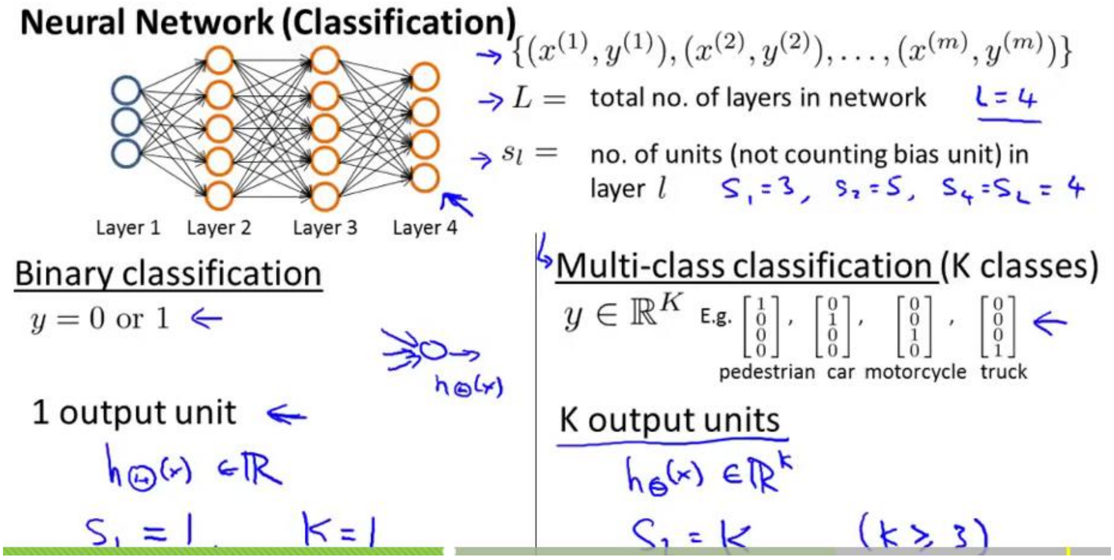
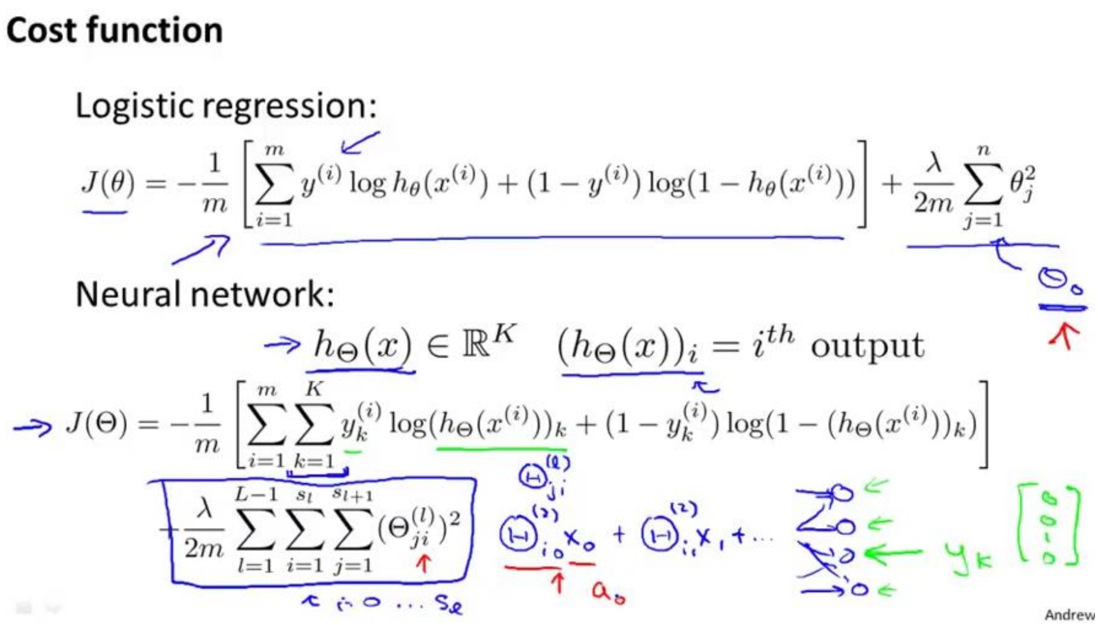


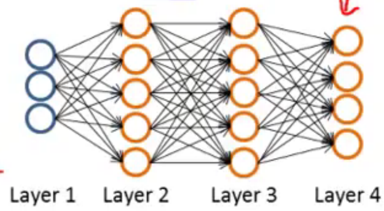


反向传播算法 Backpropagation

Gradient computation: Backpropagation algorithm

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

$\delta^{(4)}$



For each output unit (layer L = 4)

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

$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$ $a^{(3)} .* (1 - a^{(3)})$

$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$ $a^{(2)} .* (1 - a^{(2)})$

Backpropagation algorithm

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

Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j).   (used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$)

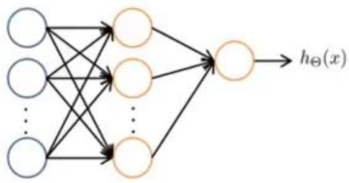
For $i = 1$ to m ← $(x^{(i)}, y^{(i)})$.

- Set $a^{(1)} = x^{(i)}$
- Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots, L$
- Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
- Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$ $\cancel{\delta^{(1)}}$
- $\boxed{\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}}$ ← $\triangle^{(l)} := \triangle^{(l)} + \delta^{(l+1)} (a^{(l)})^T$.

$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$

$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)}$ if $j = 0$

Example



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

$\boxed{\Theta^{(1)} \in \mathbb{R}^{10\times11}}, \Theta^{(2)} \in \mathbb{R}^{10\times11}, \Theta^{(3)} \in \mathbb{R}^{1\times11}$

$D^{(1)} \in \mathbb{R}^{10\times11}, D^{(2)} \in \mathbb{R}^{10\times11}, D^{(3)} \in \mathbb{R}^{1\times11}$

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

```
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

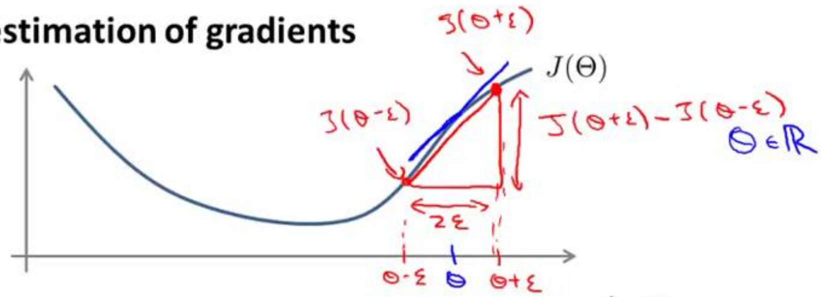
- Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
- Unroll to get `initialTheta` to pass to
- `fminunc(@costFunction, initialTheta, options)`

```
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`.

梯度 checking

Numerical estimation of gradients



$$\underline{\frac{d}{d\theta}} J(\theta) \approx \underbrace{\frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}}$$

$\varepsilon = 10^{-4}$

$$\cancel{\frac{J(\theta+\varepsilon) - J(\theta)}{\varepsilon}}$$

nplement: `gradApprox = (J(theta + EPSILON) - J(theta - EPSILON)) /(2*EPSILON)`

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

```
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;
```

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

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

From back prop.

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.

随机初始化

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)

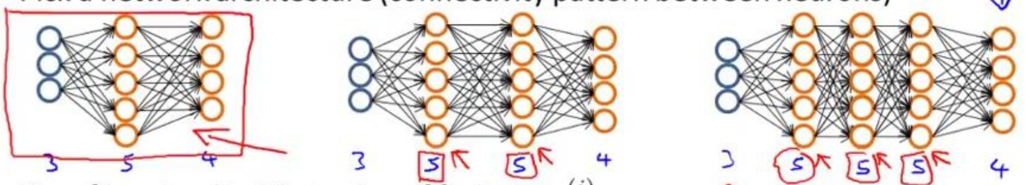
```
Theta1 =  rand(10,11)*(2*INIT_EPSILON)
          - INIT_EPSILON;
```

$[-\epsilon, \epsilon]$

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

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)

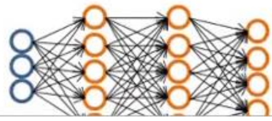
Training a neural network

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

```
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 $l = 2, \ldots, L$).



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 Θ