CS229a - Week 5

Written by hackerwins *on* June 19, 2019

ML CS229a

Cost Function and Back-propagation

Cost Function

Suppose we have neural networks to classification problems.

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

L = total number of layers in network

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

Binary classification: y = 0 or 1, 1 output unit, $s_L = 1$

Multi-class classification (K classes): $s_L = K$, (K $\geq$ 3)

Cost function of Logistic regression:

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

Cost function of Neural network:

$$h_\Theta(x) \in \R^K \ (h_\Theta(x))_i = i^{th} output$$

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(h_\theta(x^{(i)}))_k + (1 - y_k^{(i)}) log(1 - h_\theta(x^{(i)})_k)]$$

$$+ \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{i=1}^{s_l+1}(\theta_{j,i}^{(l)})^2$$

Back-propagation algorithm

Back-propagation: Minimize computation cost of gradient in gradient descent

Gradient computation

$$\min_\Theta J(\Theta)$$

Need code to compute:

$$\text{Cost function} = J(\Theta), \text{ Gradient} = \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$$

Vectorized implementation of forward propagation:

$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = g(z^{(2)}), \text{ add } a_0^{(2)}$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = g(z^{(3)}), \text{ add } a_0^{(3)}$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$

Gradient computation: intuition of back-propagation algorithm

Error term(l ≥ 2, don't associate an error term with the input layer):

$$\delta_j^{(l)} = \text{error of node } j \text{ in layer } l$$

For each output unit (layer L = 4)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

Vectorized implementation(vector whose dimension is equal to the number of output units):

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

Term g prime of z3(derivative of the g function of the activation function):

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

$$\text{Gradient: } \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^l \delta_i^{(l+1)}$$

Back-propagation algorithm

Training set

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

Set Δ:

$$\Delta_{ij}^{(l)} = 0, \text{ for all l,i,j}$$

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

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

- Vectorized implementation of above equation:

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

Outside for loop(The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative):

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

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

Gradient:

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

> And I've actually used backpropagation, you know, pretty successfully for many years. And even today I still don't sometimes feel like I have a very good sense of just what it's doing, or intuition about what back propagation is doing. - Andrew Ng

Back-propagation Intuition

For the purpose of intuition, feel free to think of the cost function as being the sort of the squared error cost function.

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

Delta term: partial derivative with respect to z,l,j, that is this weighted sum of inputs that were confusing these z terms. Partial derivatives with respect to these things of the cost function.

$$\delta_j^{(l)} = \text{error of node } j \text{ in layer } l$$

$$\text{Formally, } \delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(i), \text{ for } j \geq 0, \text{ where}$$

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

$$\delta_2^{(2)} = \Theta_{12}^{(2)} * \delta_1^{(3)} + \Theta_{22}^{(2)} * \delta_2^{(3)}$$

Back-propagation in Practice

Implementation Note: Unrolling Parameters

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

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

Neural Network (L=4):

$$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)} \text{ -matrices(Theta1, Theta2, Theta3)}$$
$$D^{(1)}, D^{(2)}, D^{(3)} \text{ -matrices(D1, D2, D3)}$$

Example

$$s_1 = 10, s_2 = 10, s_3 = 1$$
$$\Theta^{(1)} \in \R^{10\times11}, \Theta^{(2)} \in \R^{10\times11}, \Theta^{(3)} \in \R^{1\times11}$$
$$D^{(1)} \in \R^{10\times11}, D^{(2)} \in \R^{10\times11}, D^{(3)} \in \R^{1\times11}$$

In order to use optimizing functions such as `fminunc()` , we will want to unroll all the elements and put them into one long vector.

```
% Unrolling
```

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

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

Gradient Checking

One unfortunate property is that there are many ways to have subtle bugs in back propagation. So that if you run it with gradient descent or some other optimization algorithm, it could actually look like it's working. There's an idea called Gradient Checking that eliminates almost all of these problems.

Numerical estimation of gradients

$$J(\Theta), \Theta \in \R$$

Compare derivative with two sided difference

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

use small value for epsilon i.g. $\epsilon = 10^{-4}$

```
gradApprox = (J(theta + EPSILON) - J(theta - EPSILON))
```

Parameter vector theta

$$\frac{\partial}{\partial\theta_1}J(\Theta) \approx \frac{J(\theta_1+\epsilon,\theta_2,\dots\theta_n)-J(\theta_1-\epsilon,\theta_2,\dots\theta_n)}{2\epsilon}$$
$$\dots$$
$$\frac{\partial}{\partial\theta_n}J(\Theta) \approx \frac{J(\theta_1,\theta_2,\dots\theta_n+\epsilon)-J(\theta_1,\theta_2,\dots\theta_n-\epsilon)}{2\epsilon}$$

```
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*E
end

% Check that gradApprox with DVec(from back-propagation
```

Implementation Note

- Implement back-prop to compute DVec(unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
- Implement numerical gradient check to compute gradApprox.
- Turn off gradient checking(Important: this is very slow). Using back-prop code for learning.

Random initialization

Zero initialization

```
% Consider gradient descent
% Whereas this worked okay when we were using logistic
% initializing all of your parameters to zero actually
% when you are trading on your own network.
initialTheta = zeros(n, 1) % ??
```

```
% need initial theta
optTheta = fminunc(@costFunction, intialTheta, options)
```

$$\Theta_{ij} = 0 \text{ for all } i, j, l$$
$$a_1^{(2)} = a_2^{(2)}, \delta_1^{(2)} = \delta_2^{(2)}$$

Even after one iteration gradient descent, You'll find that your two hidden units are still computing exactly the same functions of the inputs. This is a highly redundant representation. In order to get around this problem, the way we initialize the parameters of a neural network therefore is with random initialization.

Random initialization: Symmetry breaking

Initialize each theta_{ij}^{(l)} to a random value between minus epsilon and plus epsilon.

$$\text{i.g. } -\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$$

```
% rand(r, c): r x c matrix between 0 and 1.
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

First, pick a network architecture(connectivity pattern between neurons): Choices of how many units in each layer and how many hidden layers, those are architecture choices.

- Number of input units: dimension of features
- Number of output units: number of classes
- Number of hidden units and number of hidden layers
    - Reasonable default
        - 1 hidden layer
        - if more than one hidden layers, have same number of hidden units in every layer
    - Usually, the more hidden units the better; it's just that if you have a lot of hidden units, it can become more computationally expensive, but very often, having more hidden units is a good thing
    - And usually the number of hidden units in each layer will be maybe comparable to the dimension of x, comparable to the number of features, or it could be any where from same number of hidden units of input features to maybe so that three or four times of that

Implement training

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

Some of you may have heard of advanced, and frankly very advanced factorization methods where you don't have a four-loop over the m-training examples. There should be a for-loop in your implementation of back-prop, at least the first time implementing it. And then there are frankly somewhat complicated ways to do this without a for-loop, but I definitely do not recommend trying to do that much more complicated version the first time you try to implement back-prop.

```
for i = 1:m
  % Perform forward prop and back-prop using example: x
```

```
    % Get activations a^{(l)} and delta terms \delta^{(l)
end
```

$$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T \text{: each steps in for loop}$$

$$w = \frac{\partial}{\partial \theta_{jk}^{(l)}} J(\Theta) \text{: out side the for loop}$$

1. Use gradient checking to compare partial derivative terms that were computed using back-prop vs using numerical estimate of gradient of J of Theta. Then disable gradient checking code.
2. Use gradient descent or advanced optimization method with back-prop to try to minimize J of Theta as a function of parameters Theta.

For neural networks, this cost function j of theta is non-convex and so it can theoretically be susceptible to local minima, and in fact algorithms like gradient descent and the advance optimization methods can, in theory, get stuck in local optima. But it turns out that in practice this is not usually a huge problem and even though we can't guarantee that these algorithms will find a global optimum, usually algorithms like gradient descent will do a very good job minimizing this cost function j of theta and get a very good local minimum, even if it doesn't get to the global optimum.

> Neural network learning and back propagation is a complicated algorithm. And even though I've seen the math behind back propagation for many years and I've used back propagation, I think very successfully, for many years, even today I still feel like I don't always have a great grasp of exactly what back propagation is doing sometimes. And what the optimization process looks like of minimizing j if theta. Much this is a much harder algorithm to feel like I have a much less good handle on

> exactly what this is doing compared to say, linear regression or logistic regression. Which were mathematically and conceptually much simpler and much cleaner algorithms. But so in case if you feel the same way, you know, that's actually perfectly okay. - Andrew Ng.

Autonumous Driving

ALVINN is a system of artificial neural networks that learns to steer by watching a person drive. ALVINN is designed to control the NAVLAB 2, a modified Army Humvee who had put sensors, computers, and actuators for autonomous navigation experiments.