Neural Networks

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$$
$$L = \text{total no. of layers in network}$$
$$s_l = \text{no. of units (not counting bias unit) in layer } l$$

Binary classification: $y = 0 \text{ or } 1$:

- $h_\Theta \in \mathbb{R}$;
- $s_L = 1$.

Multi-class classification (K classes): $y \in \mathbb{R}^K$:

$$y \in \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \right\}$$

- $h_\Theta \in \mathbb{R}^K$;
- $s_L = K$.

Cost function

- Generalization of the logistic regression's cost function:
  - $h_\Theta(x) \in \mathbb{R}^K$: $\big(h_\Theta(x)\big)_i = i$-th output.

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

Back-propagation Algorithm

- Aim: $\underset{\Theta}{min} J$.
- Need to compute:
  - $J(\Theta)$;
  - $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$.

Computation:

Given one training example (x, y):

- Forward propagation:

$$
\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \quad (\text{add } a_0^{(2)}) \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g(z^{(3)}) \quad (\text{add } a_0^{(3)}) \\
z^{(4)} &= \Theta^{(3)} a^{(3)} \\
a^{(4)} &= h_\Theta(x) = g(z^{(4)})
\end{aligned}
$$

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

For each **output** unit (*e. g.*, layer L= 4):

- $\delta_j^{(4)} = a_j^{(4)} - y_j$:
  - $a_j^{(4)} = (h_\Theta(x))_j$.
- $\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} \odot g'(z^{(3)})$:
  - $g'(\cdot)$ is the derivative of the activation function;
  - For the sigmoid function, $g'(z^{(3)}) = a^{(3)} \cdot (1 - a^{(3)})$.
- $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} \odot g'(z^{(2)})$;
- There is no $\delta^{(1)}$.

$$
\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)} \quad (\lambda = 0)
$$

Algorithm:

Training set $\left\{ (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)}) \right\}$.

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

For $i = 1$ to m :

- Set $a^{(1)} = x^{(i)}$;
- Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$;
- Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$;
- Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$;
- $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$:
  - <u>Matrix form</u>: $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$.

$$
D_{ij}^{(l)} := \begin{cases} \dfrac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}, & j \neq 0 \\ \dfrac{1}{m} \Delta_{ij}^{(l)}, & j = 0 \end{cases} \quad \Rightarrow \quad \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}
$$

Unrolling parameters:

Example:

$$
\begin{aligned}
&s_1 = 10, s_2 = 10, s_3 = 1 \\
&\Theta^{(1)} \in \mathbb{R}^{10 \times 11}, \Theta^{(2)} \in \mathbb{R}^{10 \times 11}, \Theta^{(3)} \in \mathbb{R}^{1 \times 11} \\
&D^{(1)} \in \mathbb{R}^{10 \times 11}, D^{(2)} \in \mathbb{R}^{10 \times 11}, D^{(3)} \in \mathbb{R}^{1 \times 11}
\end{aligned}
$$

```
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(@costFuncion, initialTheta, options)`.

`function [jVal, gradientVec] = costFunction(thetaVec)`:

- From `thetaVec`, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
- 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`.

Numerical estimation of gradients

Two-sided difference:

$$\frac{d}{d\Theta} J(\Theta) \approx \frac{J(\Theta + \varepsilon) - J(\Theta - \varepsilon)}{2\varepsilon}, \quad \varepsilon \ll 1.$$

- For instance, $\varepsilon = 10^{-4}$.

One-sided difference:

$$\frac{d}{d\Theta} J(\Theta) \approx \frac{J(\Theta + \varepsilon) - J(\Theta)}{\varepsilon}, \quad \varepsilon \ll 1.$$

- The two-sided difference usually provides a better estimation.

Implementation:

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

Parameter vector θ :

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

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

Then,

$$\begin{aligned}
\frac{\partial}{\partial \theta_1} J(\theta) &\approx \frac{J(\theta_1 + \varepsilon, \theta_2, \dots, \theta_n) - J(\theta_1 - \varepsilon, \theta_2, \dots, \theta_n)}{2\varepsilon} \\
\frac{\partial}{\partial \theta_2} J(\theta) &\approx \frac{J(\theta_1, \theta_2 + \varepsilon, \dots, \theta_n) - J(\theta_1, \theta_2 - \varepsilon, \dots, \theta_n)}{2\varepsilon} \\
&\vdots \\
\frac{\partial}{\partial \theta_n} J(\theta) &\approx \frac{J(\theta_1, \theta_2, \dots, \theta_n + \varepsilon) - J(\theta_1, \theta_2, \dots, \theta_n - \varepsilon)}{2\varepsilon}
\end{aligned}$$

Implementation:

```
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` $\approx$ `DVec`.
  - `DVec` results from back-propagation.

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.

Random initialization

Initial value of Θ :

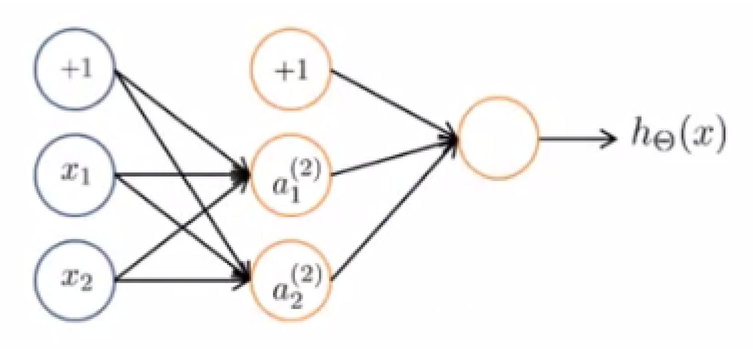
For gradient descent and advanced optimization method, we need an initial value for Θ .

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

Consider gradient descent:

- Set `initialTheta = zeros(n,1)` ?

Zero initialization:



- If $\Theta_{ij}^{(l)} = 0, \; \forall \, i, j, l$:
  - $a_1^{(2)} = a_2^{(2)}$ and $\delta_1^{(2)} = \delta_2^{(2)}$;
  - $\dfrac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \dfrac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$ and $\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$;
  - After each update, parameters corresponding to inputs going into each two hidden units are identical.

Random initialization: Symmetry breaking

- Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$: $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$.
  - This ϵ is unrelated to that of gradient checking.

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

Putting it together

Training a neural network

Pick a network architecture (connectivity pattern between neurons)

- Number of input units: dimension of features $x^{(i)}$;
- Number of output units: number of classes;
- Reasonable default: one hidden layer:
  - If more than one hidden layer, have the same number of hidden units in every layer (usually the more the better).

Steps:

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 back-prop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$.

`for i = 1:m`

- Perform forward propagation and back-propagation using example $(x^{(i)}, y^{(i)})$;
  - Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, 3, \ldots, L$.

5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using back-propagation vs. using numerical estimate of gradient of $J(\Theta)$.

- Then disable gradient checking code.

6. Use gradient descent or advanced optimization method with back-propagation to try to minimize $J(\Theta)$ as a function of parameters Θ .