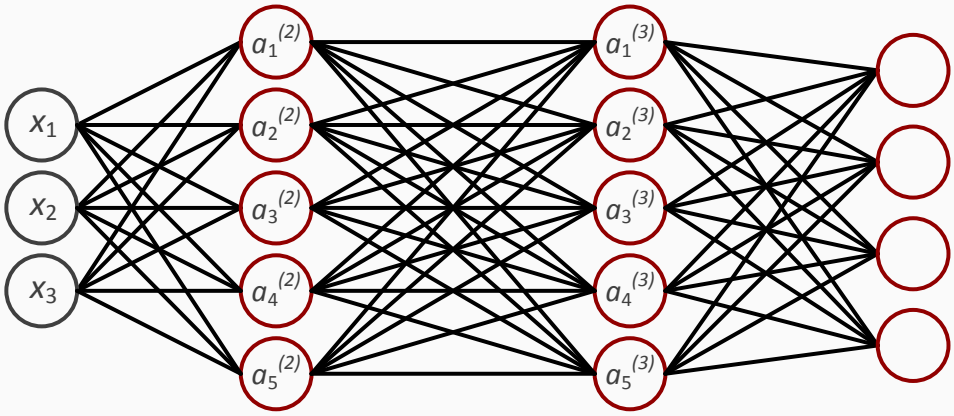
artificial neural networks ⇄ learning

cost function and backpropagation

cost function

neural network (classification)



L = total number of layers in network

s_l = number of units (excluding bias unit) in layer l

$L = 4$ $s_1 = 3, s_2 = 5, s_3 = 4, s_4 = s_L = 4$

binary classification	multiclass classification (K classes)
y = 0 or 1	y = a vector of binary classification
1 output unit	K output units
$h_\theta(x) \in \mathbb{R}$	$h_\theta(x) \in \mathbb{R}^K$
$s_L = 1$ $K = 1$	$s_L = K$ K is typically ≥ 3

cost function

logistic regression

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

For logistic regression, the goal is to minimize the above cost function. For a neural network, there will be not just a single logistic regression output unit, there will be K.

neural network

$h_\theta(x) \in \mathbb{R}^K$ $\left(h_\theta(x)\right)_i = i^{\text{th}}$ output

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{i=1}^{K} y_k^{(i)} \log\left(h_\theta(x^{(i)})\right)_k + \left(1 - y_k^{(i)}\right) \log\left(1 - \left(h_\theta(x^{(i)})\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_1}\sum_{j=1}^{s_1+1}\left(\theta_{ji}^{(l)}\right)^2$$

example

to minimize $J(\theta)$ as a function of θ, when programming code into an algorithm, the necessary code to supply is for $J(\theta)$ and the (partial) derivative terms $\frac{\partial}{\partial \theta_{ij}^{(l)}}$ for every *i, j, l*.

backpropagation algorithm

gradient computation

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{i=1}^{K} y_k^{(i)} \log\left(h_\theta(x^{(i)})\right)_k + \left(1 - y_k^{(i)}\right) \log\left(1 - \left(h_\theta(x^{(i)})\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_1}\sum_{j=1}^{s_1+1}\left(\theta_{ji}^{(l)}\right)^2$$

with the goal to $\min_\theta J(\theta)$, it is necessary to compute $-J(\theta)$ and $-\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta)$

given **one** training example (x, y):

first apply forward propagation in order to compute the hypothesis outputs

$a^{(1)} = x$ *input layer*

$z^{(2)} = \theta^{(1)} a^{(1)}$ *1st hidden layer*

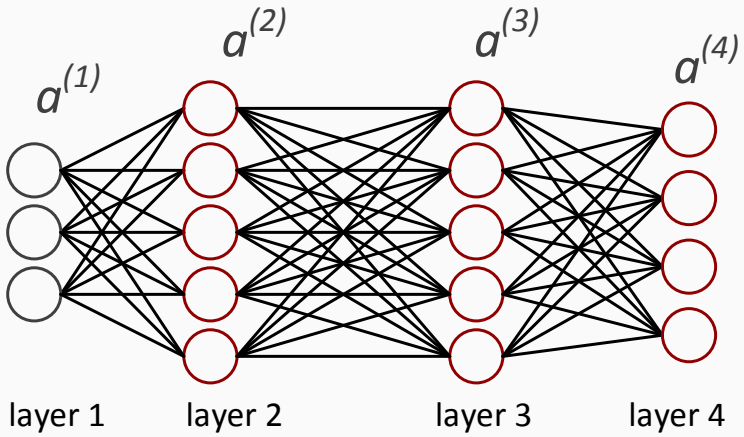
$a^{(2)} = g(z^{(2)})$ add $a_0^{(2)} = 1$

$z^{(3)} = \theta^{(2)} a^{(2)}$ *2nd hidden layer*

$a^{(3)} = g(z^{(3)})$ add $a_0^{(2)} = 1$

$z^{(4)} = \theta^{(3)} a^{(3)}$ *output layer*

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



this is a vectorized implementation of forward propagation allowing to compute the activation values for all neurons in the neural network.

next apply backward propagation in order to compute the partial derivatives

intuition: $\delta_i^{(l)}$ = "error" of node *j* in layer *l*.

recall $a_i^{(l)}$ that does the activation of *j* unit in layer *l*; thus, the $\delta_i^{(l)}$ is going to capture the error in that activation of neural duo.

for each output unit (layer L = 4)

$\delta_i^{(4)} = a_j^{(4)} - y_j$ → vectorized implementation $\delta^{(4)} = a^{(4)} - y$

The error ($\delta_i^{(4)}$) is = to the activation of that unit ($a_j^{(4)} = (h_\theta(x))_j$) minus the actual value of 0 in our training example (y_j)

next compute the delta terms for prior network layers.

each factor in the below function ($(\theta^{(3)})^{T}\delta^{(4)}$ and $g'(z^{(3)})$) are vectors; the .* operator indicates element wise multiplication of the vectors.

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

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

(no $\delta^{(1)}$ term), the first layer corresponds to the input layer; being the input features (do not want to change those values); the computation starts at the output later and solves backward

backpropagation algorithm in the context of a larger sample

bring together the theory above to compute derivatives with respect to parameters

in the training set: {($x^{(1)}$, $y^{(1)}$),..., ($x^{(m)}$, $y^{(m)}$)}

set $\Delta_{ji}^{(l)} = 0$ (for all l, i, j) (used to compute the partial derivatives, slowly adding together in the process)

next will loop through the training set ($x^{(i)}$, $y^{(i)}$)

for $i = 1\ to\ m$

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

perform forward propagation to compute $a^{(1)}$ for $l = 2, 3, \ldots, L$

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

compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ as before, no $\delta^{(1)}$ because error is not associated with input layer

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

then outside the for loop

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

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

$$\frac{\partial}{\partial\theta_{ij}^{(l)}}J(\theta) = D_{ij}^{(l)}$$

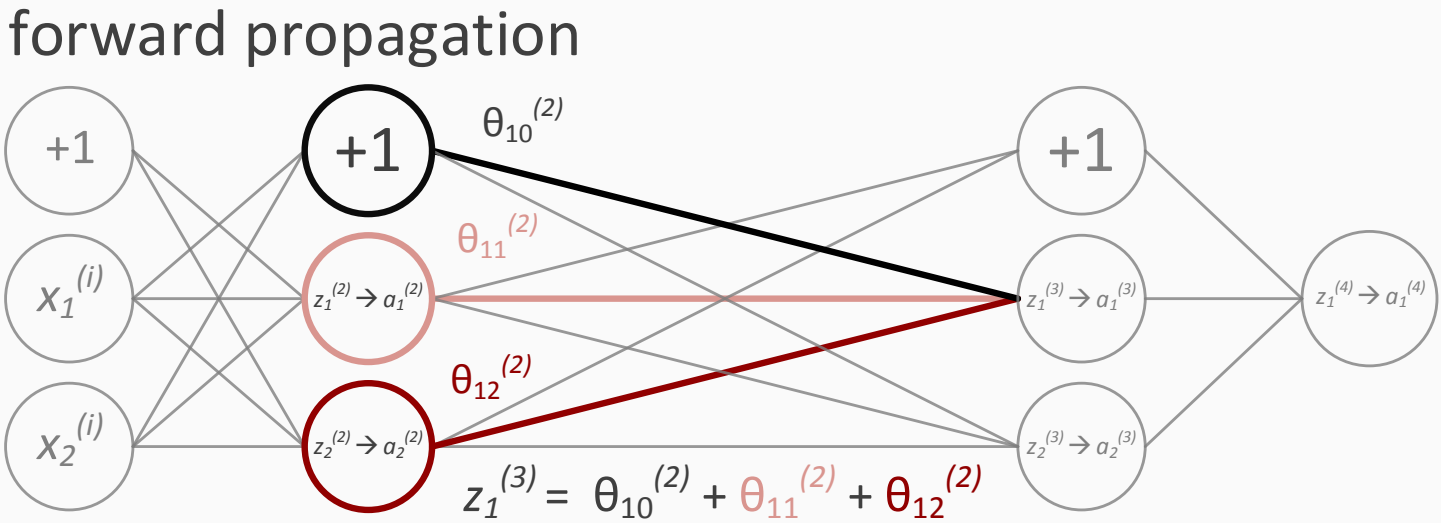
example

with two training examples ($x^{(1)}$, $y^{(1)}$) and ($x^{(2)}$, $y^{(2)}$), the correct sequence of operations for computing the gradient are as follows (FP = forward propagation and BP = backward propagation)

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

backpropagation intuition

stepping through the mechanics in a systematic method: using backpropagation to compute the derivatives of a function



in performing forward propagation, the i inputs will be inserted into the first layer. Then the weighted sum of inputs (z) of the input units are forward propagated to the second layer. The sigmoid logistic function (a) will be applied to the z values, giving the activation values. The step will be repeated in an additional forward propagation to the next layer. The final output value will be given in the output layer.

what is back propagation doing?

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{i=1}^{K} y_k^{(i)} \log\left(h_\theta\left(x^{(i)}\right)\right)_k + \left(1 - y_k^{(i)}\right)\log\left(1 - \left(h_\theta\left(x^{(i)}\right)\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_1}\sum_{j=1}^{s_1+1}\left(\theta_{ji}^{(l)}\right)^2$$

focusing on a single example , the case of 1 output unit, and ignoring regularization:

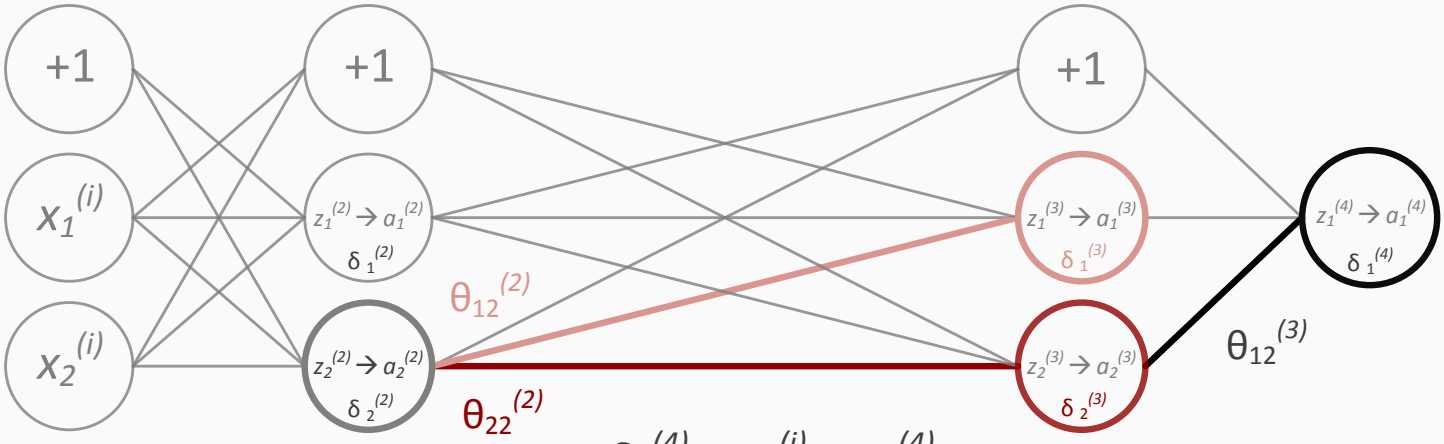
$$\text{cost}(i) \approx\ y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right)\log h_\theta\left(x^{(i)}\right)$$

think of the cost as the sum of least squared errors: $\text{cost}(i) \approx\ \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$

the cost(i) is measuring how well the network is performing on example i

backward propagation

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



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

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

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

backpropagation in practice

implementation note: unrolling parameters

the process of unrolling parameters form matrices into vectors in order to use advanced optimization:

function taken to calculate cost function (`jVal`) and derivatives (`gradient`). then pass to advanced optimization algorithm (`fminunc`), taking the input, pointing to the cost function and an initial value of theta:

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

the gradient, theta, and initialTheta terms are vectors $\mathbb{R}^{n+1}$ which work in terms of logistic regression.

in neural networks, the parameters are matrices:

Neural Network (L=4):

$\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$ – matrices (`Theta1, Theta2, Theta3`)

$D^{(1)}, D^{(2)}, D^{(3)}$ – matrices (`D1, D2, D3`)

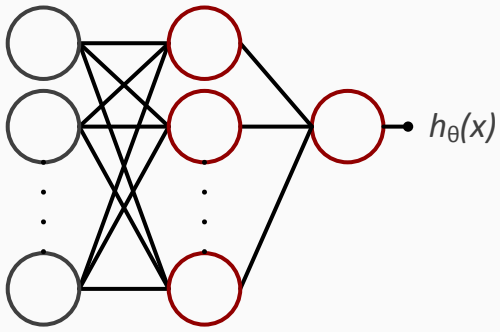
"unroll" into vectors

example

$s^{(1)} = 10, s^{(2)} = 10, s^{(3)} = 1$

$\theta^{(1)} \epsilon\ \mathbb{R}^{10\times11}, \theta^{(2)} \epsilon\ \mathbb{R}^{10\times11}, \theta^{(3)} \epsilon\ \mathbb{R}^{1\times11}$

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



code to unroll the matrices into a vector:

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

code to roll the vector back into the original matrices:

```
Theta1 = reshape(thetaVec(1:110), 10, 11) ;
Theta2 = reshape(thetaVec(111:120), 10, 11) ;
Theta3 = reshape(thetaVec(221:231), 1, 11) ;
```

utilizing unrolling to implement the learning algorithm

have initial parameters $\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$

unroll matrices into a long vector to get `initialTheta` to pass to `fminunc (@costFunction, initialTheta, options)`

then implement the cost function:

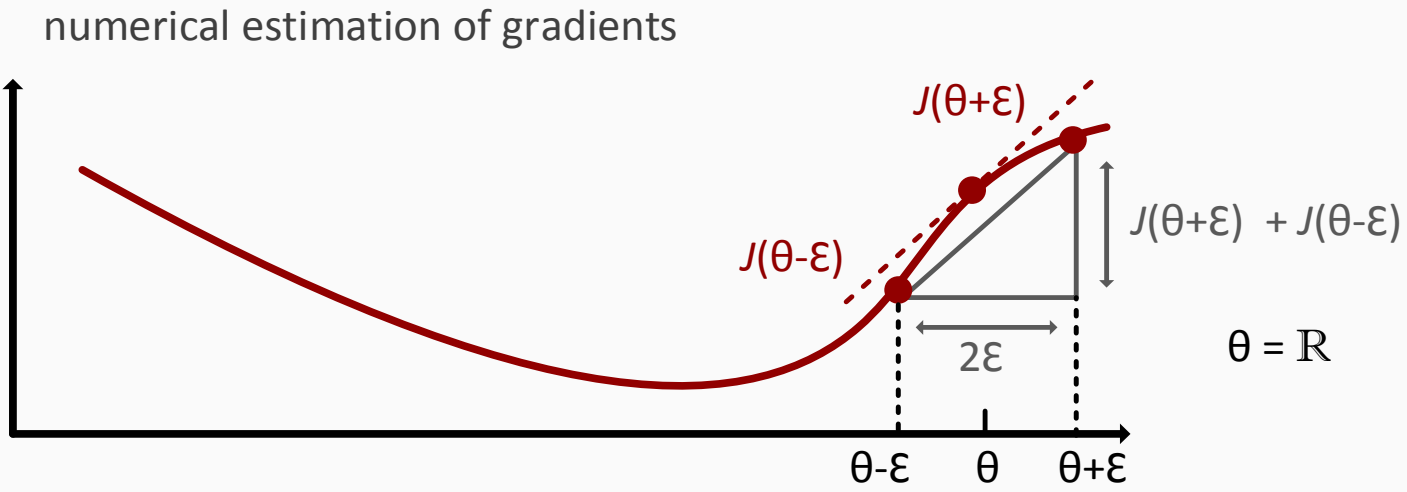
```
function [ jVal, gradient ] = costFunction (thetaVec)
```

from `thetaVec`, get $\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$

use forward/backward propagation to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\theta)$ unroll to get `gradientVec`

gradient checking

backpropagation is prone to errors in practice which can appear to decrease for every iteration in gradient descent. gradient checking is a way to ensure there are no unforeseen bugs in implementation and the function is correctly computing the derivatives of cost function $J(\theta)$



$\frac{\partial}{\partial\theta}J(\theta) \approx \frac{J(\theta+\varepsilon)-(\theta-\varepsilon)}{2\varepsilon}$ typically Ɛ will be labeled as a small value ($Ɛ=10^{-4}$), while avoiding an overly small value, because the right hand term mathematically becomes to the slope of the function at the given point.

implementation in octave:

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

example

$J(\theta) = \theta^3, \theta = 1, \varepsilon = 0.01$

using the formula $\frac{J(\theta+\varepsilon)-(\theta-\varepsilon)}{2\varepsilon}$ the derivative approximates a value of 3.0001

(when $\theta = 1$, the true derivative is $\frac{\partial}{\partial\theta}J(\theta) = 3$)

a more general case when θ ≠ $\mathbb{R}$

parameter vector θ

$\theta \in \mathbb{R}^n$ (e.g. θ is unrolled version of $\theta^{(1)}$, $\theta^{(2)}$, $\theta^{(3)}$)

$\theta = [\theta_1, \theta_2, \theta_3, \ldots, \theta_n]$

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

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

$$\vdots$$

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

The above functions provide a numerical computation of the partial derivative of J with respect to any one of the parameters θ_i

implementation in octave:

```
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;
```

check that `gradApprox` ≈ `DVec` (`DVec` is derivative from backpropagation)

the above computes the derivative of the cost function with respect to every parameter in the neural network, then taking the gradient computed from backpropagation

implementation note:

implement backpropagation to compute `DVec` (unrolled $D^{(1)}$, $D^{(2)}$, $D^{(3)}$)
implement numerical gradient check to compute `gradApprox`
ensure the above results in similar values
turn off gradient checking. using backpropagation code for learning

important:

be sure to disable gradient checking code before training the classifier. If numerical gradient computation is run on every iteration of gradient descent (or in the inner loop of `costFunction(...)`) the code will be sufficiently slow.

The main reason for using backpropagation algorithm rather than numerical gradient descent computation method during **learning**:

The numerical gradient descent algorithm is very slow to execute

random initialization

initial value of θ

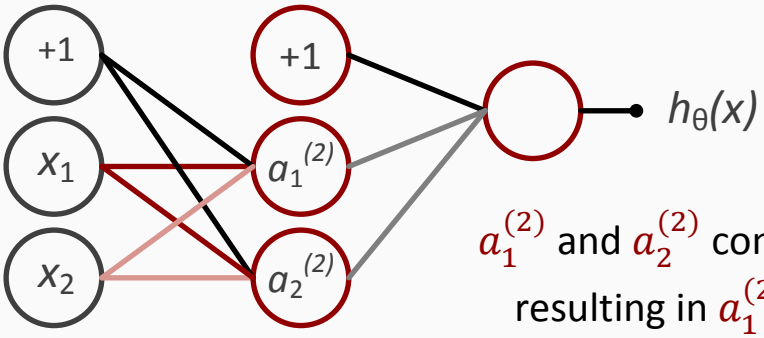
for gradient descent and advanced optimization method, initial θ value is needed

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

consider gradient descent

set `initialTheta = zeros(n,1)` → setting θ to 0 initially works in logistic regression but does not work in neural networks

zero initialization



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

all weights on initialization will be the same and result in both hidden units $a_1^{(2)}$ and $a_2^{(2)}$ computing the same function of the inputs, resulting in $a_1^{(2)} = a_2^{(2)}$. after each update, parameters corresponding to the inputs going into each of the two hidden units are identical

random initialization: symmetry breaking

initialize each $\theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$

(i.e. $-\epsilon \leq \theta_{ij}^{(l)} \leq \epsilon$)

implementation in octave:

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

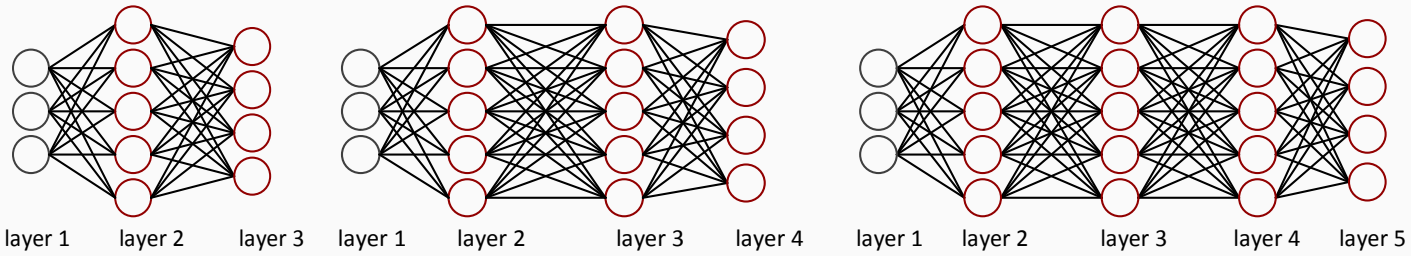
putting it together

fitting the constructs of neural networks together and developing the process of implementing a neural network learning algorithm

training a neural network

network architecture

determining a network architecture depends on the number of input units (dimensions of features $x^{(i)}$ and the number of output units (number of classes; single or multiclass classification)



in multiclass classification where the output units $y \in \{1, 2, 3, \ldots, 10\}$, the network will not be shown the number of outputs as a notation of y for each possible classes (i.e. y = 5) but as vectors:

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \; or \; y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$ etc.

common practice is typical of a single hidden layer. if hidden layers >1, it is best practice to have the same number of hidden units for each hidden layer (i.e. if a^1 has 5 units, then a^2 has 5 units). lastly having a number of features in the hidden layers is typically comparable to the number of input units.

implementation process to train a neural network

set up inputs and randomly initialize values of the weights (typically values near zero)

implement forward propagation to compute $h_\theta(x)$, the output vector of the *y* values

compute the cost function of $J(\theta)$

implement backpropagation to compute partial derivative terms $\frac{\partial}{\partial \theta_1} J(\theta)$ with respect to the parameters. usually achieved with a for loop over the training examples:

for `i = 1:m`

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

the loop iterated FP and BP over the first example $(x^{(1)}, y^{(1)})$, second example $(x^{(2)}, y^{(2)})$, and finally the last example , … , $(x^{(m)}, y^{(m)})$

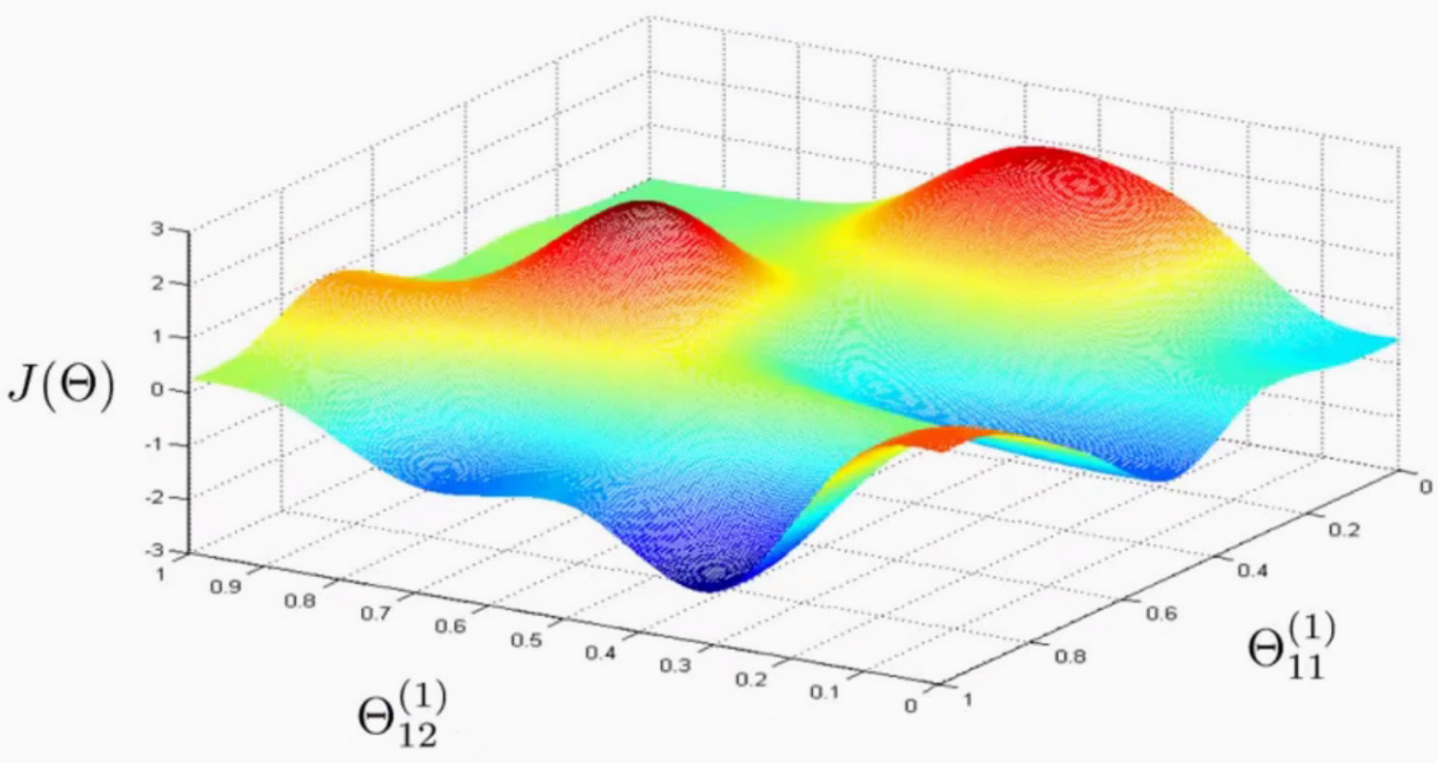
(get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \dots, L$)

use gradient checking to compare $\frac{\partial}{\partial \theta_1} J(\theta)$, computed using backpropagation versus using numerical estimate gradient of $J(\theta)$ to ensure similar values

subsequently disable the gradient checking code

use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\theta)$ as a function of parameters θ

in neural networks, cost function $J(\theta)$ is non-convex and susceptible to local optima



example

in order to verify a learning algorithm is running correctly when using gradient descent with backpropagation to minimize $J(\theta)$ as a function of θ, plot $J(\theta)$ as a function of the number of iterations and ensure it is decreasing (or at least not increasing) with every iteration

1. while training a three layer neural network to use backpropagation to compute the gradient of the cost function. one of the steps is to update
$\Delta_{ij}^{(2)} := \Delta_{ij}^{(2)} + \delta_i^{(3)} \times \left(a^{(2)}\right)_j$ in the backpropagation algorithm

    for every i, j, $\Delta^{(2)} := \Delta^{(2)} + \delta^{(3)} \times \left(a^{(2)}\right)^T$ is a vectorization of this step

2. `Theta1` is a 5x3 matrix, and `Theta2` is a 4x6 matrix.

`thetaVec` is set to `thetaVec=[Theta1(:);Theta2(:)]`.

the following correctly recovers `Theta2` → `reshape(thetaVec(16:39),4,6)`

3. let $J(\theta)=2\theta^4+2$. Let $\theta=1$, and $\epsilon=0.01$. Use the formula $J(\theta+\epsilon)-J(\theta-\epsilon)$ / 2ϵ to numerically compute an approximation to the derivative at $\theta=1$.

(when θ=1, the true/exact derivative is dJ(θ)dθ=8.) the computed value is 8.0008

4. the following statements are true to neural networks

    using gradient checking can help verify if one's implementation of backpropagation is bug-free.

    if our neural network overfits the training set, one reasonable step to take is to increase the regularization parameter λ.

5. the following statements are true to neural networks

    if we are training a neural network using gradient descent, one reasonable "debugging" step to make sure it is working is to plot j(θ) as a function of the number of iterations, and make sure it is decreasing (or at least non-increasing) after each iteration.

    suppose you are training a neural network using gradient descent. depending on your random initialization, your algorithm may converge to different local optima (i.e., if you run the algorithm twice with different random initializations, gradient descent may converge to two different solutions).