Multi-class Classification

Introduction

In this exercise, you will implement one-vs-all logistic regression and neural networks to recognize hand-written digits.

Files included in this exercise

`ex3.m` - Octave script that will help step you through part 1
`ex3 nn.m` - Octave script that will help step you through part 2
`ex3data1.mat` - Training set of hand-written digits
`ex3weights.mat` - Initial weights for the neural network exercise
`displayData.m` - Function to help visualize the dataset
`fmincg.m` - Function minimization routine (similar to fminunc)
`sigmoid.m` - Sigmoid function
`lrCostFunction.m` - Logistic regression cost function
`oneVsAll.m` - Train a one-vs-all multi-class classifier
`predictOneVsAll.m` - Predict using a one-vs-all multi-class classifier
`predict.m` - Neural network prediction function

1 Multi-class Classification

For this exercise, you will use logistic regression and neural networks to recognize handwritten digits (from 0 to 9). Automated handwritten digit recognition is widely used today - from recognizing zip codes (postal codes) on mail envelopes to recognizing amounts written on bank checks. This exercise will show you how the methods you've learned can be used for this classification task.

In the first part of the exercise, you will extend your previous implemention of logistic regression and apply it to one-vs-all classification.

1.1 Dataset

You are given a data set in `ex3data1.mat` that contains 5000 training examples of handwritten digits.[1] The `.mat` format means that that the data has been saved in a native Octave/Matlab matrix format, instead of a text (ASCII) format like a csv-file. These matrices can be read directly into your program by using the `load` command. After loading, matrices of the correct dimensions and values will appear in your program's memory. The matrix will already be named, so you do not need to assign names to them.

```
% Load saved matrices from file
load('ex3data1.mat');
% The matrices X and y will now be in your Octave environment
```

There are 5000 training examples in `ex3data1.mat`, where each training example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The 20 by 20 grid of pixels is "unrolled" into a 400-dimensional vector. Each of these training examples become a single row in our data matrix X. This gives us a 5000 by 400 matrix `X` where every row is a training example for a handwritten digit image.

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}$$

The second part of the training set is a 5000-dimensional vector `y` that contains labels for the training set. To make things more compatible with Octave/Matlab indexing, where there is no zero index, we have mapped the digit zero to the value ten. Therefore, a "0" digit is labeled as "10", while the digits "1" to "9" are labeled as "1" to "9" in their natural order.

1.2 Visualizing the data

You will begin by visualizing a subset of the training set. In Part 1 of `ex3.m`, the code randomly selects selects 100 rows from `X` and passes those rows to the `displayData` function. This function maps each row to a 20 pixel by 20 pixel grayscale image and displays the images together. We have provided

[1] This is a subset of the MNIST handwritten digit dataset (http://yann.lecun.com/exdb/mnist/).

the `displayData` function, and you are encouraged to examine the code to see how it works. After you run this step, you should see an image like Figure 1.

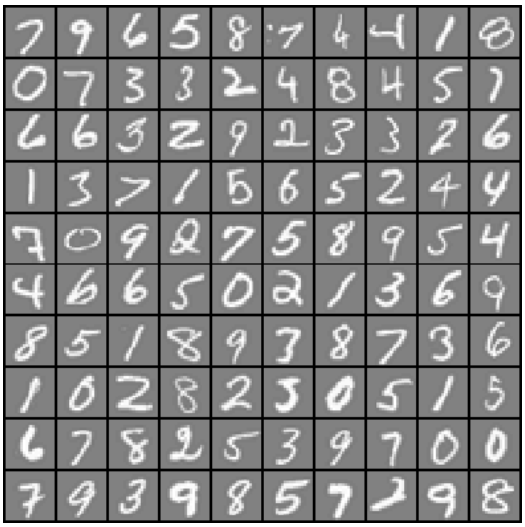

Figure 1: Examples from the dataset

1.3 Vectorizing Logistic Regression

You will be using multiple one-vs-all logistic regression models to build a multi-class classifier. Since there are 10 classes, you will need to train 10 separate logistic regression classifiers. To make this training efficient, it is important to ensure that your code is well vectorized. In this section, you will implement a vectorized version of logistic regression that does not employ any `for` loops. You can use your code in the last exercise as a starting point for this exercise.

1.3.1 Vectorizing the cost function

We will begin by writing a vectorized version of the cost function. Recall that in (unregularized) logistic regression, the cost function is

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left[-y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right].$$

To compute each element in the summation, we have to compute $h_\theta(x^{(i)})$ for every example i, where $h_\theta(x^{(i)}) = g(\theta^T x^{(i)})$ and $g(z) = \frac{1}{1+e^{-z}}$ is the

sigmoid function. It turns out that we can compute this quickly for all our examples by using matrix multiplication. Let us define X and θ as

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix} \quad \text{and} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}.$$

Then, by computing the matrix product $X\theta$, we have

$$X\theta = \begin{bmatrix} — (x^{(1)})^T\theta — \\ — (x^{(2)})^T\theta — \\ \vdots \\ — (x^{(m)})^T\theta — \end{bmatrix} = \begin{bmatrix} — \theta^T(x^{(1)}) — \\ — \theta^T(x^{(2)}) — \\ \vdots \\ — \theta^T(x^{(m)}) — \end{bmatrix}.$$

In the last equality, we used the fact that $a^Tb = b^Ta$ if a and b are vectors. This allows us to compute the products $\theta^T x^{(i)}$ for all our examples i in one line of code.

Your job is to write the unregularized cost function in the file `lrCostFunction.m` Your implementation should use the strategy we presented above to calculate $\theta^T x^{(i)}$. You should also use a vectorized approach for the rest of the cost function. A fully vectorized version of `lrCostFunction.m` should not contain any loops.

(Hint: You might want to use the element-wise multiplication operation (`.*`) and the sum operation `sum` when writing this function)

1.3.2 Vectorizing the gradient

Recall that the gradient of the (unregularized) logistic regression cost is a vector where the j^{th} element is defined as

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right).$$

To vectorize this operation over the dataset, we start by writing out all

the partial derivatives explicitly for all θ_j,

$$\begin{aligned}
\begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{bmatrix} &= \frac{1}{m} \begin{bmatrix} \sum_{i=1}^{m} \left((h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \right) \\ \sum_{i=1}^{m} \left((h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)} \right) \\ \sum_{i=1}^{m} \left((h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)} \right) \\ \vdots \\ \sum_{i=1}^{m} \left((h_\theta(x^{(i)}) - y^{(i)}) x_n^{(i)} \right) \end{bmatrix} \\
&= \frac{1}{m} \sum_{i=1}^{m} \left((h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} \right) \\
&= \frac{1}{m} X^T (h_\theta(x) - y). \qquad (1)
\end{aligned}$$

where

$$h_\theta(x) - y = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(1)}) - y^{(m)} \end{bmatrix}.$$

Note that $x^{(i)}$ is a vector, while $(h_\theta(x^{(i)}) - y^{(i)})$ is a scalar (single number). To understand the last step of the derivation, let $\beta_i = (h_\theta(x^{(i)}) - y^{(i)})$ and observe that:

$$\sum_i \beta_i x^{(i)} = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} = X^T \beta,$$

where the values $\beta_i = (h_\theta(x^{(i)}) - y^{(i)})$.

The expression above allows us to compute all the partial derivatives without any loops. If you are comfortable with linear algebra, we encourage you to work through the matrix multiplications above to convince yourself that the vectorized version does the same computations. You should now implement Equation 1 to compute the correct vectorized gradient. Once you are done, complete the function `lrCostFunction.m` by implementing the gradient.

Debugging Tip: Vectorizing code can sometimes be tricky. One common strategy for debugging is to print out the sizes of the matrices you are working with using the `size` function. For example, given a data matrix X of size 100×20 (100 examples, 20 features) and θ, a vector with dimensions 20×1, you can observe that $X\theta$ is a valid multiplication operation, while θX is not. Furthermore, if you have a non-vectorized version of your code, you can compare the output of your vectorized code and non-vectorized code to make sure that they produce the same outputs.

1.3.3 Vectorizing regularized logistic regression

After you have implemented vectorization for logistic regression, you will now add regularization to the cost function. Recall that for regularized logistic regression, the cost function is defined as

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

Note that you should *not* be regularizing θ_0 which is used for the bias term.

Correspondingly, the partial derivative of regularized logistic regression cost for θ_j is defined as

$$\frac{\partial J}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad \text{for j} = 0$$

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\left(\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \lambda\theta_j\right) \qquad \text{for j} \geq 1.$$

Now modify your code in `lrCostFunction` to account for regularization. Once again, you should not put any loops into your code.

Tip: When implementing the vectorization for regularized lo-gistic regression, you might often want to only sum and update certain elements of θ. In Octave, you can index into the matrices to access and update only certain elements. For example, `A(:, 3:5) = B(:, 1:3)` will replaces the columns 3 to 5 of A with the columns 1 to 3 from B. One special keyword you can use in indexing is the `end` keyword in indexing. This allows us to select columns (or rows) until the end of the matrix. For example, `A(:, 2:end)` will only return elements from the 2^{nd} to last column of A. Thus, you could use this together with the `sum` and `.^` op-erations to compute the sum of only the elements you are interested in (e.g., `sum(z(2:end).^2)`). In the starter code, `lrCostFunction.m`, we have also provided hints on yet *another* possible method computing the regularized gradient.

1.4 One-vs-all Classification

In this part of the exercise, you will implement one-vs-all classification by training multiple regularized logistic regression classifiers, one for each of the K classes in our dataset (Figure 1). In the handwritten digits dataset, $K = 10$, but your code should work for any value of K.

You should now complete the code in `oneVsAll.m` to train one classifier for each class. In particular, your code should return all the classifier parameters in a matrix $\Theta \in \mathbb{R}^{K \times (N+1)}$, where each row of Θ corresponds to the learned logistic regression parameters for one class. You can do this with a "for"-loop from 1 to K, training each classifier independently.

Note that the `y` argument to this function is a vector of labels from 1 to 10, where we have mapped the digit "0" to the label 10 (to avoid confusions with indexing).

When training the classifier for class $k \in \{1, ..., K\}$, you will want a m-dimensional vector of labels y, where $y_j \in 0, 1$ indicates whether the j-th training instance belongs to class k ($y_j = 1$), or if it belongs to a different class ($y_j = 0$). You may find logical arrays helpful for this task.

Octave Tip: Logical arrays in Octave are arrays which contain binary (0 or 1) elements. In Octave, evaluating the expression `a == b` for a vector `a` (of size $m \times 1$) and scalar `b` will return a vector of the same size as `a` with ones at positions where the elements of `a` are equal to b and zeroes where they are different. To see how this works for yourself, try the following code in Octave:

```
a = 1:10; % Create a and b
b = 3;
a == b    % You should try different values of b here
```

Furthermore, you will be using `fmincg` for this exercise (instead of `fminunc`). `fmincg` works similarly to `fminunc`, but is more more efficient for dealing with a large number of parameters.

After you have correctly completed the code for `oneVsAll.m`, the script `ex3.m` will continue to use your `oneVsAll` function to train a multi-class classifier.

1.4.1 One-vs-all Prediction

After training your one-vs-all classifier, you can now use it to predict the digit contained in a given image. For each input, you should compute the "probability" that it belongs to each class using the trained logistic regression classifiers. Your one-vs-all prediction function will pick the class for which the corresponding logistic regression classifier outputs the highest probability and return the class label (1, 2,..., or K) as the prediction for the input example.

You should now complete the code in `predictOneVsAll.m` to use the one-vs-all classifier to make predictions.

Once you are done, `ex3.m` will call your `predictOneVsAll` function using the learned value of Θ. You should see that the training set accuracy is about 94.9% (i.e., it classifies 94.9% of the examples in the training set correctly).

References:
Programming exercises by Andrew N.g., Machine Learning.