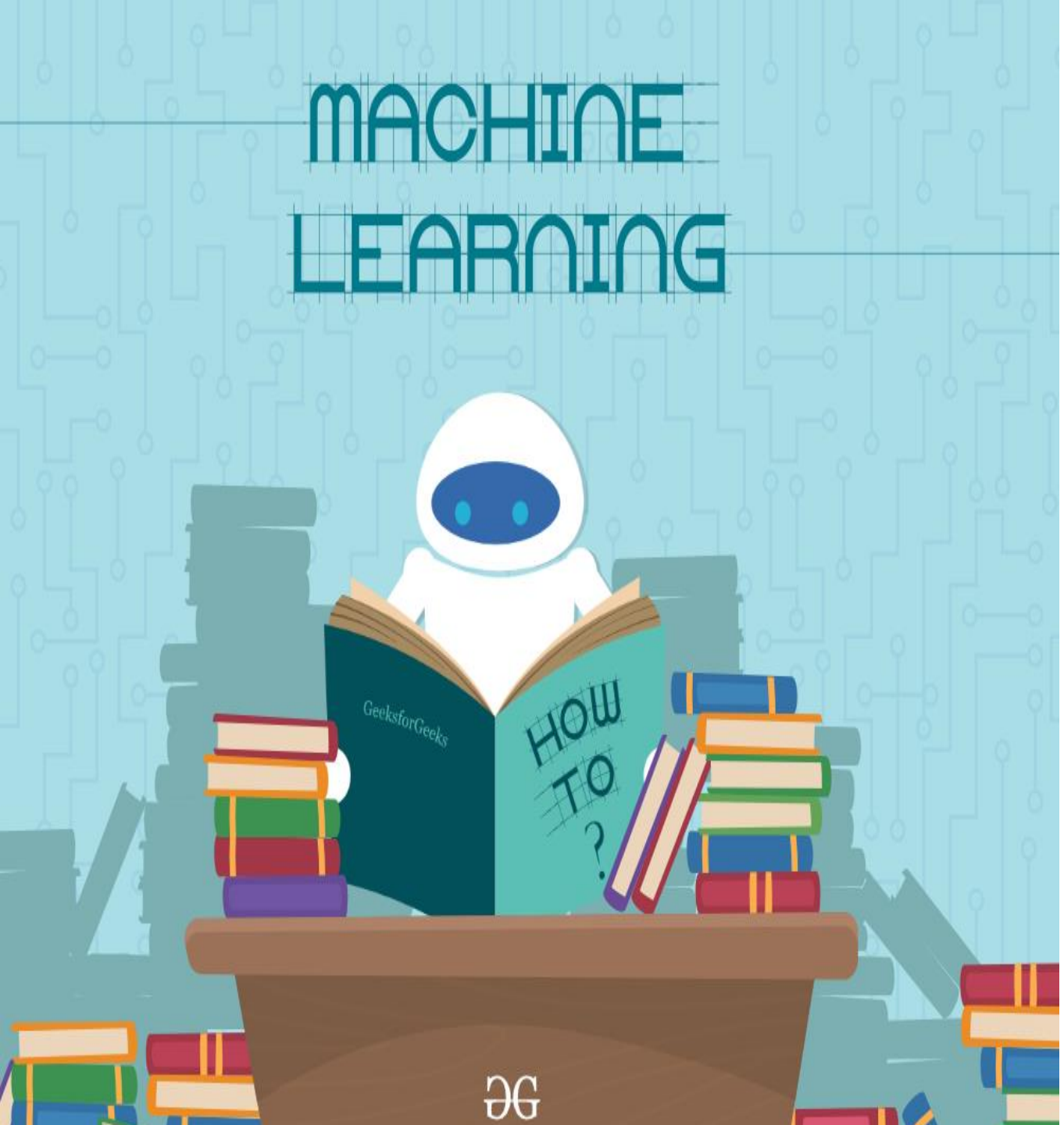


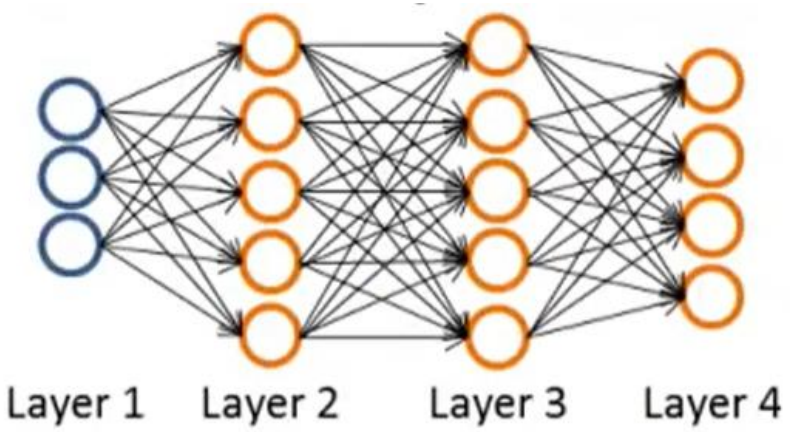
>TCC
THE CODERS' CLUB

NEURAL NETWORKS (LEARNING)

WEEK 5

Neural Networks: Classification Problem

Consider a neural network for certain classification problem.



Let the training set be represented by $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots , (x^{(m)}, y^{(m)})\}$ where,

L = total no. of layers in network

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

K = no. of output units/classes

Thus, for the given neural network, L=4
$s_1 = 3, s_2 = 5, s_3 = 5, s_4 = 4$

Note: No. of units in the output unit of a neural network is also denoted by s_L

Binary classification:

In binary classification, the neural network will have only one output unit.

- Possible values: $y = 0$ or $y = 1$
- $h_\Theta(x) \in R$
- No. of units in output unit**:** $s_L = 1$

Multi-class classification: (K classes)

In multi-class classification, the neural network will have K output units.

$y \in R^K$

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

- $h_\Theta(x) \in R^K$
- $s_L = K$ where $K \geq 3$

Neural Networks: Cost Function

The cost function for logistic regression is given by,

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

We generalize this cost function for Neural Network having K output units as follows:

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

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

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.

Suppose we want to try to minimize J(Θ) as a function of Θ, using one of the advanced optimization methods (fminunc, conjugate gradient, BFGS, L-BFGS, etc.). What do we need to supply code to compute (as a function of Θ)?

- Θ
- J(Θ)
- The (partial) derivative terms $\frac{\partial}{\partial \Theta_{ij}^{(l)}}$ for every i, j, l
- J(Θ) and the (partial) derivative terms $\frac{\partial}{\partial \Theta_{ij}^{(l)}}$ for every i, j, l

Neural Networks: Backpropagation Algorithm

Backpropagation algorithm is used to **minimize** the cost function of neural network.

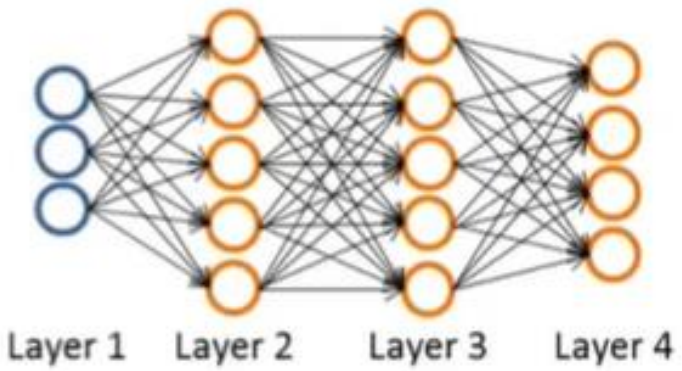
$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

Our goal is to write a code to compute:

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

Gradient Computation

For a given training example (x, y), consider the neural network as shown.



Step 1: Compute forward propagation

$a^{(1)} = x$

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

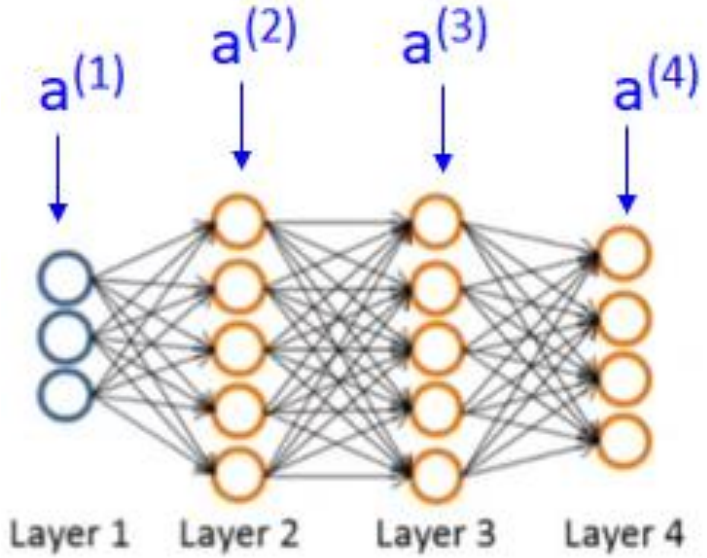
$a^{(2)} = g(z^{(2)})$ add ($a_0^{(2)}$ as the bias unit)

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

$a^{(3)} = g(z^{(3)})$ add ($a_0^{(3)}$ as the bias unit)

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

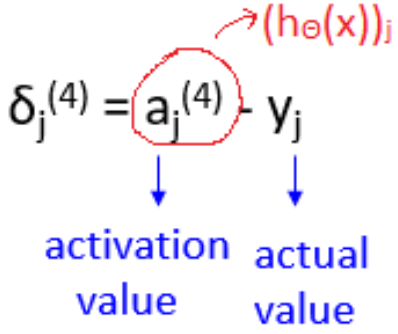
$a^{(4)} = h_\Theta(x) = g(z^{(4)})$



Step 2: In order to compute the derivatives, we make use of backpropagation algorithm.

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

For each output unit (layer L=4)



The vectorized implementation is given by,

$\delta^{(4)} = a^{(4)} - y$

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

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

Now since we have the values of δ, we use them to compute the derivatives.

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

Backpropagation algorithm

- Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$
- 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

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

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

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

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

  $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ **[vectorized implementation: $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} \cdot (a^{(l)})^T$]**

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

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

Suppose you have two training examples $(x^{(1)}, y^{(1)})$, $(x^{(2)}, y^{(2)})$. Which of the following is a correct sequence of operations for computing the gradient?
(Below, FP = forward propagation, BP = backward propagation).

- FP using $x^{(1)}$ followed by FP using $x^{(2)}$. Then BP using $y^{(1)}$ followed by BP using $y^{(2)}$.
- FP using $x^{(1)}$ followed by BP using $y^{(2)}$. Then FP using $x^{(2)}$ followed by BP using $y^{(1)}$.
- BP using $y^{(1)}$ followed by FP using $x^{(1)}$. Then BP using $y^{(2)}$ followed by FP using $x^{(2)}$.
- FP using $x^{(1)}$ followed by BP using $y^{(1)}$. Then FP using $x^{(2)}$ followed by BP using $y^{(2)}$.

Backpropagation Algorithm: Intuition

The cost function for a neural network is given by,

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

If we consider simple non-multiclass classification (k = 1) and ignore regularization, the cost is computed with:

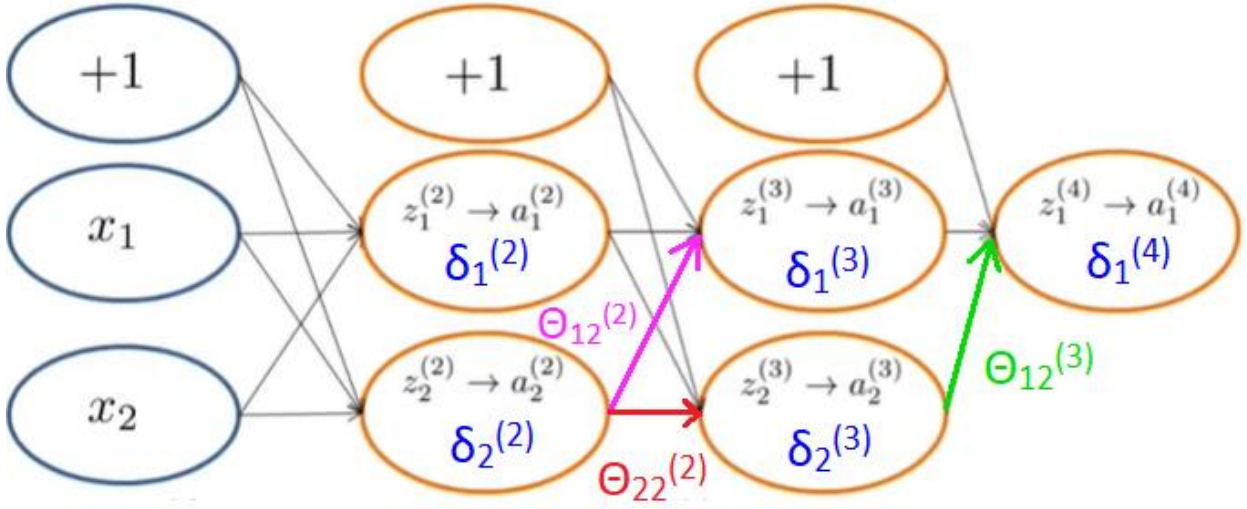
$$\text{cost}(t) = y^{(t)} \log(h_\Theta(x^{(t)})) + (1 - y^{(t)}) \log(1 - h_\Theta(x^{(t)}))$$

Intuitively, $\delta_j^{(l)}$ is the "error" for $a_j^{(l)}$ (unit j in layer l). More formally, the delta values are actually the derivate of the cost function.

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

We know that the derivative is the slope of a line tangent to the cost function, so the steeper the slope, the more incorrect we are.

Forward propagation:



In the above neural network,

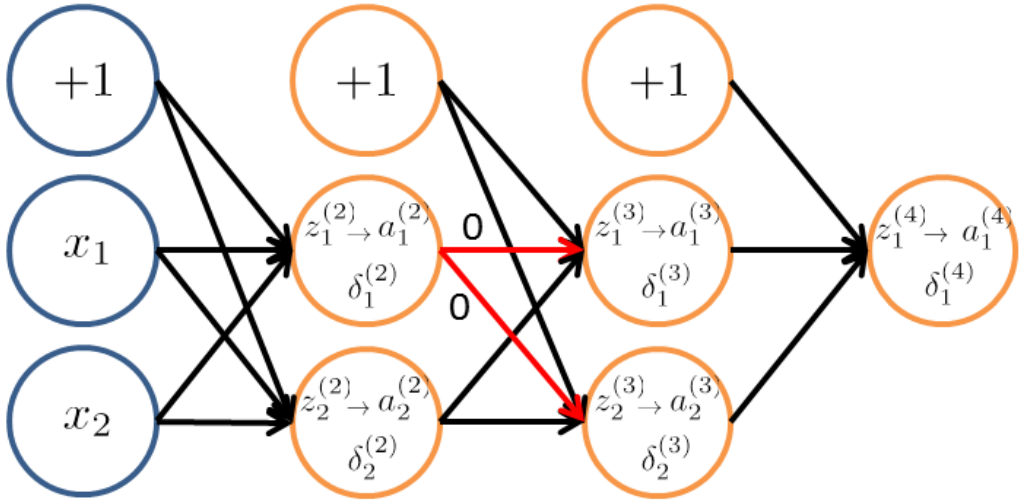
$$\delta_2^{(2)} = \Theta_{12}^{(2)} * \delta_1^{(3)} + \Theta_{22}^{(2)} * \delta_2^{(3)}$$

To calculate every single possible $\delta_j^{(l)}$, we could start from the right of our diagram.
We can think of our edges as our Θ_{ij}. Going from left to right, to calculate the value of $\delta_j^{(l)}$, you can just take the over all sum of each weight times the δ it is coming from.

Hence, another example would be

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

Consider the following neural network:



Suppose both of the weights shown in red ($\Theta_{11}^{(2)}$ and $\Theta_{21}^{(2)}$) are equal to 0. After running backpropagation, what can we say about the value of $\delta_1^{(3)}$?

- $\delta_1^{(3)} > 0$
- $\delta_1^{(3)} = 0$ only if $\delta_1^{(2)} = \delta_2^{(2)} = 0$, but not necessarily otherwise
- $\delta_1^{(3)} \leq 0$ regardless of the values of $\delta_1^{(2)}$ and $\delta_2^{(2)}$
- There is insufficient information to tell

Implementation Note: Unrolling Parameters

Advanced Optimization:

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

Consider a neural network with L=4.

Let $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$ (Theta1, Theta2, Theta3) and $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ (D1, D2, D3) be the matrices that we shall implement in program.

In order to use optimizing functions such as "fminunc()", we will want to "unroll" all the elements and put them into one long vector.

In other words, we convert the matrices into one long vector.

```
thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]  % vector of matrices
deltaVector = [ D1(:); D2(:); D3(:); ]
```

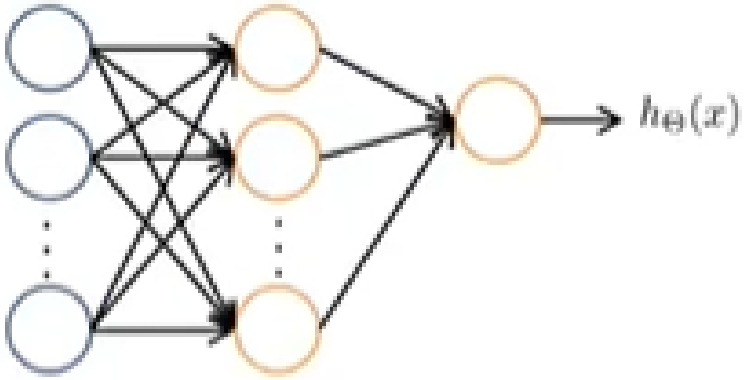
Example:

Consider a neural network:

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

$\Theta^{(1)} \in R^{10x11}$, $\Theta^{(2)} \in R^{10x11}$, $\Theta^{(3)} \in R^{1x11}$

$D^{(1)} \in R^{10x11}$, $D^{(2)} \in R^{10x11}$, $D^{(3)} \in R^{1x11}$



Unrolling the matrices Theta1, Theta2, Theta3 and D1, D2, D3 into one single vector thetaVec and DVec respectively.

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

To get back the original matrices, we use "reshape" function

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

Suppose D1 is a 10x6 matrix and D2 is a 1x11 matrix. You set:

DVec = [D1(:); D2(:)];

Which of the following would get D2 back from DVec?

- reshape(DVec(60:71), 1, 11)
- reshape(DVec(61:72), 1, 11)
- reshape(DVec(61:71), 1, 11)
- reshape(DVec(60:70), 11, 1)

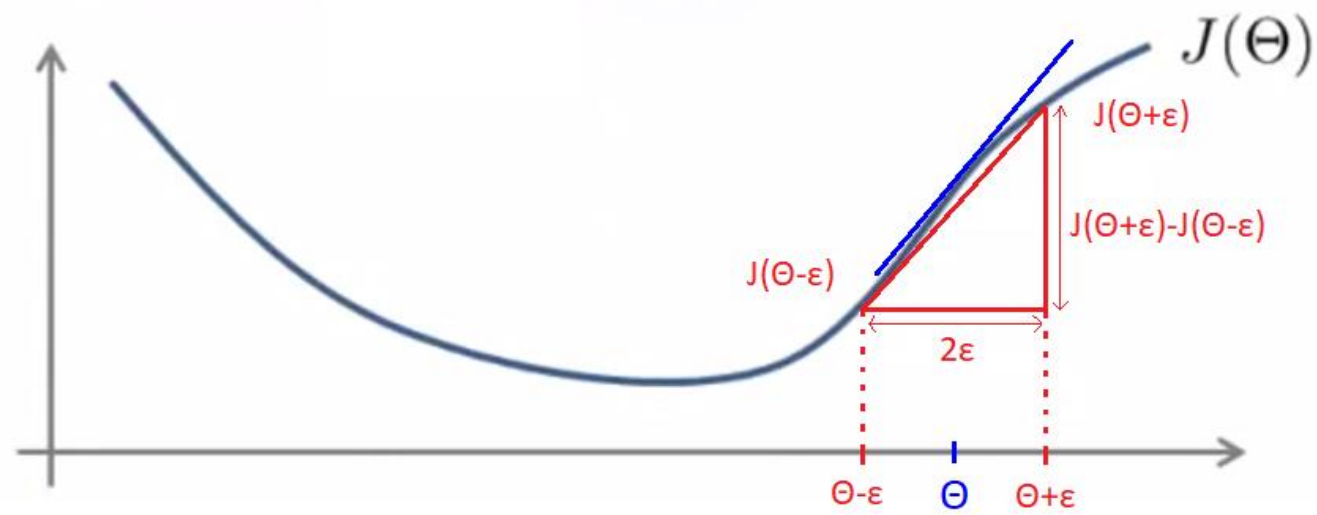
Summary:

The steps for the learning algorithm are:

- 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 $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$.
- Use forward propagation/backward propagation to compute $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ and $J(\Theta)$.
- Unroll $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ to get gradientVec.

Neural Networks: Gradient Checking

To make sure that the backpropagation algorithm is working well as intended, without the possibility of any errors, we use the concept called "Gradient Checking".



We can approximate the derivative of our cost function with $\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta+\varepsilon)-J(\Theta-\varepsilon)}{2\varepsilon}$

Implementation:

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

Let $J(\theta) = \theta^3$. Furthermore, let $\theta = 1$ and $\varepsilon = 0.01$. You use the formula:

$\frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}$ to approximate the derivative.

What value do you get using this approximation? (When $\theta = 1$, the true, exact derivation is $\frac{d}{d\theta} J(\theta) = 3$).

- 3.0000
- 3.0001
- 3.0301
- 6.0002

Parameter vector θ

$\theta \in R^n$ (E.g.: θ is "unrolled" version of $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$)

$\theta = \theta_1 , \theta_2 , \theta_3 , \dots , \theta_n$

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

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

.

.

.

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

```
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;
```

Compare the value of gradApprox with that of DVec obtained from backpropagation.

If the backpropagation algorithm is correct, it is found that:

gradApprox ≈ DVec

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 i.e. gradApprox ≈ DVec
- Once you have verified once that your backpropagation algorithm is correct, turn off gradient checking. You don't need to compute gradApprox again.

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.

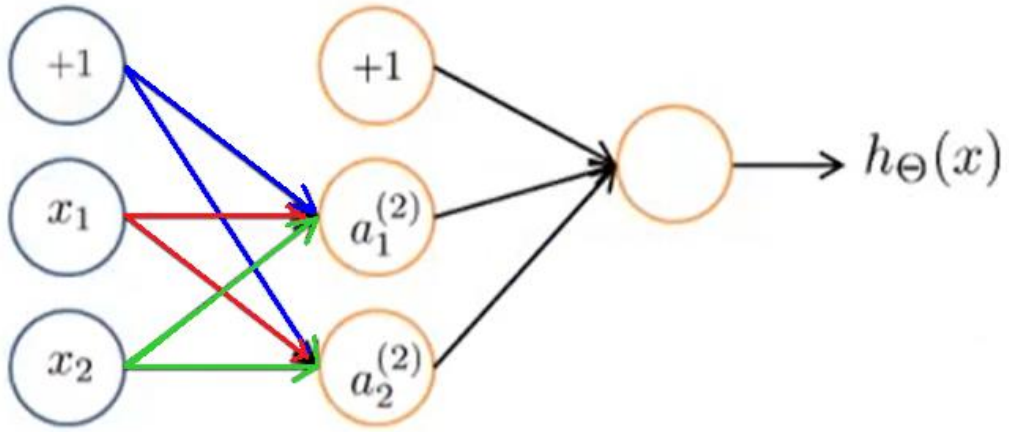
What is the main reason that we use the backpropagation algorithm rather than the numerical gradient computation method during learning?

- The numerical gradient computation method is much harder to implement.
- The numerical gradient algorithm is very slow.
- Backpropagation does not require setting the parameter EPSILON.
- None of the above.

Neural Networks: Random Initialization

Zero initialization:

Consider a neural network as shown:



If $\Theta_{ij}^{(l)} = 0$ for all i, j, l then:

- $a_1^{(2)} = a_2^{(2)}$
- $\delta_1^{(2)} = \delta_2^{(2)}$
- Eventually, $\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$
- Eventually, $\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$

- Initializing all theta weights to zero does not work with neural networks.
- When we backpropagate, all nodes will update to the same value repeatedly.
- Hence, after each update, parameters corresponding to inputs going into each of two hidden inputs are identical.
- To overcome this problem, we use the method of "Random Initialization".

Random initialization:

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

Example:

```
Theta1 = rand(10, 11) * (2 * INIT_EPSILON) – INIT_EPSILON;
Theta2 = rand(1, 11) * (2 * INIT_EPSILON) – INIT_EPSILON;
```

Note: The epsilon used above is unrelated to the epsilon from Gradient Checking

Consider this procedure for initializing the parameters of a neural network:

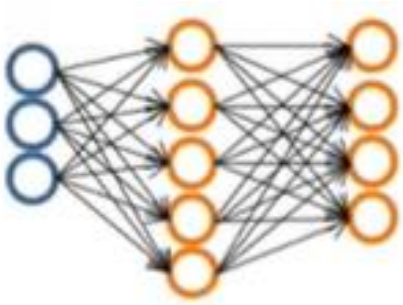
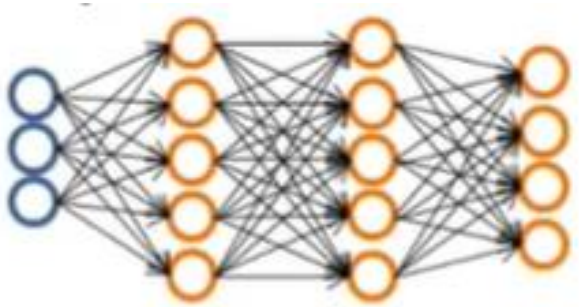
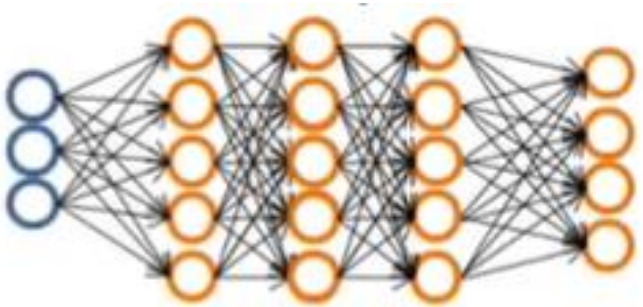
1. Pick a random number r = rand(1,1) * (2 * INIT_EPSILON) - INIT_EPSILON;
2. Set $\Theta_{i\,j}^{(l)} = r$ for all i, j, l

Does this work?

- Yes, because the parameters are chosen randomly.
- Yes, unless we are unlucky and get r=0 (up to numerical precision).
- Maybe, depending upon the training set inputs x(i).
- No, because this fails to break symmetry.

Neural Networks: Putting it Together

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers in total you want to have.

- No. of input units = Dimension of features $x^{(i)}$
- No. of output units = No. of classes
- No. of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
- By default, there is 1 hidden layer for any neural network.
  If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

Training a Neural Network:

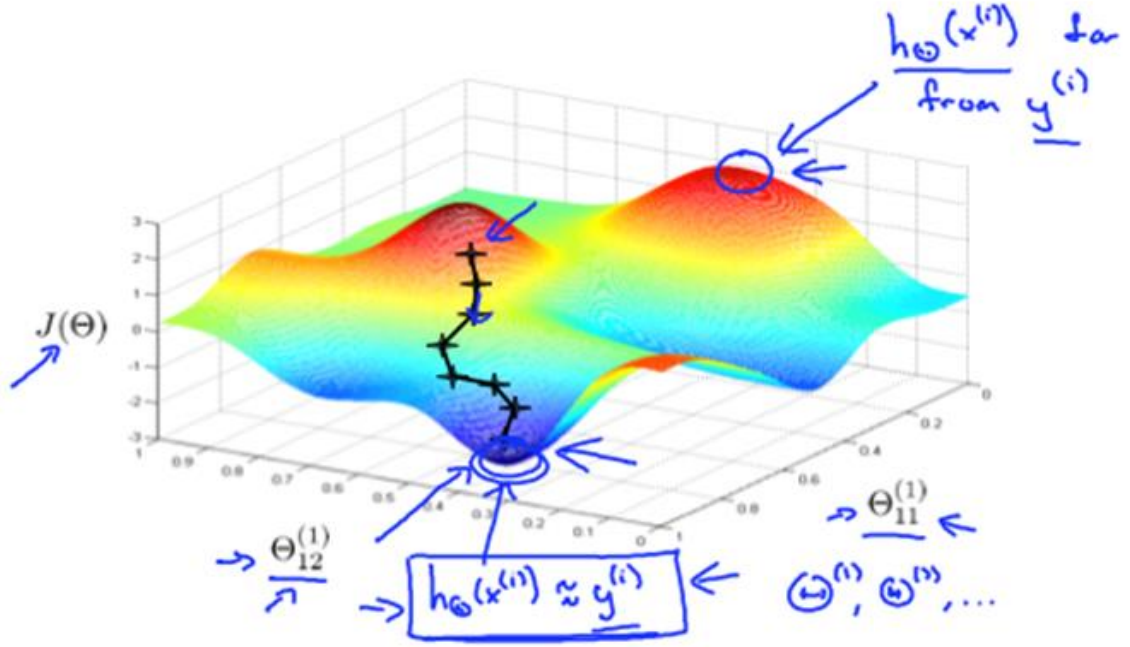
1. Randomly initialize the weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. 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(Θ) is not convex and thus we can end up in a local minimum instead.

Suppose you are using gradient descent together with backpropagation to try to minimize J(Θ) as a function of Θ. Which of the following would be a useful step for verifying that the learning algorithm is running correctly?

- Plot J(Θ) as a function of Θ, to make sure gradient descent is going downhill.
- J(Θ) as a function of the number of iterations and make sure it is increasing (or at least non-decreasing with every iteration).
- J(Θ) as a function of the number of iterations and make sure it is decreasing (or at least non-increasing with every iteration).
- J(Θ) as a function of the number of iterations and make sure the parameter values are improving in classification accuracy.

Neural Networks: Autonomous Driving Example

- ALVINN and Tesla are examples of self-driving cars that operate using a Neural Network.

ALVINN (Autonomous Land Vehicle In a Neural Network)

Tesla Autopilot

Additional Sources:
https://www.youtube.com/watch?v=WPexu1mUH5s
https://www.youtube.com/watch?v=H0igiP6Hg1k
https://www.youtube.com/watch?v=tlThdr3O5Qo