10 11 2020 Week 3 - Logistic Regression Functions

Logistic Regression for Classification Problems

$h_\theta(x) = g(\theta^T x)$

$g(z) = \frac{1}{1+e^{-z}}$ → Sigmoid function

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$

$0 \le h_\theta(x) \le 1$

$h_\theta(x)$ is going to be conceptually represented as the probability that $y = 1$

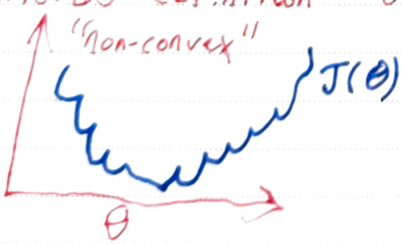
$y \in \{0, 1\}$

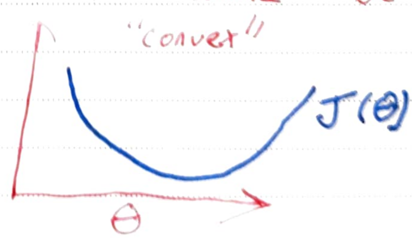
Cost Function:

Linear R

$$\text{cost}(h_\theta(x^{(i)}), y) = \tfrac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

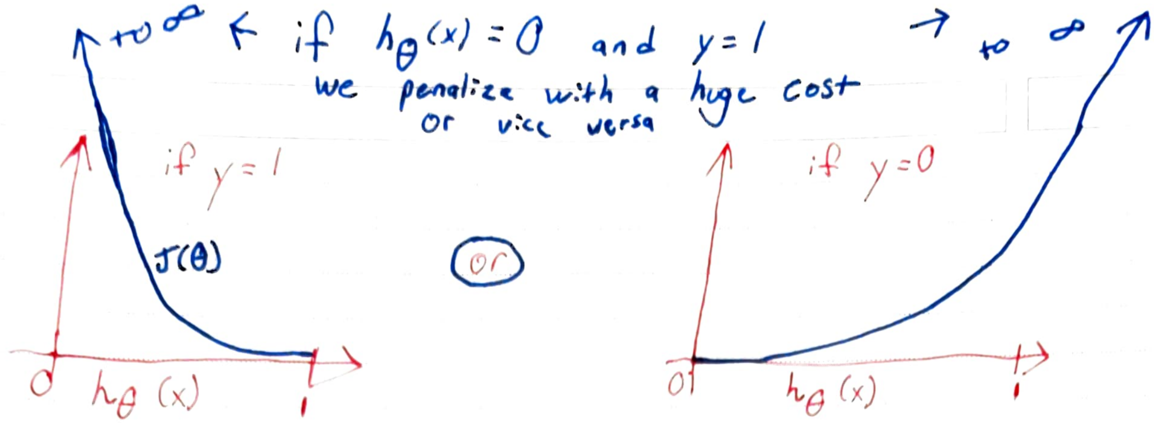
if using the linear regr... ...tion with the sigmoid definition of $(h_\theta(x^i))$ ...would be "non-convex"





Logistic Regression:

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y=1 \\ -\log(1-h_\theta(x)) & \text{if } y=0 \end{cases}$$



Logistic Regression Cost Function:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(h_\theta(x^{(i)})) + (1-y^{(i)}) \log(1-h_\theta(x^{(i)}))$$

Gradient Descent:

Uni Variate

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

$$\theta := \theta - \alpha \frac{1}{m} \sum_{i=1}^{m} [(h_\theta(x^{(i)}) - y^{(i)}) x^{(i)}]$$

$$\theta := \theta - \frac{\alpha}{m} X^T (g(X\theta) - \vec{y})$$

Week 3 - Overfitting

Regularization helps with overfitting by making theta smaller.

Gradient Descent:

$$\theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right]$$

Cost Function :

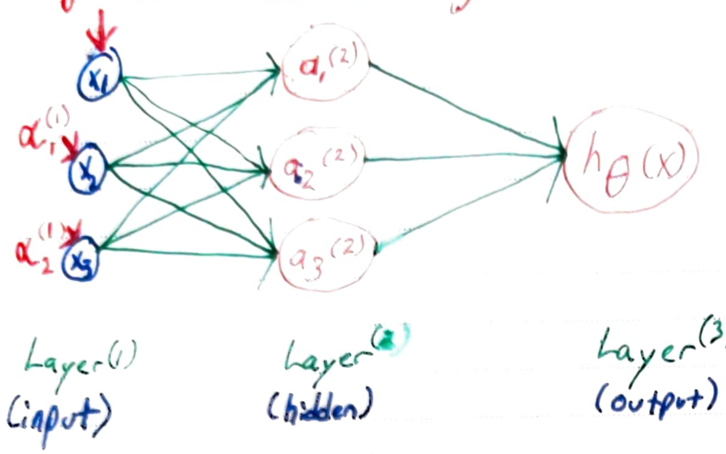
$$J(\theta) = \left[-\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

We can find performance issues ... ie. low cost but not the lowest cost through gradient checking

Gradient Checking:

Week 4 - Neural Networks

$a_0^{(1)} = x$ * can be thaght of as the 1st activation layer



$a_i^{(j)}$ = activation of unit i in layer j

θ_j = matrix of weights controlling function mapping from layer j to j+1

$$a_1^{(2)} = g(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3)$$

$$a_2^{(2)} = g(\theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3)$$

$$a_3^{(2)} = g(\theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3)$$

$$h_\theta(x) = a_1^{(3)} = g(\theta_{10}^{(2)} a_0^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)})$$

Vectorized Approach

Previously $g(z) = g(\theta^T x)$ therefore:

now $z_1^{(2)} = \theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3$

$$z_2^{(2)} = \theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3$$

$$z_3^{(2)} = \theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3$$

$$z^{(2)} = \theta^{(1)} x \quad \text{or} \quad z^{(2)} = \theta^{(1)} a^{(1)}$$

$a^{(2)} = g(z^{(2)})$ ← element wise sigmoid of z

*we also need to add a bias - $a_0^{(2)}$ *

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

$$\boxed{h_\theta(x) = a^{(3)} = g(z^3)}$$

Week 5 - Neural Network Backpropagation

Cost Function

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

*not ith training example

L = total # of layers
s_l = # of units (not counting bias) in layer l
K = # of output units / classes
j = node in layer l

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

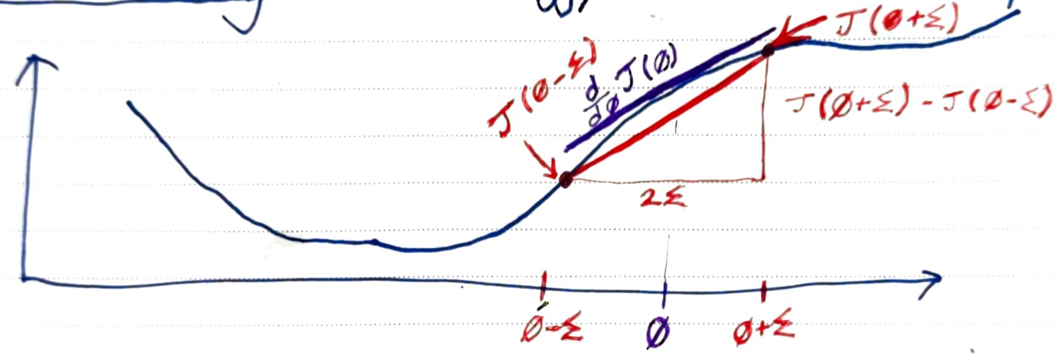
To calculate error for any given node:

$$\delta_j^{(l)} = a_j^{(4)} - y_j \quad \text{or} \quad \delta^{(4)} = a^{(4)} - y \rightarrow \delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

# of output units

$\boxed{\text{NO } \delta^{(1)}}$

Gradient Checking: finds "buggy" or inefficient implementation



QED: $$\frac{d}{d\theta} J\theta \approx \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$$ $\varepsilon \approx 10^{-4}$

Vectorized:

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

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

$\vdots$

$$\frac{d}{d\theta_n}$$

continued →

Week 5 - Neural Network Backpropagation

Gradient Checking Implementation:

theta is a vector (unrolled)

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

1. Implement backprop to compute DVec (unrolled deltas)
2. Implement Gradient Checking to compute gradApprox
3. Turn off gradient checking before training

Random Initialization:

How can we initialize theta?

* We cannot initialize all thetas to zero or all neurons will be the same.

aka Symmetry Breaking

Set each θ to a random value between $-\varepsilon$ and ε

10 x 11 matrix between 0 and 1

theta1 = rand(10, 11) * (2* epsilon) - epsilon*

* not the same as epsilon in gradient checking

Week 5 - Overview for Neural Networks

1. Network Architecture

- input
- output
- hidden (usually 1 layer or same # of units per layer)
  - more units the better (usually more than # of inputs)

2. Training

1. Randomly initialize weights for θ
2. Implement foward propagation to get $h_\theta(x^{(i)})$ or a
3. Implement cost function to get $J(\theta)$
4. Implement backward propagation to get $\frac{\partial}{\partial \theta_{jk}^{(i)}} J(\theta)$

   for i = 1:m
   forward prop(); returns a (activations)
   backward prop(); returns $\frac{\partial}{\partial \theta_{jk}^{(i)}} J(\theta)$ (derivative)
5. Use gradient checking to verify backprop, then disable it.
6. Use a optimization function like gradient descent to minimize $J(\theta)$

English Translations

1. Where do we start on the hill?
2. What is our guess? hypothesis?
3. How wrong was our guess?
4. Which direction to the bottom of the hill? (compass)
5. Is our compass accurate?
6. Use all the info above to make better guesses... each iteration.

$$\frac{\partial}{\partial \theta_{jk}^{(i)}} J\theta = a_j^{(l)} \delta_i^{(l+1)} *$$

ignoring λ | if $\lambda = 0$

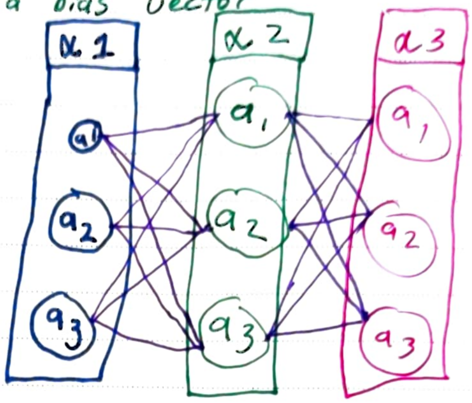
Week 5 - Mechanical Implementation of Backprop

Forward Prop:

```
y_bin = eye(num_labels)[y.T]      # create an identity matrix for y

bias    = np.ones((m, 1))        # create a bias vector
noBias1 = np.delete(Theta1, 0, 1)
noBias2 = np.delete(Theta2, 0, 2)

a1 = concat([bias, X], axis=1)
z2 = a1 @ Theta1.T
⋮        # work forward
```



Cost Function:

```
J = (-1/m) * np.sum(y_bin*log(h) + (1-y_bin)*log(1-h))
```

Regularization:

```
reg = (lambda_/(2*m))*(np.sum(noBias1**2) + np.sum(noBias2**2))

J += reg
```

Backprop:

```
d3 = a3 - y_bin
d2 = d3@noBias2 * sigmoidgradient(z2)
d1 = "no such thing as error for layer 1"

Delta1 = d2.T@a1       # use Delta1 (Δ) as an "accumulator" for our deltas (δ)
Delta2 = d3.T@a2

Theta1_grad = (1/m)*Delta1     # Adjust our respective theta
Theta2_grad = (1/m)*Delta2
# Add our regularization term
Theta1_grad[:,1:] = Theta1_grad[:,1:]+1*lambda_/m*Theta1[:,1:]
Theta2_grad[:,1:] = Theta2_grad[:,1:]+1*lambda_/m*Theta1[:,1:]

grad = np.concatenate([Theta1_grad.ravel(), Theta2_grad.ravel()])
```