Week 5 Notes

Cost Function and Backpropagation

Cost Function

Suppose we have a neural network with L total layers and s_l units (not counting the bias unit) in layer l. For binary classification, i.e., $y = \{0, 1\}$, then we get one output unit $h_\Theta(x) \in \mathbb{R}$, or $K = 1$. For multiclass classification, $y \in \mathbb{R}^K$, with K output units.

The cost function will be a generalization of the logistic regression cost function:

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

where $(h_\Theta(x))_i$ is the i^{th} output. Note that each sum starts at 1 - this is because of the convention that we don't do this for the bias units.

Backpropagation Algorithm

Gradient computation

Given one training example (x, y):

$$\text{forward propagation}:$$

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

Intuition: $\delta_j^{(l)} =$ "error" of node j in layer l. The error for each layer can be calculated using backpropagation:

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

where $.*$ is the element-wise multiplication operator and $g'(z^{(l)}) = a^{(l)} .* (1 - a^{(l)})$.

For a training set with m elements:

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

For $i = 1$ to m

- set $a^{(1)} = x^{(i)}$
- use forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$
- using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
- compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
- $\Delta_{ij}^{(l)} := \Delta_{ij} + a^{(l)} \delta^{(l+1)}$
- vectorized: $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

Adding in regularization:

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

$D_{ij}^{(l)}$ is called an "accumulator", and it is equal to the partial derivative of $J(\Theta)$.

Backpropagation Intuition

What is backpropagation doing? it is computing the "error" in the activation values for the network.

Backpropagation in Practice

Implementation Note: Unrolling Parameters

```
function [jVal, gradient] = costFunction(theta)
  ...

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

For a neural network with $L = 4$:

$\Theta^{(1)}, \Theta^{(1)}, \Theta^{(1)}$ - matrices (`Theta1` , `Theta2` , `Theta3`)

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

To "unroll" these matrices into vectors:

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

% reshape into matrices (for s1=10, s2=10, s3=1):
Theta1 = reshape(thetaVec(1:110), 10, 11);
Theta2 = reshape(thetaVec(111:220), 10, 11);
Theta3 = reshape(thetaVec(221:231), 10, 11);
```

When we are given initial parameters for Θ, unroll them to get `initialTheta` to pass to `fminunc` .

```
function [jVal, gradientVec] = costFunction(thetaVec)
  % from thetaVec, get Theta1, Theta2, Theta3, etc.
  % use propagation to compute D1, D2, D3, and J(Theta)
  % unroll D1, D2, D3 to return gradientVec
```

Gradient Checking

We can numerically approximate the gradient of $J(\theta)$ as $\frac{d}{d\theta}J(\theta) \approx \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}$ where ε is some small real number.

Implemented in Octave:

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

When θ is a vector instead of a single real number, the calculation is similar.

```
% for n-dimensional vector theta
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;
```

We want to check that `gradApprox` $\approx$ `DVec` from backpropagation.

When implementing the above code, be sure to only run the numerical gradient checking to make sure that `DVec` is working correctly. When training the neural network, make sure it is turned off, otherwise it will run very slowly!

Random Initialization

In order to run the optimization algorithms, we need to choose some initial values for Θ. Before, starting with all 0s worked, but it will not for training a neural network. This will lead to a redundant set of weights. Instead, we initialize each $\Theta_{ij}^{(l)}$ to be a random value in $[\epsilon, \epsilon]$. In Octave:

```
% creates a random 10x11 matrix between -INIT_EPSILON and +INIT_EPSILON
Theta1 = rand(10,11)*(2*INIT_EPSILON) - INIT_EPSILON;
```

Putting it Together

When training a neural network, first choose some architecture (the connectivity between neurons). The number of input units will be the dimension of the features $x^{(i)}$ and the number of output units will be the number of classes. For multiclass problems, the output classes should be one-hot encoded as column vectors.

For hidden layers, it is most common to use the same number of units in each layer. The number of units in each layer is also usually equal to or slighly greater than the number of features.

The general steps of training a neural network:

1. randomly initialize weights
2. implement forward propagation to get $h_\Theta(x)$ for any $x^{(i)}$
3. implement code to compute the cost function $J(\Theta)$
4. implement backpropagation to compute partial derivatives $\partial/\partial\Theta_{jk}^{(l)}\ J(\Theta)$

At first, steps 2 - 3 should be done within a `for` loop. Later, advanced vectorization methods can be applied to optimize this process. Inside the loop, do forward and backpropagation for each training example to get the activations $a^{(l)}$ and the delta terms $\delta^{(l)}$ for each layer.

5. Use gradient checking to compare the derivatives computed using backpropagation to the numerical estimate (then be sure to disable it before step 6)
6. Use gradient descent or another advanced optimization method with backpropagation to minimize $J(\Theta)$ as a function of the parameters Θ

Application of Neural Networks

Autonomous Driving

YouTube Link