Multi-class Classification and Neural Networks

Yunlong Pan 113061415

2020.11.26

1.Introduction

In this project, I will implement one-vs-all logistic regression and the backpropagation algorithm for neural networks to recognize hand-written digits.

2.Data Set

There are 5000 training examples in data1.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 becomes 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. The second part of the training set is a 5000-dimensional vector y that contains labels for the training set.

In Part 1 of project.m, the code randomly 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.

3.Multi-class Classification

In the first part, I will use the logistic regression and apply it to one-vs-all classification.For regularized logistic regression, the cost function is defined as

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

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

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}((h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}).$$

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

I write these functions in lrCostFunction.m file.

In this part, I will implement one-vs-all classification by training multiple regularized logistic regression classifiers, one for each of the K classes in our dataset. In the handwritten digits dataset, $K = 10$. The code in oneVsAll.m is to train one classifier for each class. In particular,the code 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.

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. When training the classifier for class $k \in \{1, ..., K\}$, I take 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 $(y_j = 0)$. The script project.m will continue to use oneVsAll function to train a multi-class classifier.

After training the one-vs-all classifier, we now use it to predict the digit contained in a given image. For each input, we compute the “probability” that it belongs to each class using the trained logistic regression classifiers. The 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.

The code in predictOneVsAll.m is to use the one-vs-all classifier to make predictions. project.m will call predictOneVsAll function using the learned value of Θ. The training set accuracy is about 94.9%.

4.Backpropagation for Neural Networks

Our neural network has 3 layers - an input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digit images. Since the images are of size 20 × 20, this gives us 400 input layer units (not counting the extra bias unit which always outputs +1). The training data will be loaded into the variables X and y by the project.m script.

In this part, the backpropagation algorithm is to compute the gradient for the neural network cost function. The nnCostFunction.m returns an appropriate value for grad. Using the gradient, the neural network will be trainned by minimizing the cost function $J(\Theta)$ using an advanced optimizer fmincg.

When training neural networks, it is important to randomly initialize the parameters for symmetry breaking. One $\mathrm{eff\,ective}$ strategy for random initialization is to randomly select values for $\Theta^{(l)}$ uniformly in the range $[-\epsilon_{init}, \epsilon_{init}]$. I just use $\epsilon_{init} = 0.12$ .This range of values ensures that the parameters are kept small and makes the learning more efficient.

Recall that the intuition behind the backpropagation algorithm is as follows. Given a training example $(x^{(t)}, y^{(t)})$, we will first run a “forward pass” to compute all the activations throughout the network, including the output value of the hypothesis $h_\Theta(x)$. Then, for each node *j* in layer *l*, we would like to compute an “error term” $\delta_j^{(l)}$ that measures how much that node was “responsible” for any errors in our output.

For an output node, we can directly measure the difference between the network’s activation and the true target value, and use that to define $\delta_j^{(3)}$ (since layer 3 is the output layer). For the hidden units, you will compute $\delta_j^{(l)}$ based on a weighted average of the error terms of the nodes in layer $(l + 1)$.

In detail, here is the backpropagation algorithm. The steps 1 to 4 in a loop that processes one example at a time. Concretely, you should implement a for-loop for $\mathrm{t = 1{:}m}$ and place steps 1-4 below inside the for-loop, with the t^{th} iteration performing the calculation on the t th training example $(x^{(t)}, y^{(t)})$. Step 5 will divide the accumulated gradients by m to obtain the gradients for the neural network cost function.

1. Set the input layer’s values $(a^{(1)})$ to the t-th training example $x^{(t)}$.Perform a feedforward pass, computing the activations $(z^{(2)}, a^{(2)}, z^{(3)}, a^{(3)})$ for layers 2 and 3.

2. For each output unit k in layer 3, set

$$\delta_k^{(3)} = (a_k^{(3)} - y_k),$$

where $y_k \in \{0,1\}$ indicates whether the current training example belongs to class $k(y_k = 1)$, or if it belongs to a different class $(y_k = 0)$

3. For the hidden layer $l = 2$, set

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

4. Accumulate the gradient from this example using the following formula.

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

5. Obtain the (unregularized) gradient for the neural network cost function by dividing the accumulated gradients by $\frac{1}{m}$:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

After the backpropagation algorithm, the script project.m will proceed to run gradient checking. The gradient check will allow to increase the confidence that the code is computing the gradients correctly.

After computing $\Delta_{ij}^{(l)}$ by vackpropagation, we add regularization

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}, \text{for j=0}$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \frac{\lambda}{m} \Theta_{ij}^{(l)} \text{for j} \geq 1$$

The code computing grad is in *nnCostFunction.m* to account for regularization. After the training completes, the project.m script will proceed to report the training accuracy of the classifier by computing the percentage of examples it got correct. A reported training accuracy of is about 95.3%. It is possible to get higher training accuracies by training the neural network for more iterations.

5.Conclusions

```
num_labels = 10;
load('data1.mat');
m = size(X, 1); rand_indices = randperm(m);
theta_t = [-2; -1; 1; 2];
X_t = [ones(5,1) reshape(1:15,5,3)/10];
y_t = ([1;0;1;0;1] >= 0.5);
lambda_t = 3;
[J, grad] = lrCostFunction(theta_t, X_t, y_t, lambda_t);
lambda = 0.1;
[all_theta] = oneVsAll(X, y, num_labels, lambda);
pred = predictOneVsAll(all_theta, X);
y((y==10)) = 0;
fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);
```

```
Training Set Accuracy: 94.980000
```

```
clear ; close all; clc
```

```
input_layer_size  = 400;
hidden_layer_size = 25;
num_labels = 10;
load('data1.mat');
m = size(X, 1);
sel = randperm(size(X, 1));
sel = sel(1:100);
load('weights.mat');
nn_params = [Theta1(:) ; Theta2(:)];
g = sigmoidGradient([-1 -0.5 0 0.5 1]);
initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size);
initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels);
initial_nn_params = [initial_Theta1(:) ; initial_Theta2(:)];
lambda = 3;
checkNNGradients(lambda);
```

```
Relative Difference: 2.31407e-11
```

```
debug_J  = nnCostFunction(nn_params, input_layer_size, ...
                          hidden_layer_size, num_labels, X, y, lambda);
options = optimset('MaxIter', 50);
lambda = 1;
costFunction = @(p) nnCostFunction(p, ...
                                   input_layer_size, ...
                                   hidden_layer_size, ...
                                   num_labels, X, y, lambda);
[nn_params, cost] = fmincg(costFunction, initial_nn_params, options);
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
                 hidden_layer_size, (input_layer_size + 1));

Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end), ...
                 num_labels, (hidden_layer_size + 1));

pred = predict(Theta1, Theta2, X);

fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);
```

```
Training Set Accuracy: 95.260000
```

Reference

- The MNIST handwritten digit dataset http://yann.lecun.com/exdb/mnist/