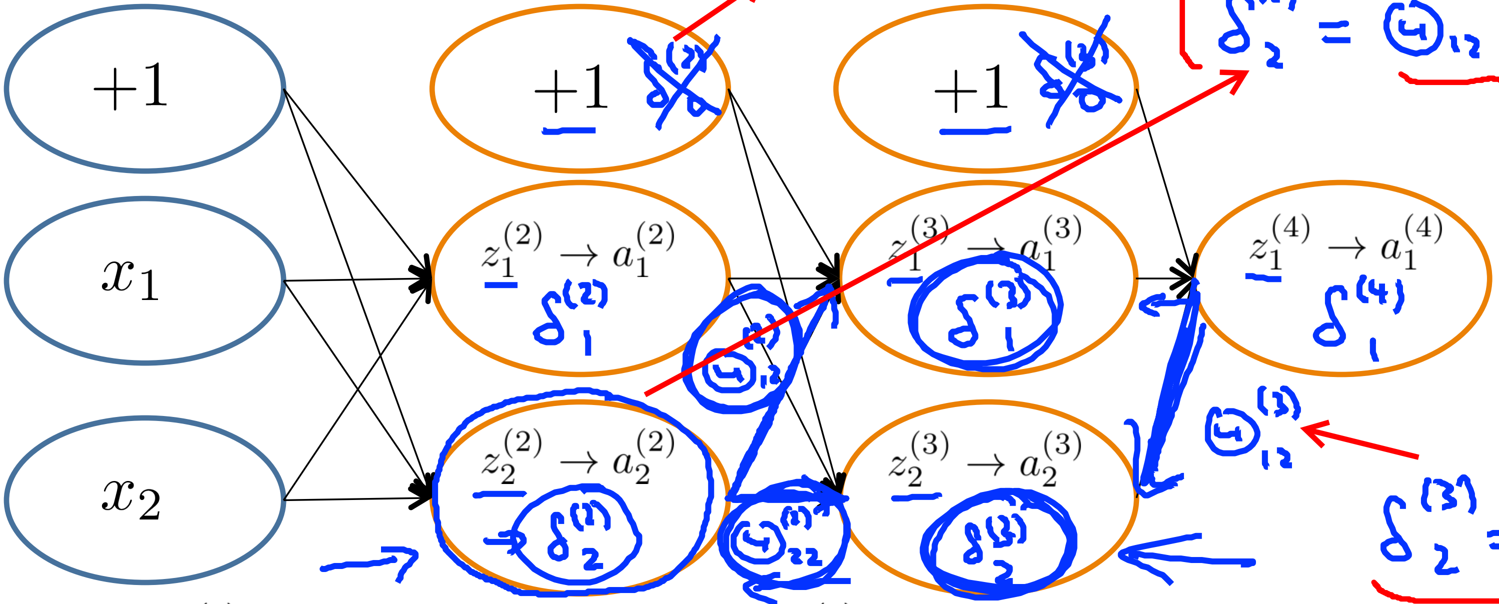
I.e. how well is the network doing on example i?

# Forward Propagation

we do not use weights for bias values

$$\delta_1^{(4)} = y^{(i)} - a_1^{(4)}$$

$$\delta_2^{(2)} = \underbrace{\Theta_{12}^{(2)} \delta_1^{(3)} + \Theta_{22}^{(2)} \delta_2^{(3)}}$$

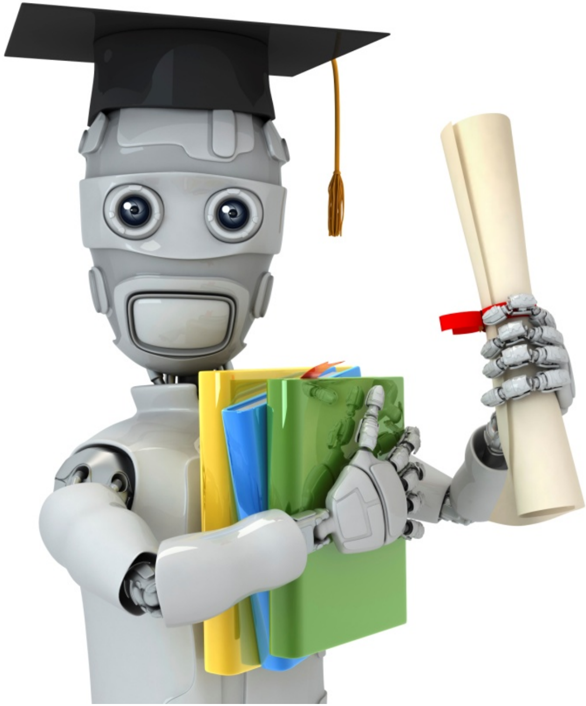



$$\delta_2^{(3)} = \underbrace{\Theta_{12}^{(3)} \cdot \delta^{(4)}}.$$

→ $\delta_j^{(l)}$ = "error" of cost for $a_j^{(l)}$ (unit $j$ in layer $l$).

Formally, $\boxed{\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(i)}$ (for $j \geq 0$), where

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



Machine Learning

# Neural Networks: Learning

## Implementation note: Unrolling parameters

展开

from matrices to vectors, which we need in order to use the advanced optimization routines

# Advanced optimization

```
function [jVal, gradient] = costFunction(theta)
   ...
optTheta = fminunc(@costFunction, initialTheta, options)
```

$\mathbb{R}^{n+1}$ takes vector input $\mathbb{R}^{n+1}$ (vectors)

advanced optimization algorithm

Neural Network (L=4):

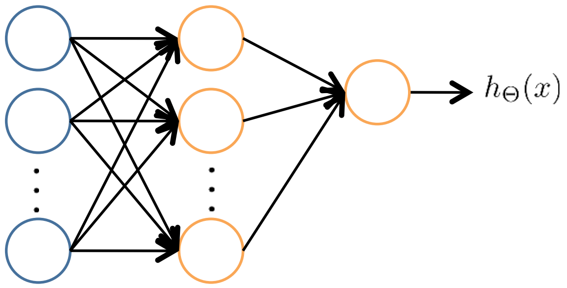
→ $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (Theta1, Theta2, Theta3)

→ $D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (D1, D2, D3)

"Unroll" into vectors

main idea here!
bascially is convert matrices to vectors for computational purpose!

# Example




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

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

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

$\Theta^{(1)}$ $\Theta^{(2)}$ $\Theta^{(3)}$

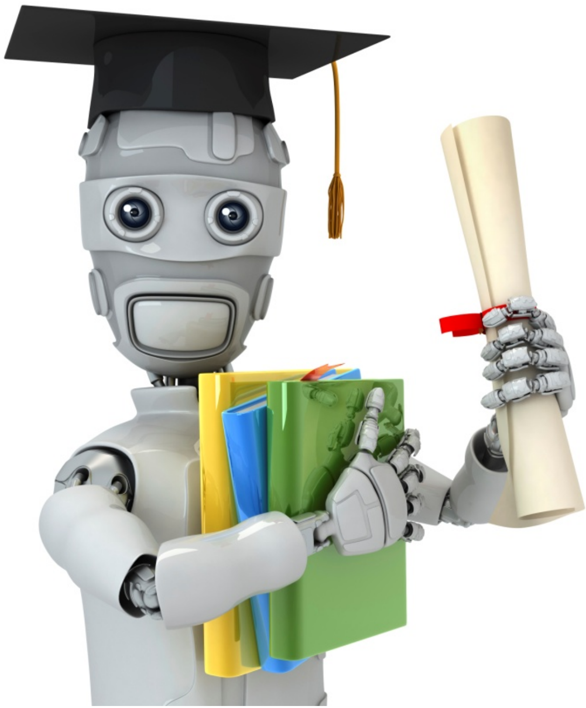
```
thetaVec = [ Theta1(:); Theta2(:); Theta3(:)];
DVec = [D1(:); D2(:); D3(:)];

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)`

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

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

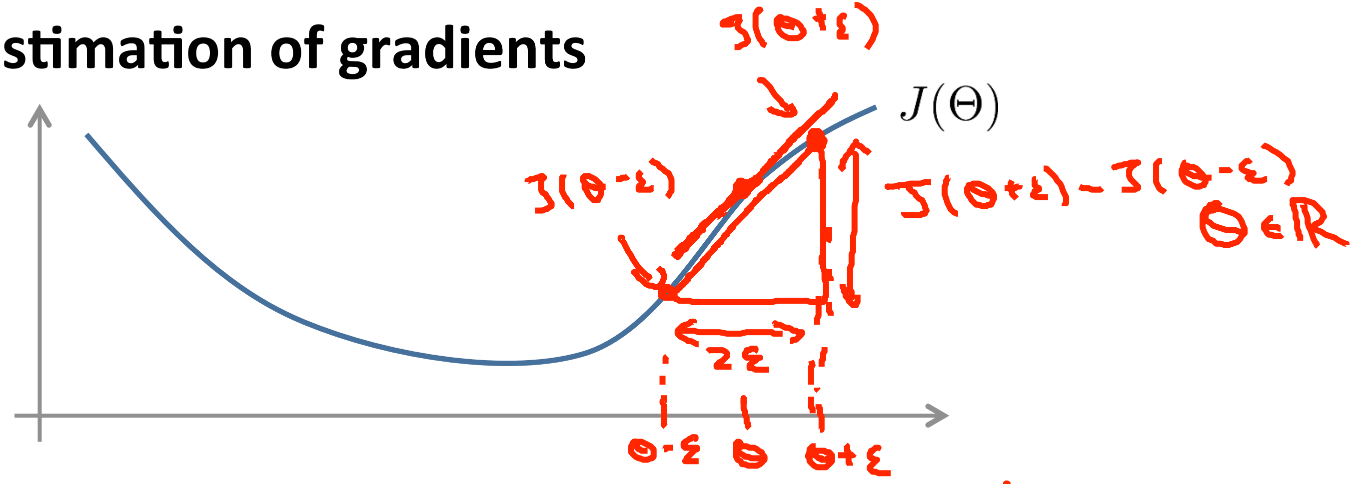


Machine Learning

# Neural Networks: Learning

## Gradient checking

# Numerical estimation of gradients




$$\frac{d}{d\theta}J(\theta) \approx \underbrace{\frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}} \qquad \cancel{\frac{J(\theta+\epsilon)-J(\theta)}{\epsilon}}$$

$$\epsilon = 10^{-4}$$

Implement:
```
gradApprox = (J(theta + EPSILON) – J(theta – EPSILON))
                                      /(2*EPSILON)
```

## Parameter vector $\theta$

- $\theta \in \mathbb{R}^n$ (E.g. $\theta$ is "unrolled" version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ )
- $\theta = [\theta_1, \theta_2, \theta_3, \ldots, \theta_n]$

$$\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\boxed{\theta_1 + \epsilon}, \theta_2, \theta_3, \ldots, \theta_n) - J(\boxed{\theta_1 - \epsilon}, \theta_2, \theta_3, \ldots, \theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \boxed{\theta_2 + \epsilon}, \theta_3, \ldots, \theta_n) - J(\theta_1, \boxed{\theta_2 - \epsilon}, \theta_3, \ldots, \theta_n)}{2\epsilon}$$

$$\vdots$$

$$\frac{\partial}{\partial \boxed{\theta_n}} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \ldots, \boxed{\theta_n + \epsilon}) - J(\theta_1, \theta_2, \theta_3, \ldots, \boxed{\theta_n - \epsilon})}{2\epsilon}$$

```
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;
```

$$\begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_i + \varepsilon \\ \vdots \\ \theta_n \end{bmatrix} \rightarrow \theta_i - \varepsilon$$

$$\frac{\partial}{\partial \theta_i} J(\theta).$$

Check that $\boxed{\texttt{gradApprox}} \approx \boxed{\texttt{DVec}}$

From backprop.

**Implementation Note:**

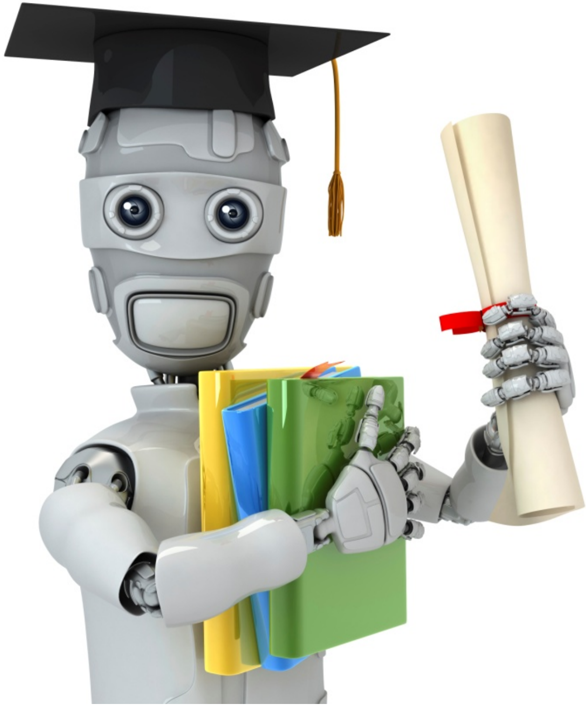
- Implement backprop to compute `DVec` (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
- Implement numerical gradient check to compute `gradApprox`.
- Make sure they give similar values.
- Turn off gradient checking. Using backprop code for learning.

DVec

$\delta^{(4)}, \delta^{(3)}, \delta^{(2)}$

**Important:**

- Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of `costFunction(…)`)your code will be <u>very</u> slow.



Machine Learning

# Neural Networks: Learning

## Random initialization

## Initial value of $\Theta$

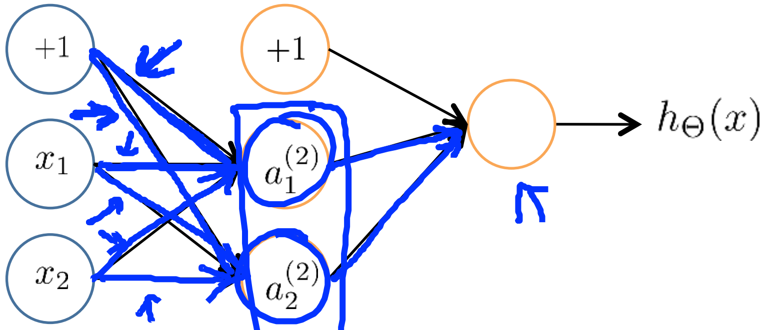
For gradient descent and advanced optimization method, need initial value for $\Theta$.

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

Consider gradient descent

Set `initialTheta = zeros(n,1)` ?

# Zero initialization




$$\Theta_{ij}^{(l)} = 0 \text{ for all } i, j, l.$$

$a_1^{(2)} = a_2^{(2)}$. Also $\delta_1^{(2)} = \delta_2^{(2)}$.

$$\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$$

$$\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$$

After each update, parameters corresponding to inputs going into each of two hidden units are identical.

$a_1^{(2)} = a_2^{(2)}$

# Random initialization: Symmetry breaking

- Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
(i.e. $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$ )

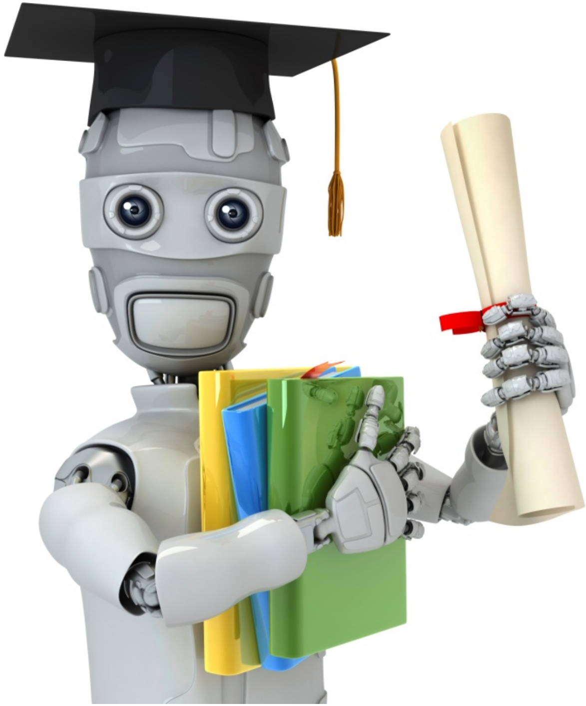
E.g.

random 10×11 matrix (betw. 0 and 1)

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

$[-\epsilon, \epsilon]$

```
Theta2 =  rand(1,11)*(2*INIT_EPSILON)
          - INIT_EPSILON;
```



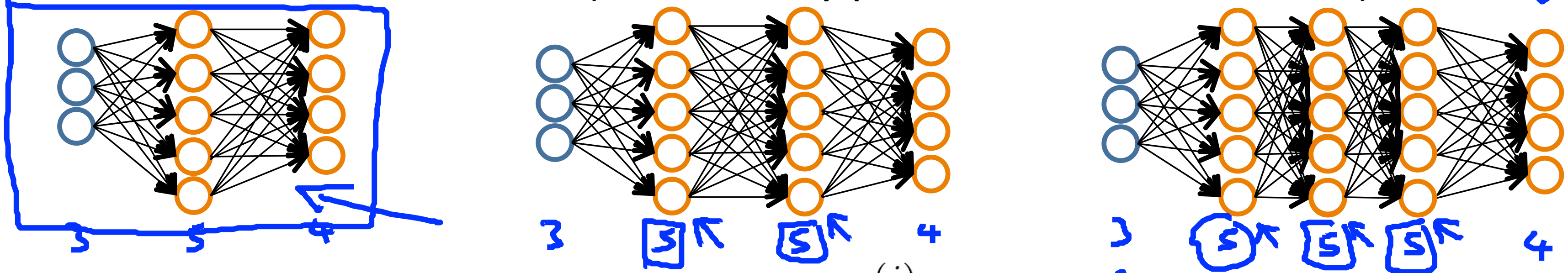
Machine Learning

# Neural Networks: Learning

## Putting it together

# Training a neural network

Pick a network architecture (connectivity pattern between neurons)




- No. of input units: Dimension of features $x^{(i)}$
- No. output units: Number of classes

Reasonable default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden units in every layer (usually the more the better)

$y \in \{1, 2, 3, \dots, 10\}$

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

## Training a neural network

1. Randomly initialize weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement code to compute cost function $J(\Theta)$
4. Implement backprop to compute partial derivatives $\boxed{\frac{\partial}{\partial \Theta^{(l)}_{jk}} J(\Theta)}$

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

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

Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$

(Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \dots, L$).

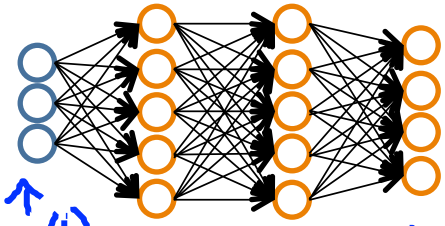
$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

....

}

...

compute $\frac{\partial}{\partial \Theta^{(l)}_{jk}} J(\Theta)$.



$x^{(i)}$

Andrew Ng

## Training a neural network

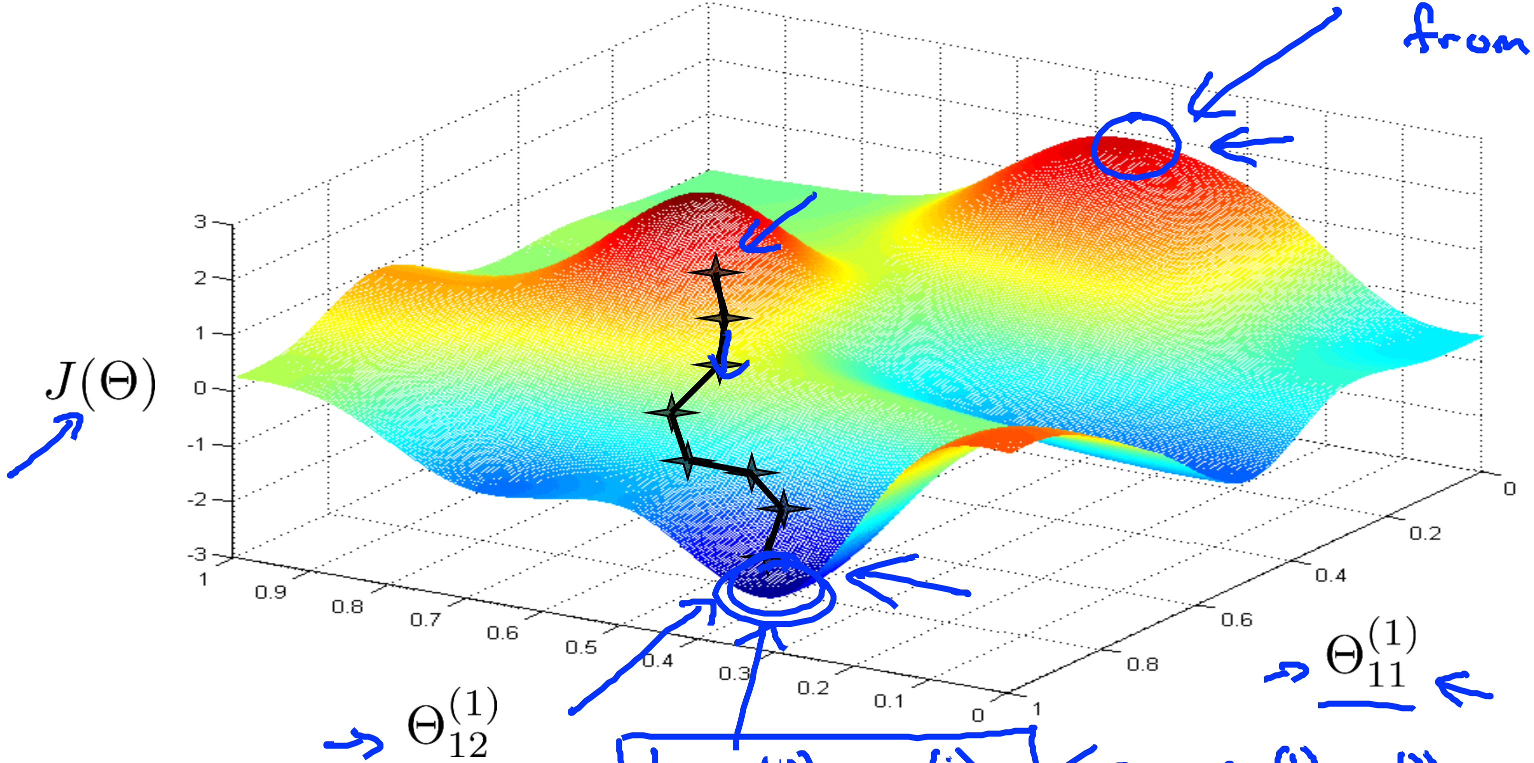
5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient of $J(\Theta)$.

   Then disable gradient checking code.

6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters $\Theta$

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

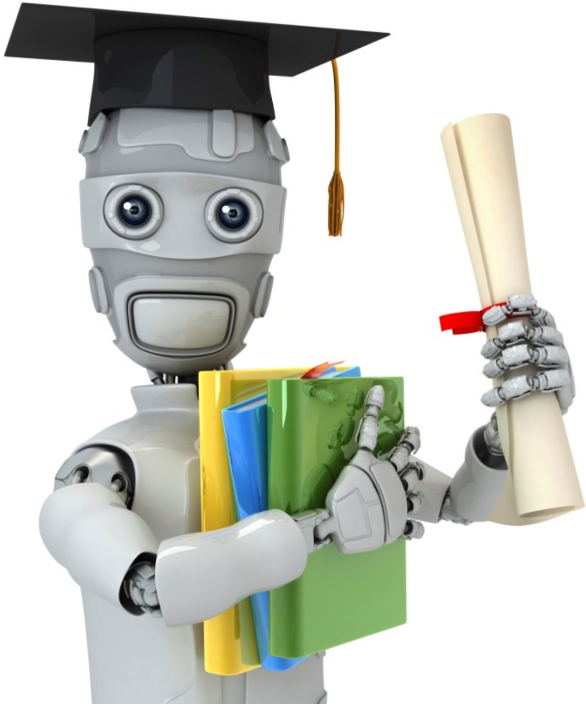
$J(\Theta)$ — non-convex.




$h_\Theta(x^{(i)})$ far from $y^{(i)}$

$\boxed{h_\Theta(x^{(i)}) \approx y^{(i)}}$

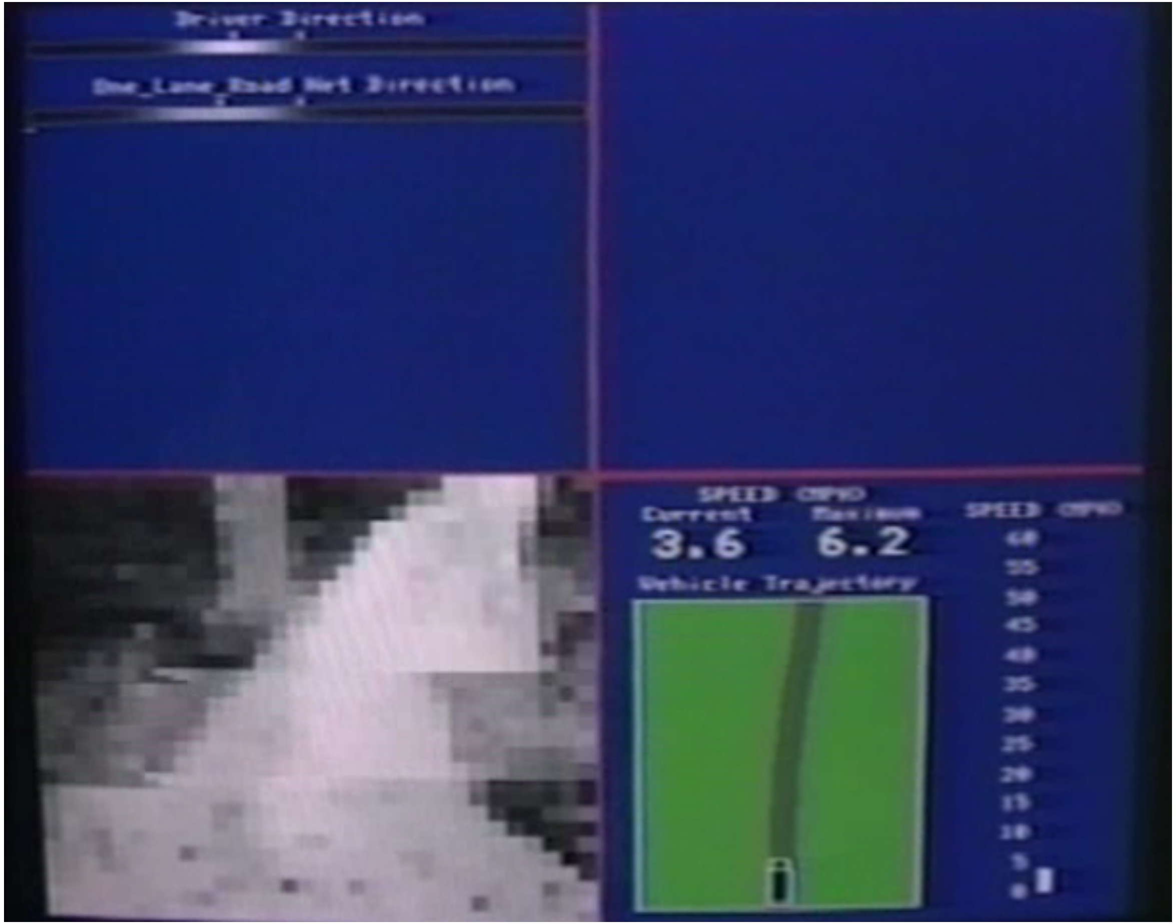
$\Theta^{(1)}, \Theta^{(2)}, \ldots$



Machine Learning

# Neural Networks: Learning

## Backpropagation example: Autonomous driving (optional)




[Courtesy of Dean Pomerleau]