coursera

Week 5 Lecture Notes

ML:Neural Networks: Learning

Cost Function

Let's first define a few variables that we will need to use:

a) L= total number of layers in the network

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

c) K= number of output units/classes

Recall that in neural networks, we may have many output nodes. We denote $h_\Theta(x)_k$ as being a hypothesis that results in the k^{th} output.

Our cost function for neural networks is going to be a generalization of the one we used for logistic regression.

Recall that the cost function for regularized logistic regression was:

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

For neural networks, it is going to be slightly more complicated:

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

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, between the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

Note:

- the double sum simply adds up the logistic regression costs calculated for each cell in the output layer; and
- the triple sum simply adds up the squares of all the individual Θs in the entire network.
- the i in the triple sum does **not** refer to training example i

Backpropagation Algorithm

"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression.

Our goal is to compute:

$\min_\Theta J(\Theta)$

That is, we want to minimize our cost function J using an optimal set of parameters in theta.

In this section we'll look at the equations we use to compute the partial derivative of J(Θ):

$$\frac{\partial}{\partial\Theta_{i,j}^{(l)}}J(\Theta)$$

In back propagation we're going to compute for every node:

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

Recall that $a_j^{(l)}$ is activation node j in layer l.

For the **last layer**, we can compute the vector of delta values with:

$\delta^{(L)} = a^{(L)} - y$

Where L is our total number of layers and $a^{(L)}$ is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y.

To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:

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

The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by z(l).

The g-prime derivative terms can also be written out as:

$$g'(u) = g(u)\ .*\ (1 - g(u))$$

The full back propagation equation for the inner nodes is then:

$$\delta^{(l)} = ((\Theta^{(l)})^T\delta^{(l+1)})\ .*\ a^{(l)}\ .*\ (1 - a^{(l)})$$

A. Ng states that the derivation and proofs are complicated and involved, but you can still implement the above equations to do back propagation without knowing the details.

We can compute our partial derivative terms by multiplying our activation values and our error values for each training example t:

$$\frac{\partial J(\Theta)}{\partial\Theta_{i,j}^{(l)}} = \frac{1}{m}\sum_{t=1}^m a_j^{(t)(l)}\delta_i^{(t)(l+1)}$$

This however ignores regularization, which we'll deal with later.

Note: δ^{l+1} and a^{l+1} are vectors with s_{l+1} elements. Similarly, $a^{(l)}$ is a vector with s_l elements. Multiplying them produces a matrix that is s_{l+1} by s_l which is the same dimension as $\Theta^{(l)}$. That is, the process produces a gradient term for every element in $\Theta^{(l)}$. (Actually, $\Theta^{(l)}$ has s_l + 1 column, so the dimensionality is not exactly the same).

We can now take all these equations and put them together into a backpropagation algorithm:

Back propagation Algorithm

Given training set $\{(x^{(1)},y^{(1)})\cdots(x^{(m)},y^{(m)})\}$

- Set $\Delta_{i,j}^{(l)}$:= 0 for all (l,i,j)

For training example t =1 to m:

- Set $a^{(1)} := x^{(t)}$
- Perform forward propagation to compute $a^{(l)}$ for l=2,3,...,L
- Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$
- Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T\delta^{(l-1)})\ .*\ a^{(l)}\ .*\ (1 - a^{(l)})$
- $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)}\delta_i^{(l+1)}$ or with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l-1)}(a^{(l)})^T$
- $D_{i,j}^{(l)} := \frac{1}{m}\left(\Delta_{i,j}^{(l)} + \lambda\Theta_{i,j}^{(l)}\right)$ If j≠0 NOTE: Typo in lecture slide omits outside parentheses. This version is correct.
- $D_{i,j}^{(l)} := \frac{1}{m}\Delta_{i,j}^{(l)}$ If j=0

The capital-delta matrix is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative.

The actual proof is quite involved, but, the $D_{i,j}^{(l)}$ terms are the partial derivatives and the results we are looking for:

$$D_{i,j}^{(l)} = \frac{\partial J(\Theta)}{\partial\Theta_{i,j}^{(l)}}.$$

Backpropagation Intuition

The cost function is:

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

If we consider simple non-multiclass classification (k = 1) and disregard regularization, the cost is computed with:

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

More intuitively you can think of that equation roughly as:

$$cost(t) \approx (h_\theta(x^{(t)}) - y^{(t)})^2$$

Intuitively, $\delta_j^{(l)}$ is the "error" for $a_j^{(l)}$ (unit j in layer l)

More formally, the delta values are actually the derivative of the cost function:

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}}cost(t)$$

Recall that our derivative is the slope of a line tangent to the cost function, so the steeper the slope the more incorrect we are.

Note: In lecture, sometimes i is used to index a training example. Sometimes it is used to index a unit in a layer. In the Back Propagation Algorithm described here, t is used to index a training example rather than overloading the use of i.

Implementation Note: Unrolling Parameters

With neural networks, we are working with sets of matrices:

$$\Theta^{(1)},\Theta^{(2)},\Theta^{(3)},\dots$$
$$D^{(1)},D^{(2)},D^{(3)},\dots$$

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

```
thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]
deltaVector = [ D1(:); D2(:); D3(:) ]
```

If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11, then we can get back our original matrices from the "unrolled" versions as follows:

```
Theta1 = reshape(thetaVector(1:110),10,11)
Theta2 = reshape(thetaVector(111:220),10,11)
Theta3 = reshape(thetaVector(221:231),1,11)
```

NOTE: The lecture slides show an example neural network with 3 layers. However, *3* theta matrices are defined: Theta1, Theta2, Theta3. There should be only 2 theta matrices: Theta1 (10 x 11), Theta2 (1 x 11).

Gradient Checking

Gradient checking will assure that our backpropagation works as intended.

We can approximate the derivative of our cost function with:

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

With multiple theta matrices, we can approximate the derivative **with respect to Θ_j** as follows:

$$\frac{\partial}{\partial\Theta_j}J(\Theta) \approx \frac{J(\Theta_1, \dots, \Theta_j + \epsilon, \dots, \Theta_n) - J(\Theta_1, \dots, \Theta_j - \epsilon, \dots, \Theta_n)}{2\epsilon}$$

A good small value for ϵ (epsilon), guarantees the math above to become true. If the value be much smaller, may we will end up with numerical problems. The professor Andrew usually uses the value $\epsilon = 10^{-4}$.

We are only adding or subtracting epsilon to the $Theta_j$ matrix. In octave we can do it as follows:

coursera

```
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;
```

We then want to check that gradApprox ≈ deltaVector.

Once you've verified **once** that your backpropagation algorithm is correct, then you don't need to compute gradApprox again. The code to compute gradApprox is very slow.

Random Initialization

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly.

Instead we can randomly initialize our weights:

Initialize each $\Theta^{(l)}_{ij}$ to a random value between $[-\epsilon, \epsilon]$:

$$\epsilon = \frac{\sqrt{6}}{\sqrt{\mathrm{Loutput} + \mathrm{Linput}}}$$

$$\Theta^{(l)} = 2\epsilon \; \mathrm{rand}(\mathrm{Loutput}, \mathrm{Linput} + 1) - \epsilon$$

```
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;
```

rand(x,y) will initialize a matrix of random real numbers between 0 and 1. (Note: this epsilon is unrelated to the epsilon from Gradient Checking)

Why use this method? This paper may be useful: https://web.stanford.edu/class/ee373b/nninitialization.pdf

Putting it Together

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers total.

- Number of input units = dimension of features $x^{(i)}$
- Number of output units = number of classes
- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
- Defaults: 1 hidden layer. If more than 1 hidden layer, then the same number of units in every hidden layer.

Training a Neural Network

1. Randomly initialize the weights
2. Implement forward propagation to get $h_\theta(x^{(i)})$
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

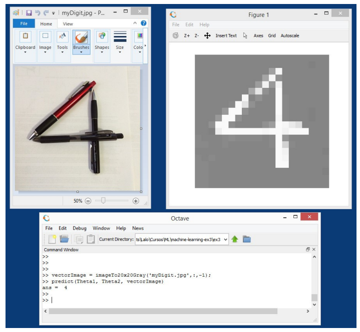
When we perform forward and back propagation, we loop on every training example:

coursera

```
for i = 1:m,
   Perform forward propagation and backpropagation using example (x(i),y(i))
   (Get activations a(l) and delta terms d(l) for l = 2,...,L
```

Bonus: Tutorial on How to classify your own images of digits

This tutorial will guide you on how to use the classifier provided in exercise 3 to classify you own images like this:



It will also explain how the images are converted thru several formats to be processed and displayed.

Introduction

The classifier provided expects 20 x 20 pixels black and white images converted in a row vector of 400 real numbers like this

```
[ 0.14532, 0.12876, ...]
```

Each pixel is represented by a real number between -1.0 to 1.0, meaning -1.0 equal black and 1.0 equal white (any number in between is a shade of gray, and number 0.0 is exactly the middle gray).

.jpg and color RGB images

The most common image format that can be read by Octave is .jpg using function that outputs a three-dimensional matrix of integer numbers from 0 to 255, representing the height x width x 3 integers as indexes of a color map for each pixel (explaining color maps is beyond scope).

```
Image3DmatrixRGB = imread("myOwnPhoto.jpg");
```

Convert to Black & White

A common way to convert color images to black & white, is to convert them to a YIQ standard and keep only the Y component that represents the luma information (black & white). I and Q represent the chrominance information (color).Octave has a function **rgb2ntsc()** that outputs a similar three-dimensional matrix but of real numbers from -1.0 to 1.0, representing the height x width x 3 (Y luma, I in-phase, Q quadrature) intensity for each pixel.

coursera

```
Image3DmatrixYIQ = rgb2ntsc(MyImageRGB);
```

To obtain the Black & White component just discard the I and Q matrices. This leaves a two-dimensional matrix of real numbers from -1.0 to 1.0 representing the height x width pixels black & white values.

```
Image2DmatrixBW = Image3DmatrixYIQ(:,:,1);
```

Cropping to square image

It is useful to crop the original image to be as square as possible. The way to crop a matrix is by selecting an area inside the original B&W image and copy it to a new matrix. This is done by selecting the rows and columns that define the area. In other words, it is copying a rectangular subset of the matrix like this:

```
croppedImage = Image2DmatrixBW(origen1:size1, origin2:size2);

```

Cropping does not have to be all the way to a square.**It could be cropping just a percentage of the way to a square**so you can leave more of the image intact. The next step of scaling will take care of streaching the image to fit a square.

Scaling to 20 x 20 pixels

The classifier provided was trained with 20 x 20 pixels images so we need to scale our photos to meet. It may cause distortion depending on the height and width ratio of the cropped original photo. There are many ways to scale a photo but we are going to use the simplest one. We lay a scaled grid of 20 x 20 over the original photo and take a sample pixel on the center of each grid. To lay a scaled grid, we compute two vectors of 20 indexes each evenly spaced on the original size of the image. One for the height and one for the width of the image. For example, in an image of 320 x 200 pixels will produce to vectors like

```
[9    25    41    57    73 ... 313] % 20 indexes
```

```
[6    16    26    36    46 ... 196] % 20 indexes
```

Copy the value of each pixel located by the grid of these indexes to a new matrix. Ending up with a matrix of 20 x 20 real numbers.

Black & White to Gray & White

The classifier provided was trained with images of white digits over gray background. Specifically, the 20 x 20 matrix of real numbers ONLY range from 0.0 to 1.0 instead of the complete black & white range of -1.0 to 1.0, this means that we have to normalize our photos to a range 0.0 to 1.0 for this classifier to work. But also, we invert the black and white colors because is easier to "draw" black over white on our photos and we need to get white digits. So in short, we **invert black and white** and **stretch black to gray**.

Rotation of image

Some times our photos are automatically rotated like in our celular phones. The classifier provided can not recognize rotated images so we may need to rotate it back sometimes. This can be done with an Octave function **rot90()** like this.

```
ImageAligned = rot90(Image, rotationStep);
```

Where rotationStep is an integer: -1 mean rotate 90 degrees CCW and 1 mean rotate 90 degrees CW.

Approach

1. The approach is to have a function that converts our photo to the format the classifier is expecting. As if it was just a sample from the training data set.
2. Use the classifier to predict the digit in the converted image.

Code step by step

Define the function name, the output variable and three parameters, one for the filename of our photo, one optional cropping percentage (if not provided will default to zero, meaning no cropping) and the last optional rotation of the image (if not provided will default to cero, meaning no rotation).

```
function vectorImage = imageTo20x20Gray(fileName, cropPercentage=0, rotStep=0)

```

Read the file as a RGB image and convert it to Black & White 2D matrix (see the introduction).

```
% Read as RGB image
Image3DmatrixRGB = imread(fileName);
% Convert to NTSC image (YIQ)
Image3DmatrixYIQ = rgb2ntsc(Image3DmatrixRGB );
% Convert to grays keeping only luminance (Y)
%        ...and discard chrominance (IQ)
Image2DmatrixBW  = Image3DmatrixYIQ(:,:,1);
```

Establish the final size of the cropped image.

```
% Get the size of your image
oldSize = size(Image2DmatrixBW);
% Obtain crop size toward centered square (cropDelta)
% ...will be zero for the already minimum dimension
% ...and if the cropPercentage is zero,
% ...both dimensions are zero
% ...meaning that the original image will go intact to croppedImage
cropDelta = floor((oldSize - min(oldSize)) .* (cropPercentage/100));
% Compute the desired final pixel size for the original image
finalSize = oldSize - cropDelta;
```

Obtain the origin and amount of the columns and rows to be copied to the cropped image.

```
% Compute each dimension origin for croping
cropOrigin = floor(cropDelta / 2) + 1;
% Compute each dimension copying size
copySize = cropOrigin + finalSize - 1;
% Copy just the desired cropped image from the original B&W image
croppedImage = Image2DmatrixBW( ...
                    cropOrigin(1):copySize(1), cropOrigin(2):copySize(2));

```

Compute the scale and compute back the new size. This last step is extra. It is computed back so the code keeps general for future modification of the classifier size. For example: if changed from 20 x 20 pixels to 30 x 30. Then the we only need to change the line of code where the scale is computed.

```
% Resolution scale factors: [rows cols]
scale = [20 20] ./ finalSize;
% Compute back the new image size (extra step to keep code general)
newSize = max(floor(scale .* finalSize),1);

```

Compute two sets of 20 indexes evenly spaced. One over the original height and one over the original width of the image.

```
% Compute a re-sampled set of indices:
rowIndex = min(round(((1:newSize(1))-0.5)./scale(1)+0.5), finalSize(1));
colIndex = min(round(((1:newSize(2))-0.5)./scale(2)+0.5), finalSize(2));
```

Copy just the indexed values from old image to get new image of 20 x 20 real numbers. This is called "sampling" because it copies just a sample pixel indexed by a grid. All the sample pixels make the new image.

```
% Copy just the indexed values from old image to get new image
newImage = croppedImage(rowIndex,colIndex,:);

```

Rotate the matrix using the **rot90()** function with the rotStep parameter: -1 is CCW, 0 is no rotate, 1 is CW.

```
% Rotate if needed: -1 is CCW, 0 is no rotate, 1 is CW
newAlignedImage = rot90(newImage, rotStep);

```

Invert black and white because it is easier to draw black digits over white background in our photos but the classifier needs white digits.

```
% Invert black and white
invertedImage = - newAlignedImage;

```

Find the min and max gray values in the image and compute the total value range in preparation for normalization.

```
% Find min and max grays values in the image
maxValue = max(invertedImage(:));
minValue = min(invertedImage(:));
% Compute the value range of actual grays
delta = maxValue - minValue;

```

Do normalization so all values end up between 0.0 and 1.0 because this particular classifier do not perform well with negative numbers.

```
% Normalize grays between 0 and 1
normImage = (invertedImage - minValue) / delta;
```

Add some contrast to the image. The multiplication factor is the contrast control, you can increase it if desired to obtain sharper contrast (contrast only between gray and white, black was already removed in normalization).

```
% Add contrast. Multiplication factor is contrast control.
contrastedImage = sigmoid((normImage -0.5) * 5);
```

Show the image specifying the black & white range [-1 1] to avoid automatic ranging using the image range values of gray to white. Showing the photo with different range, does not affect the values in the output matrix, so do not affect the classifier. It is only as a visual feedback for the user.

```
% Show image as seen by the classifier
imshow(contrastedImage, [-1, 1] );
```

Finally, output the matrix as a unrolled vector to be compatible with the classifier.

```
% Output the matrix as a unrolled vector
vectorImage = reshape(normImage, 1, newSize(1) * newSize(2));
```

End function.

```
end;
```

Usage samples

Single photo

- Photo file in myDigit.jpg
- Cropping 60% of the way to square photo
- No rotationvectorImage = imageTo20x20Gray('myDigit.jpg',60);
  predict(Theta1, Theta2, vectorImage)
- Photo file in myDigit.jpg
- No cropping
- CCW rotationvectorImage = imageTo20x20Gray('myDigit.jpg',:,-1);
  predict(Theta1, Theta2, vectorImage)

Multiple photos

- Photo files in myFirstDigit.jpg, mySecondDigit.jpg
- First crop to square and second 25% of the way to square photo
- First no rotation and second CW rotationvectorImage(1,:) = imageTo20x20Gray('myFirstDigit.jpg',100);
  vectorImage(2,:) = imageTo20x20Gray('mySecondDigit.jpg',25,1);
  predict(Theta1, Theta2, vectorImage)

Tips

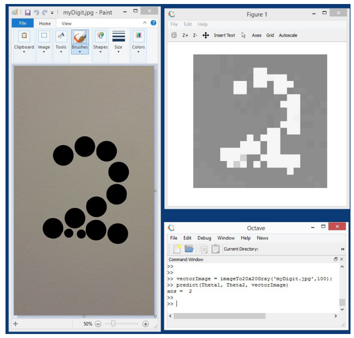
- JPG photos of black numbers over white background
- Preferred square photos but not required
- Rotate as needed because the classifier can only work with vertical digits
- Leave background space around digit. Al least 2 pixels when seen at 20 x 20 resolution. This means that the classifier only really works in a 16 x 16 area.
- Play changing the contrast multipier to 10 (or more).

Complete code (just copy and paste)

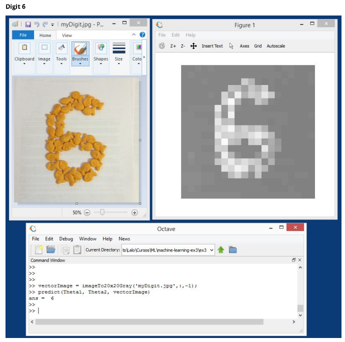
```
function vectorImage = imageTo20x20Gray(fileName, cropPercentage=0, rotStep=0)
%IMAGETO20X20GRAY display reduced image and converts for digit classification
%
% Sample usage:
%       imageTo20x20Gray('myDigit.jpg', 100, -1);
%
%       First parameter: Image file name
%             Could be bigger than 20 x 20 px, it will
%             be resized to 20 x 20. Better if used with
%             square images but not required.
%
%       Second parameter: cropPercentage (any number between 0 and 100)
%             0  0% will be cropped (optional, no needed for square images)
%            50  50% of available croping will be cropped
%           100  crop all the way to square image (for rectangular images)
%
%       Third parameter: rotStep
%            -1  rotate image 90 degrees CCW
%             0  do not rotate (optional)
%             1  rotate image 90 degrees CW
%
% (Thanks to Edwin Frühwirth for parts of this code)
% Read as RGB image
Image3DmatrixRGB = imread(fileName);
% Convert to NTSC image (YIQ)
Image3DmatrixYIQ = rgb2ntsc(Image3DmatrixRGB );
% Convert to grays keeping only luminance (Y) and discard chrominance (IQ)
Image2DmatrixBW  = Image3DmatrixYIQ(:,:,1);
% Get the size of your image
oldSize = size(Image2DmatrixBW);
% Obtain crop size toward centered square (cropDelta)
% ...will be zero for the already minimum dimension
% ...and if the cropPercentage is zero,
% ...both dimensions are zero
% ...meaning that the original image will go intact to croppedImage
cropDelta = floor((oldSize - min(oldSize)) .* (cropPercentage/100));
% Compute the desired final pixel size for the original image
finalSize = oldSize - cropDelta;
% Compute each dimension origin for croping
cropOrigin = floor(cropDelta / 2) + 1;
```

Photo Gallery

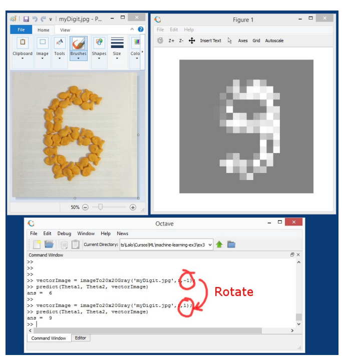
Digit 2



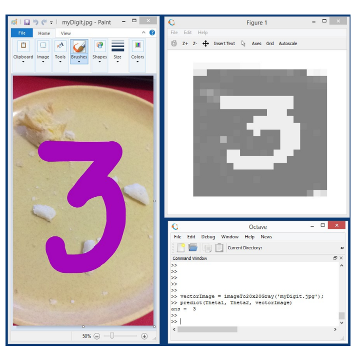
Digit 6



Digit 6 inverted is digit 9. This is the same photo of a six but rotated. Also, changed the contrast multiplier from 5 to 20. You can note that the gray background is smoother.



Digit 3



Explanation of Derivatives Used in Backpropagation

- We know that for a logistic regression classifier (which is what all of the output neurons in a neural network are), we use the cost function, $J(\theta) \quad y log(h_\theta(x)) \quad (1 \quad y) log(1 \quad h_\theta(x))$, and apply this over the K output neurons, and for all m examples.
- The equation to compute the partial derivatives of the theta terms in the output neurons:
- $\frac{\partial J(\theta)}{\partial \theta^{(L\ 1)}} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial \theta^{(L\ 1)}}$
- And the equation to compute partial derivatives of the theta terms in the [last] hidden layer neurons (layer L-1):
- $\frac{\partial J(\theta)}{\partial \theta^{(L\ 2)}} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L\ 1)}} \frac{\partial a^{(L\ 1)}}{\partial z^{(L\ 1)}} \frac{\partial z^{(L\ 1)}}{\partial \theta^{(L\ 2)}}$

- Clearly they share some pieces in common, so a delta term ($\delta^{(L)}$) can be used for the common pieces between the output layer and the hidden layer immediately before it (with the possibility that there could be many hidden layers if we wanted):
- $\delta^{(L)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$
- And we can go ahead and use another delta term ($\delta^{(L-1)}$) for the pieces that would be shared by the final hidden layer and a hidden layer before that, if we had one. Regardless, this delta term will still serve to make the math and implementation more concise.
- $\delta^{(L-1)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}$
- $\delta^{(L-1)} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}$
- With these delta terms, our equations become:
- $\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial \theta^{(L-1)}}$
- $\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = \delta^{(L-1)} \frac{\partial z^{(L-1)}}{\partial \theta^{(L-2)}}$
- Now, time to evaluate these derivatives:
- Let's start with the output layer:
- $\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial \theta^{(L-1)}}$
- Using $\delta^{(L)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$, we need to evaluate both partial derivatives.
- Given $J(\theta) = -ylog(a^{(L)}) - (1-y)log(1-a^{(L)})$, where $a^{(L)} = h_\theta(x))$, the partial derivative is:
- $\frac{\partial J(\theta)}{\partial a^{(L)}} = \frac{1-y}{1-a^{(L)}} - \frac{y}{a^{(L)}}$
- And given a=g(z), where $g = \frac{1}{1+e^{-z}}$, the partial derivative is:
- $\frac{\partial a^{(L)}}{\partial z^{(L)}} = a^{(L)}(1 - a^{(L)})$
- So, let's substitute these in for $\delta^{(L)}$:

$$\delta^{(L)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$$

$$\delta^{(L)} = (\frac{1-y}{1-a^{(L)}} - \frac{y}{a^{(L)}})(a^{(L)}(1 - a^{(L)}))$$

$$\delta^{(L)} = a^{(L)} - y$$

- So, for a 3-layer network (L=3),

$$\delta^{(3)} = a^{(3)} - y$$

- Note that this is the correct equation, as given in our notes.
- Now, given z=θ∗input, and in layer L the input is $a^{(L-1)}$, the partial derivative is:

$$\frac{\partial z^{(L)}}{\partial \theta^{(L-1)}} = a^{(L-1)}$$

- **Put it together for the output layer**:

$$\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial \theta^{(L-1)}}$$

$$\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = (a^{(L)} - y)(a^{(L-1)})$$

- Let's continue on for the hidden layer (let's assume we only have 1 hidden layer):

$$\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = \delta^{(L-1)} \frac{\partial z^{(L-1)}}{\partial \theta^{(L-2)}}$$

- Let's figure out $\delta(L-1)$.
- Once again, given z=θ∗input, the partial derivative is:

$$\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = \theta^{(L-1)}$$

- And: $\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} = a^{(L-1)}(1 - a^{(L-1)})$
- So, let's substitute these in for $\delta^{(L-1)}$:

$$\delta^{(L-1)} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}$$

$$\delta^{(L-1)} = \delta^{(L)}(\theta^{(L-1)})(a^{(L-1)}(1 - a^{(L-1)}))$$

$$\delta^{(L-1)} = \delta^{(L)}\theta^{(L-1)}a^{(L-1)}(1 - a^{(L-1)})$$

- So, for a 3-layer network,

$$\delta^{(2)} = \delta^{(3)}\theta^{(2)}a^{(2)}(1 - a^{(2)})$$

- **Put it together for the [last] hidden layer**:

$$\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = \delta^{(L-1)} \frac{\partial z^{(L-1)}}{\partial \theta^{(L-2)}}$$

$$\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = (\delta^{(L)} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}})(a^{(L-2)})$$

$$\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = ((a^{(L)} - y)(\theta^{(L-1)})(a^{(L-1)}(1 - a^{(L-1)})))(a^{(L-2)})$$

NN for linear systems

Introduction

The NN we created for classification can easily be modified to have a linear output. First solve the **4th** programming exercise. You can create a new function script, nnCostFunctionLinear.m, with the following characteristics

- There is only one output node, so you do not need the 'num_labels' parameter.
- Since there is one linear output, you do not need to convert y into a logical matrix.
- You still need a non-linear function in the hidden layer.
- The non-linear function is often the tanh() function - it has an output range from -1 to +1, and its gradient is easily implemented. Let g(z)=tanh(z).
- The gradient of tanh is $g'(z) = 1 - g(z)^2$. Use this in backpropagation in place of the sigmoid gradient.
- Remove the sigmoid function from the output layer (i.e. calculate a3 without using a sigmoid function), since we want a linear output.
- Cost computation: Use the linear cost function for J (from ex1 and ex5) for the unregularized portion. For the regularized portion, use the same method as ex**4**.
- Where reshape() is used to form the Theta matrices, replace 'num_labels' with '1'.

You still need to randomly initialize the Theta values, just as with any NN. You will want to experiment with different epsilon values. You will also need to create a predictLinear() function, using the tanh() function in the hidden layer, and a linear output.

Testing your linear NN

Here is a test case for your nnCostFunctionLinear()

```
% inputs
nn_params = [31 16 15 -29 -13 -8 -7 13 54 -17 -11 -9 16]'/ 10;
il = 1;
hl = 4;
X = [1; 2; 3];
y = [1; 4; 9];
lambda = 0.01;

% command
[j g] = nnCostFunctionLinear(nn_params, il, hl, X, y, lambda)

% results
j =  0.020815
g =
    -0.0131002
    -0.0110085
    -0.0070569
     0.0189212
    -0.0189639
    -0.0192539
    -0.0102291
     0.0344732
     0.0024947
     0.0080624
     0.0021964
     0.0031675
    -0.0064244

```

Now create a script that uses the 'ex5data1.mat' from ex5, but without creating the polynomial terms. With 8 units in the hidden layer and MaxIter set to 200, you should be able to get a final cost value of 0.3 to 0.4. The results will vary a bit due to the random Theta initialization. If you plot the training set and the predicted values for the training set (using your predictLinear() function), you should have a good match.

Deriving the Sigmoid Gradient Function

We let the sigmoid function be $\sigma(x) \quad \frac{1}{1+e^{-x}}$

Deriving the equation above yields to $(\frac{1}{1+e^{-x}})^2 \frac{d}{dx} \frac{1}{1+e^{-x}}$

Which is equal to $(\frac{1}{1-e^{-x}})^2 e^{\ x}(\ \ 1)$

$(\frac{1}{1-e^{-x}})(\frac{1}{1-e^{-x}})(\ \ e^{\ x})$

$(\frac{1}{1-e^{-x}})(\frac{-e^{\ x}}{1-e^{-x}})$

$\sigma(x)(1 \quad \sigma(x))$

Additional Resources for Backpropagation

- Very thorough conceptual [example] (https://web.archive.org/web/20150317210621/https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf)
- Short derivation of the backpropagation algorithm: http://pandamatak.com/people/anand/771/html/node37.html
- Stanford University Deep Learning notes: http://ufldl.stanford.edu/wiki/index.php/Backpropagation_Algorithm
- Very thorough explanation and proof: http://neuralnetworksanddeeplearning.com/chap2.html