# 1 Neural Networks: Learning

## 1.1 I. Cost Function

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

$L =$total number of layers in network

$s_l =$number of units (not counting bias unit) in layer $l$

$K =$number of units in the output layer

- Binary classification:
  - $y = 0\, or\, 1$ labels
  - 1 output unit $h_\Theta(x) \in \mathbb{R}$
  - $S_L = 1$
  - $K = 1$
- Multi-class classification ($K$ classes)
  - $y \in \mathbb{R}^K$
    e.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$
  - $K$ output units
    * $h_\Theta(x) \in \mathbb{R}^k$
    * $s_2 = K, \quad (k \geq 3)$

The cost function for the neural network is going to be a generalization of the one we used for logistic regression, where instead of having one logistic regression output unit, we'll have K of them.

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

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

Inner sum is a sum over $k$ output units.

Regularization term sums over $\Theta_{ji}^l$, but we are not summing over the terms corresponding to where $i = 0$, like $\Theta_{i0} \times x_0$. In other words, we don't regularize the bias term.

l - layer

To minimize $J(\Theta)$ as a function of $\Theta$, using one of the advanced optimization methods (fminunc, conjugate gradient, BFGS, L-BFGS, etc.), we need to compute

$$J(\Theta),\ \tfrac{\partial}{\partial \Theta_{ij}^l}, \quad \forall i, j, l$$

### 1.1.1 Formulas:

Neural network now outputs vectors

$K$ $\longrightarrow$number of output elements

$h_\Theta(x) \in \mathbb{R}^K$$\rightarrow$K dimensional vector

$(h_\Theta(x))_i = i^{th} output$ $\longrightarrow$i selects ith element

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

$J(\Theta)$ $\longrightarrow$cost function

## 1.2 Backpropagation Algorithm

Algorithm to minimize the cost function.

In order to use gradient descent or one of the advanced optimization algorithms, we need to write code that takes $\Theta$ as a parameter and computes $J(\Theta)$ and the partial derivative terms $\frac{\partial}{\partial \Theta_{ij}^l} J(\Theta)$, $\Theta_{ij}^{(l)} \in \mathbb{R}$.

- The first thing we do is apply forward propagation to compute what the hypothesis actually outputs. Say you have just one training example $(x,\, y)$.
  - $a^{(1)} = x$
  - $z^{(2)} = \Theta^{(1)} a^{(1)}$
  - $a^{(2)} = g(z^{(2)})\ \ (add\, a_0^{(2)})$
  - $z^{(3)} = \Theta^{(2)} a^{(2)}$
  - $a^{(3)} = g(z^{(3)})\ \ (add\, a_0^{(3)})$
  - $z^{(4)} = \Theta^{(3)} a^{(3)}$
  - $a^{(4)} = h_\Theta(x) = g(z^{(4)})$
- Then we compute gradient (the derivatives) by using backpropagation. We compute the error in the activation of node j in layer l: $\delta_j^{(l)}$.
  - Ex. with 4 layers, we're going to compute:
    $\delta_j^{(l)} = "error"\ of\, node\, j\, in\, layer\, l.$
    For each output unit (layer $L = 4$):
    $\delta_j^{(4)} = a_j^{(4)} - y_j$ $\longrightarrow$the $a$ term can also be written as $(h_\Theta(x))_j$. The $\delta$term outputs the difference between what the hypothesis outputs and what's in the training set. Additionally if you think of $\delta$, $a$, $y$ as vectors we can vectorize the computation and have $\delta^{(4)} = a^{(4)} - y$. Each of $\delta$, $a$, $y$'s dimension is equal to the number of output units in the network. Now we have the error terms. Next we comput $\delta$ terms for the earlier terms in the network.

    $$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} . * g'(z^{(3)})$$
    $$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} . * g'(z^{(2)})$$

    To compute

    $$g'(z^{(3)}) = a^{(3)} . * (1 - a^{(3)})$$

    1 is a vector of 1s

  - There is **no** $\delta^{(1)}$term.

- To calc backpropagation given a training set $\left\{(x^{(1)},\, y^{(1)},\, ...,\, (x^{(m)},\, y^{(m)})\right\}$ - large training set.
- Set $\Delta_{ij}^{(l)} = 0\, (\forall_{l,\, i,\, j})$ $\longrightarrow$eventually this will be used to compute the derivative term

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)} \qquad (ignoring\, \lambda;\; if\, \lambda = 0)$$

- Then loop through the training set:
  For $i = 1$to $m$
  - set $a^{(1)} = x^{(i)}$
  - Perform forward propagation to compute $a^{(l)}$for $l = 2,\, 3,\, ...,\, L$
  - Using output label $y^{(i)}$from a specific example, compute the error term

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

    for the output layer $L$. $a^{(L)}$is what the hypothesis outputs, minus what the target label was, $y^{(i)}$
  - Use backprop algo to compute

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

  - Next, we accumulate the partial derivative terms from previous line

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

    . We can vectorize this too:

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

- Finally, outside the for loop, we compute gradient matrices $D$:

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

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

- Once we compute these terms, those are exactly the derivative of the $J(\Theta)$ terms, you can use them in gradient descent or other optimization algorithms.

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

## 1.3 Backpropagation Intuition

### 1.3.1 Forward propagation.

When performing forward propagation, we may have some $(x^{(i)},\, y^{(i)})$ that we feed into input layer $x_2^{(i)}$ and when we forward propagate it to the hidden layer, the first thing we compute are the weights $z_1^{(2)} \rightarrow a_1^{(2)}$ and $z_2^{(2)} \rightarrow a_2^{(2)}$ we apply sigmoid of the logistic function to z values give us activation values $a$. Then we again propagate to layer 3 until we get the final output value of the neural network $z_1^{(4)} \rightarrow a_1^{(4)}$, for a network with 4 layers.

Turns out back propagation is doing a similar process, but the computations flow from right to left.

Final graph at 4:58

### 1.3.2 What is backpropagation doing?

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

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

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

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

I.e. how well is the network doing on example $i$ (how close is the observed output to the label $y^{(i)}$?

[6.50] - One useful intuition is that BP is computing erros of cost for $a_j^{(l)}$.

The $\delta$ terms are a measure of how much we want to change the network's weights in order to affect the intermediate values $z$ so as to affect the final value of the neural network $h(x)$ and therefore the overal cost.

[8.44] - what is back propagation doing.

For the output layer it firsts sets the $\delta$ term, say $\delta_1^{(4)} = y^{(i)} - a_1^{(4)}$ then we'll back propagate theset terms and compute $\delta$ temrs in that layer, say 3, then propagate further and compute on layer 2. [9.48] - look how we end up at that delta computed on that node. The bias units always output $+1$ or do output $\delta$ values, depending on how back propagation is defined, but you can discard them.

## 1.4 Implementation Note: Unrolling Parameters

Unrolling from matrices or vectors to use advanced optimization algos. The routines assume that $\theta$ and gradient are vectors $\mathbb{R}^{n+1}$. That was fine when we were using logistic regression. Now they're no longer vectors but matrices. Similarly gradient terms are matrices $D^{(l)}$.

How to unroll them to be suitable for functions such as $fminunc$ - 2.05.

Ex:

$s_l$= number of units in layer $l$

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

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

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

### 1.4.1 To unroll:

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

To go back from vector representations to matrix representations:

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

Ex:

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

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

then

```
reshape(DVec(62:71), 1, 11)
```

would get D2 back from DVec.

#### 1.4.2 Learning algorithm:

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

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

-
  ```
  fminunc(@costFunction, initialTheta, options)
  function [jval, gradientVec] = costFunction(thetaVec)
  ```

  From thetaVec, use reshape to get $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$.

  - Use forward/back propagations to compute $D^{(1)}$, $D^{(2)}$, $D^{(3)}$and $J(\Theta)$.
  - Unroll $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ to get gradeintVec, what is what the function can now return.

### 1.5 Gradient Checking

Backprop can appear it's working, a.e. gradient decreasing, even though there may be an error. Use gradient checking.

Suppose you have a function $J(\Theta)$ and suppose we want to estimate a derivative.

$\Theta$is a real number

The procedure for estimating the derivative when $\Theta \in \mathbb{R}$:

- pick $\Theta - \epsilon$ and $\Theta + \epsilon$
- pick points on curve that correspond to above and connect with a line, the actual derivative is tangent to the curve at $\Theta$
- compute slope of the line:
  - vertical height: $J(\Theta + \epsilon) - J(\Theta - \epsilon)$
  - horizontal width: $2\epsilon$
- so the approximation is:

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

  this is 2 sided difference, slightly more accurate than 1 sided with $\epsilon$ in denominator.

- in octave:

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

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

$\Theta = \Theta_1, \Theta_2, ..., \Theta_n$

then

$$\frac{\partial}{\partial \Theta_1} J(\Theta) \approx \frac{J(\Theta_1 + \epsilon, \Theta_2, \Theta_3, ..., \Theta_n) - J(\Theta_1 - \epsilon, \Theta_2, \Theta_3, ..., \Theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial \Theta_2} J(\Theta) \approx \frac{J(\Theta_1, \Theta_2 + \epsilon, \Theta_3, ..., \Theta_n) - J(\Theta_1, \Theta_2 - \epsilon, \Theta_3, ..., \Theta_n)}{2\epsilon}$$

$$\vdots$$

$$\frac{\partial}{\partial \Theta_n} J(\Theta) \approx \frac{J(\Theta_1, \Theta_2, \Theta_3, ..., \Theta_n + \epsilon) - J(\Theta_1, \Theta_2, \Theta_3, ..., \Theta_n - \epsilon)}{2\epsilon}$$

### 1.5.1 In Octave:

```
for i = 1:n,
        thetaPlus = theta;
        thetaPlust(i) = thetaPlus(i) + EPSILON;
        thetaMinus = theta;
        thetaMinus(i) = thetaMinus(i) - EPSILON;
        gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*EPSILON)
end;
```

$n$ = dimension of parameter vector theta

$$\text{thetaPlus} = \begin{bmatrix} \Theta_1 + \epsilon \\ \Theta_2 + \epsilon \\ \Theta_i + \epsilon \\ \vdots \\ \Theta_n + \epsilon \end{bmatrix}$$

$$\text{thetaMinus} = \begin{bmatrix} \Theta_1 - \epsilon \\ \Theta_2 - \epsilon \\ \Theta_i - \epsilon \\ \vdots \\ \Theta_n - \epsilon \end{bmatrix}$$

gradApprox = gives the approximation to the $\frac{\partial}{\partial \Theta_i} J(\Theta)$

We take DVec from packprop and make sure that it's approximately equal to gradApprox

gradApprox$\approx$DVec

### 1.5.2 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.

### 1.6 Random initialization

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

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

Consider gradient descent

Set:

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

Initializing to 0 **does not work** with a neural network. All parameters will end up being the same, even after updates, they'll be non-zero but identical.

This problem is called a **problem of symmetric weights**.

To get around this problem, we use **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.

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

rand(n, m) $\rightarrow$random $n \times m$ matrix between 0 and 1.

### 1.7 Putting It Together

1. Pick a network architecture
   (a) Number of input units: Dimension of features $x^{(i)}$
   (b) Number of output units: Number of classes
   (c) reasonable default for number of hidden layers: 1, or if >1, have same number of hidden units in every layer (usually the more the better but more computationally expensive). The number of hidden features may be comparable to several (2, 3) times larger than number of input features.
2. Training a neural network:
   (a) Randomly initialize weights, usually to values near 0
   (b) Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
   (c) Implement code to compute cost function $J(\Theta)$
   (d) Implement backdrop to compute partial derivatives $\frac{\partial}{\partial\Theta_{jk}^{(l)}} J(\Theta)$
      i. for i=1:m (there exist implementations with no for loop, complex)
         A. Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$
         (Get activations $a^{(l)}$and delta terms $\delta^{(l)}$for $l = 2, ..., L$)
         B. compute delta terms $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$
      ii. compute derivative terms $\frac{\partial}{\partial\Theta_{jk}^{(l)}} J(\Theta)$
3. Use gradient checking to comare $\frac{\partial}{\partial\Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient of $J(\Theta)$
   Then disable gradient checking code.
4. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters $\Theta$.
   $J(\Theta)$ - **non-convex**, can get stuck in a local minimum, but it's usually not a problem in practice.

When using gradient descent with neural networks to try to minimize $J(\Theta)$, the way to make sure the learning algorithm is running correctly would be to plot $J(\Theta)$ as a function of the number of iterations and make sure it is decreasing (or at least non-increasing) with every iteration.