Neural Network: Learning

Termanteus

12 January 2020

Contents

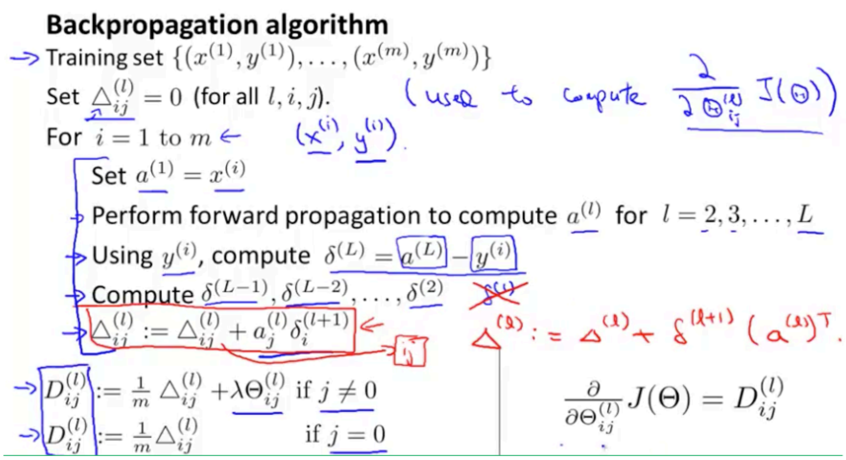
I. Cost Function

- Define:
  - L = total of layers in network
  - s_l = # units in layer l (exclude bias unit)
  - K = # output units/classes
- Cost function:

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

II. Backpropagation

- This is neural-network terminology for **minimizing cost function**.
- Goal: compute $min_\Theta J(\Theta)$ -> compute partial derivative of $J(\Theta)$



- With:

$$\delta^{(l)} = ((\Theta^{(l)})^T\delta^{l+1}).*g'(z^{(l)});(g'(z^{(l)}) = a^{(l)}.*(1-a^{(l)})$$

III. Octave: Unrolling Params

- Turn matrix to one vector: A(:)
- Combine multiple matrices to a matrix multiple column vectors: [A(:), B(:), C(:)]
- Turn one vector back to matrix: reshape(A(from_element:to_element), rows, columns)
- Apply to fminunc(@costFunc, initTheta, options):
  - First put $Theta_1, Theta_2, \ldots, Theta_n$ to columns vector to put into initTheta
  - Inside costFunc, unroll to Theta_1,Theta_2,..., Theta_n
  - Calculate $D^{(1)}, D^{(2)}, \ldots$ and then unroll to gradVec to return.

IV. Gradient Checking

- Gradient checking will assure that the backpropagation implementation works as expected.

$$\frac{\delta}{\delta\Theta_j} \approx \frac{J(\Theta_1, ..., \Theta_j + \varepsilon, ..., \Theta_n)}{2\varepsilon}$$

Hence, we are only adding or subtracting epsilon to the Θ_j matrix. In octave we can do it as follows:

```
epsilon = 1e-4;
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) += epsilon;
  thetaMinus = theta;
  thetaMinus(i) -= epsilon;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;

```

Figure 1: Gradient Checking Algorith

V. Random initialization for Theta

- Initializing all weights to zero doesn't work with neural network since all nodes will update to the same value repeatedly.

Hence, we initialize each $\Theta_{ij}^{(l)}$ to a random value between$[-\epsilon, \epsilon]$. Using the above formula guarantee bound. The same procedure applies to all the Θ's. Below is some working code you could use to expe

```
If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11.

Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta3 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;

```

Figure 2: Random Theta Algo

VI. Training a Neural Network

1. Design a neural network

- # input units = dimension of features
- # output units = # classes

- # hidden units per layer = the more the better (but the more the complex)
- **Default**: 1 hidden layer, if > 1, hidden units/layer >= the previous layer.

2. Training Neural Network

- Random initialize the weights
- Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
- Implement cost function
- Implement backpropagation to compute partial derivatives
- Use gradient checking to confirm the previous step works, then disable it..
- Use gradient descent or built-in optimization function to minimize the cost function with the weights in theta.
- **Notes**: Ideally, we want $h_\Theta(x^{(i)}) \approx y^{(i)}$. But $J(\Theta)$ is not convex so we might end up in a local minimum.