



Machine Learning

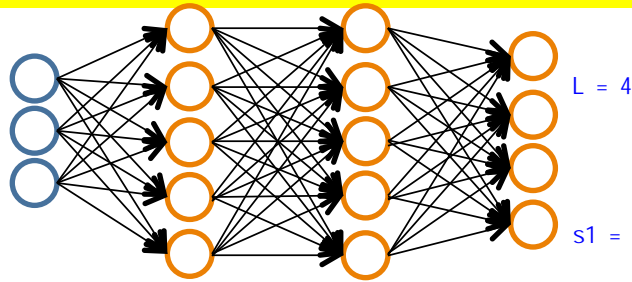
The most powerful learning algorithm we have today!!!

Neural Networks: Learning

Cost function

We focus on NN to the application of classification problems!

Neural Network (Classification)



m training examples!
 $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$

L = total no. of layers in network

s_l = no. of units (not counting bias unit) in layer l
 $s_1 = 3, s_2 = 5$

We consider two types of classification problems

Binary classification

$y = 0$ 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

④ output units

$K = 4$ in this case

Cost function

Logistic regression:

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

regularization term

Neural network:

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

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

training data set

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

outputs in layer l+1

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

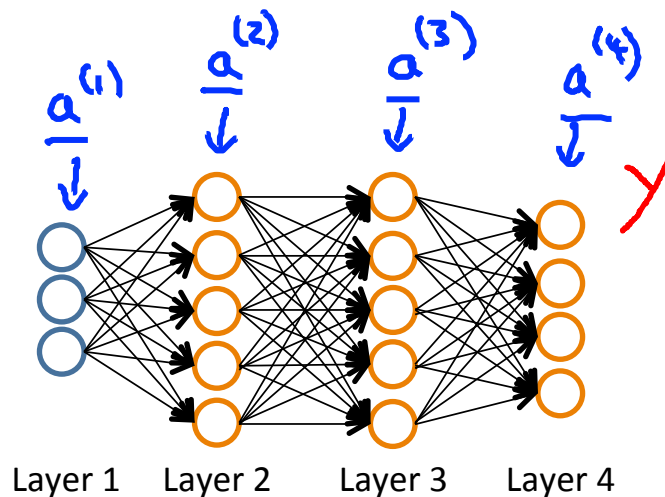
consider only one training example

Given one training example (x, y) :

Forward propagation: first we do is to apply forward propagation

- $\underline{a}^{(1)} = \underline{x}$ also add bias term here!
- $\rightarrow z^{(2)} = \Theta^{(1)} a^{(1)}$
- $\rightarrow a^{(2)} = g(z^{(2)})$ (add $\underline{a_0^{(2)}}$)
- $\rightarrow z^{(3)} = \Theta^{(2)} a^{(2)}$
- $\rightarrow a^{(3)} = g(z^{(3)})$ (add $a_0^{(3)}$)
- $\rightarrow z^{(4)} = \Theta^{(3)} a^{(3)}$
- $\rightarrow \underline{a}^{(4)} = \underline{h_{\Theta}(x)} = g(z^{(4)})$

X



we use backpropagation to compute the derivatives!

Gradient computation: Backpropagation algorithm

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

backpropagation refers to here
 we compute the error term backwards

error term in 4th layer

4 th layer

For each output unit (layer $L=4$)

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

$$(h(x))_j \quad \delta_j^{(4)} = a_j^{(4)} - y_j$$

derivative

vector implementation

Layer 1

Layer 2

Layer 3

Layer 4

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

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

$$\frac{a^{(3)} \cdot (1 - a^{(3)})}{a^{(2)} \cdot (1 - a^{(2)})}$$

(ignore λ if $\lambda = 0$)

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

delta*a' in vector form

we can quickly compute the partial derivatives

we can prove this!

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

ignore the regularization term

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

now we have large training set!

Backpropagation algorithm

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

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

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

For $i = 1$ to m ←

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

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

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

→ Using $\underline{y^{(i)}}$, compute $\delta^{(L)} = \underline{a^{(L)}} - \underline{y^{(i)}}$ → error in the output layer

→ Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$

vector form

→ $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

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

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

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

with regularization this computation is outside of the for loop!

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

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



Machine Learning

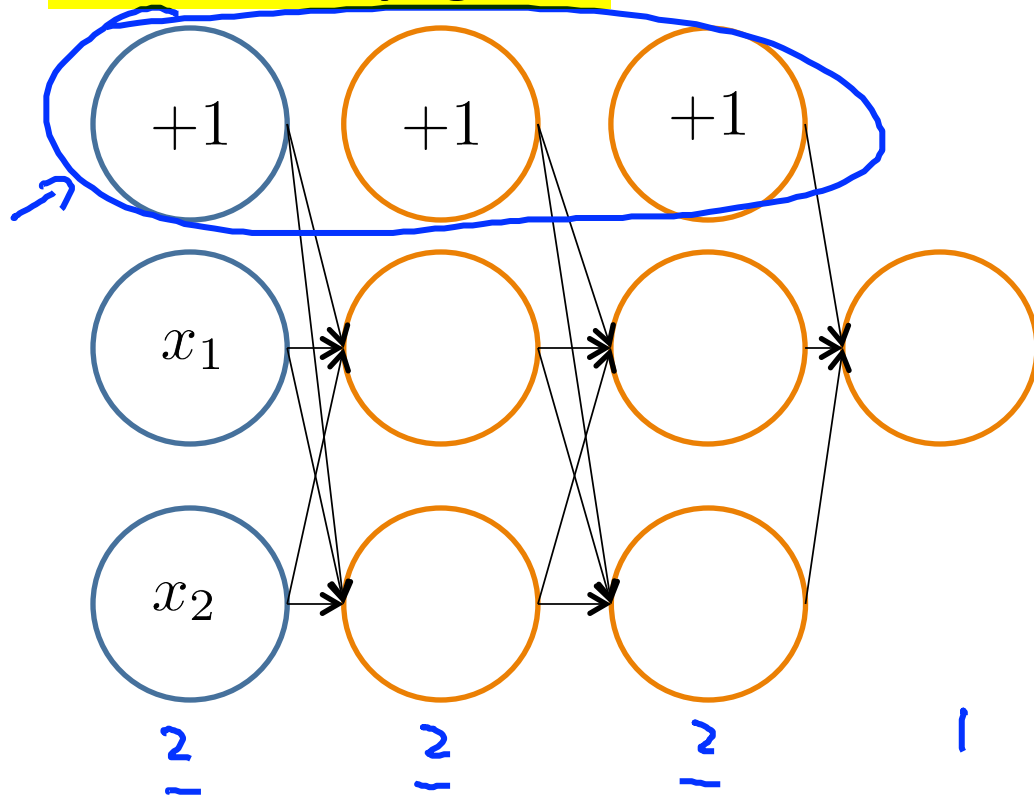
Neural Networks: Learning

Backpropagation intuition

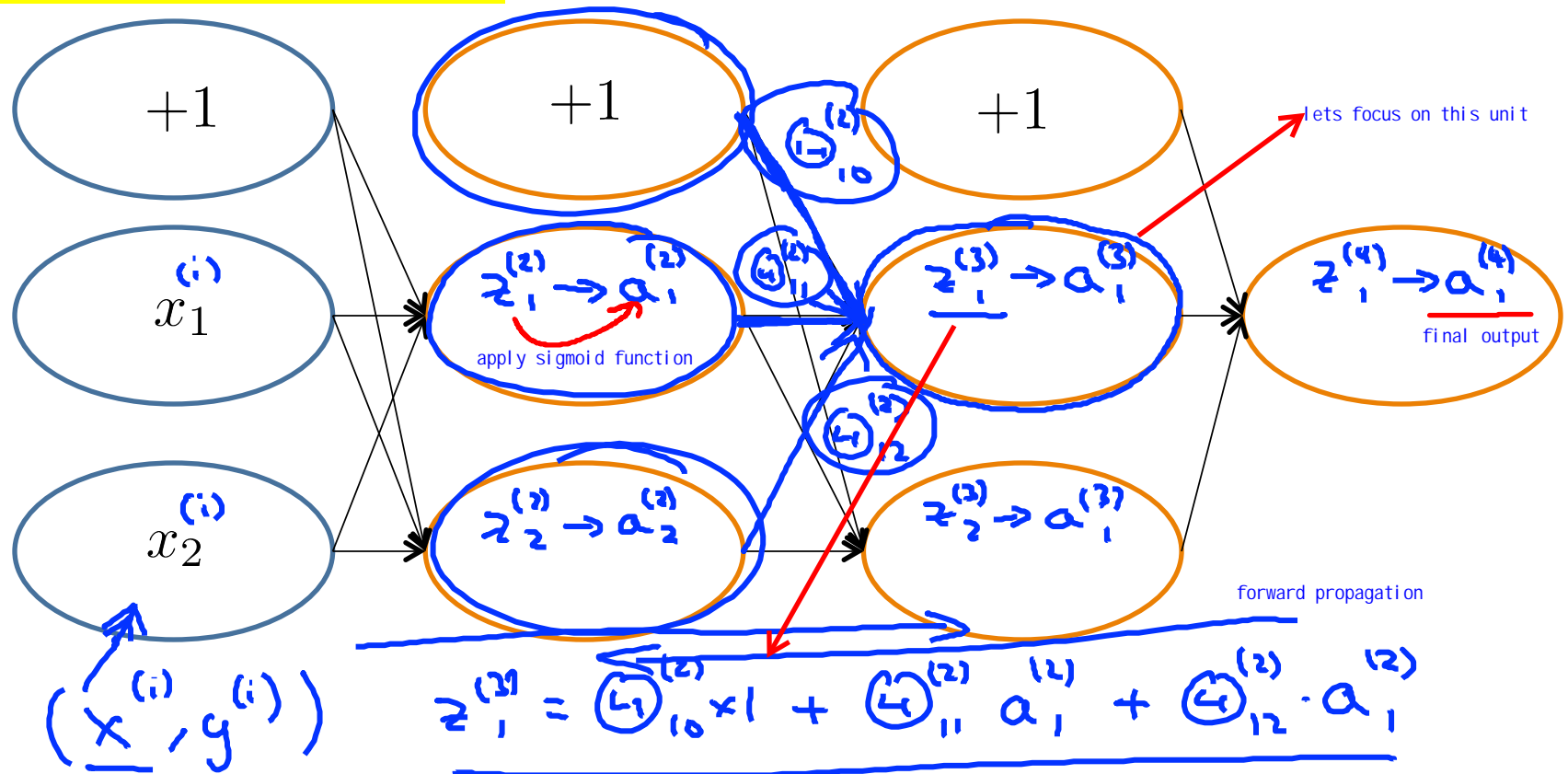
In this section, we will look at the mechanical steps of backpropagation!

Forward Propagation

not counting the bias unit



Forward Propagation



What is backpropagation doing?

to better understand what backpropagation is doing!

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

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

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

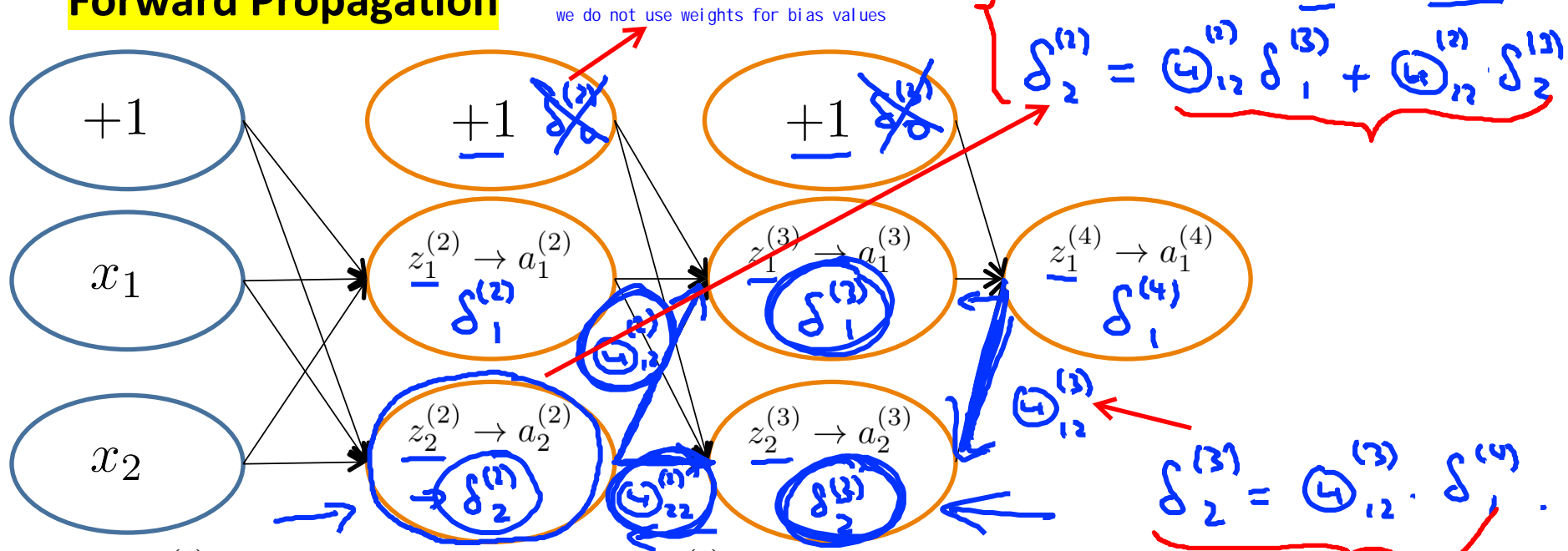
think in this way!

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

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

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

Forward Propagation



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

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(i)$ (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** 展开
parameters

from matrices to vectors, which we need in order to use the advanced optimization routines since all the advanced optimization algorithms assume we have a vector input.

Advanced optimization

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

Handwritten annotations:

- Blue arrows from gradient and theta point to \mathbb{R}^{n+1} .
- Red arrow from gradient points to "takes vector input".
- Blue arrow from theta points to \mathbb{R}^{n+1} (vectors).
- Red arrow from fminunc points to "advanced optimization algorithm".
- Blue arrow from initialTheta points to \mathbb{R}^{n+1} (vectors).

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

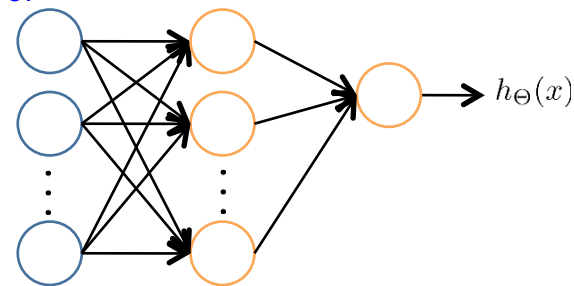
main idea here!
basically is convert matrices to vectors for computational purpose!

Example

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

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

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



$$\rightarrow \text{thetaVec} = [\text{Theta1}(:); \text{Theta2}(:); \text{Theta3}(:)];$$

$$\rightarrow \text{DVec} = [\text{D1}(:); \text{D2}(:); \text{D3}(:)];$$

reshape back to theta matrices

$$\text{Theta1} = \text{reshape}(\text{thetaVec}(1:110), 10, 11);$$

$$\rightarrow \text{Theta2} = \text{reshape}(\text{thetaVec}(111:220), 10, 11);$$


$$\rightarrow \text{Theta3} = \text{reshape}(\text{thetaVec}(221:231), 1, 11);$$

number of units in each layer

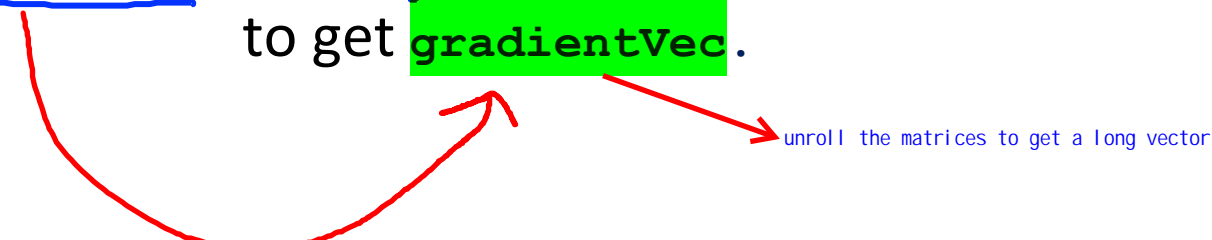
convert to a column vector

big log 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 $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$. reshape
- Use forward prop/back prop to compute $\underline{D^{(1)}, D^{(2)}, D^{(3)}}$ $\underline{J(\Theta)}$
and $\underline{D^{(1)}, D^{(2)}, D^{(3)}}$
Unroll  to get **gradientVec**.



Machine Learning

Neural Networks: Learning

Gradient checking

Numerical estimation of gradients



$$\frac{d}{d\Theta} J(\Theta) \approx$$

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

$\epsilon = 10^{-4}$

~~$$\frac{J(\Theta + \epsilon) - J(\Theta)}{\epsilon}$$~~

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

Parameter vector θ

→ $\theta \in \mathbb{R}^n$ (E.g. θ is “unrolled” version of $\underline{\Theta^{(1)}}$, $\underline{\Theta^{(2)}}$, $\underline{\Theta^{(3)}}$)

→ $\theta = [\theta_1, \theta_2, \theta_3, \dots, \theta_n]$

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

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

⋮

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

```

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;

```

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

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

Check that gradApprox \approx 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.

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 very slow.





Machine Learning

Neural Networks: Learning

Random initialization

Initial value of Θ

For gradient descent and advanced optimization method, need initial value for Θ .

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

Consider gradient descent

Set initialTheta = zeros(n,1) ?

Zero initialization



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

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

$$\frac{\partial}{\partial \Theta_{0,1}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{0,1}^{(1)}} J(\Theta)$$

$$\Theta_{0,1}^{(1)} = \Theta_{0,2}^{(1)}$$

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 10x11 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. of 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)

$y \in \{1, 2, 3, \dots, 10\}$
 ~~$y = 5$~~

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

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

→ 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 $l = 2, \dots, L$).

→ $\Delta^{(2)} := \Delta^{(2)} + \delta^{(L)} (a^{(2)})^T$

...

}

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



Training a neural network

- 5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{ik}^{(l)}} J(\Theta)$ 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

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

$J(\Theta)$ — non-convex.





Machine Learning

Neural Networks: Learning

Backpropagation
example: Autonomous
driving (optional)

