

## MACHINE LEARNING

# Machine Learning Exercises In Python, Part 5

30TH MAY 2016

---

*This post is part of a series covering the exercises from Andrew Ng's **machine learning** class on Coursera. The original code, exercise text, and data files for this post are available **here**.*

***Part 1 - Simple Linear Regression***

***Part 2 - Multivariate Linear Regression***

***Part 3 - Logistic Regression***

***Part 4 - Multivariate Logistic Regression***

***Part 5 - Neural Networks***

***Part 6 - Support Vector Machines***

***Part 7 - K-Means Clustering & PCA***

***Part 8 - Anomaly Detection & Recommendation***

In part four we wrapped up our implementation of logistic regression by extending our solution to handle multi-class classification and testing it on the hand-written digits data set. Using just logistic regression we were able to hit a classification accuracy of about 97.5%, which is reasonably good but pretty much maxes out what we can achieve with a linear model. In this blog post we'll again tackle the hand-written digits data set, but this time using a feed-forward neural network with backpropagation. We'll implement un-regularized and regularized versions of the neural network cost function and compute gradients via the

backpropagation algorithm. Finally, we'll run the algorithm through an optimizer and evaluate the performance of the network on the handwritten digits data set



I'll note up front that the math (and code) in this exercise gets a bit hairy. Implementing a neural network from scratch is not for the faint of heart. For those that venture on, be warned - here be dragons. I also strongly encourage the reader to skim through the corresponding exercise text from Andrew Ng's class. I uploaded a copy [here](#). This text contains all of the equations that we'll be implementing. With that, let's dive in!

Since the data set is the same one we used in the last exercise, we'll re-use the code from last time to load the data.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat
%matplotlib inline

data = loadmat('data/ex3data1.mat')
data
```

```
{'X': array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
             [ 0.,  0.,  0., ...,  0.,  0.,  0.],
             [ 0.,  0.,  0., ...,  0.,  0.,  0.],
             ...,
             [ 0.,  0.,  0., ...,  0.,  0.,  0.],
             [ 0.,  0.,  0., ...,  0.,  0.,  0.],
             [ 0.,  0.,  0., ...,  0.,  0.,  0.])),
 '__globals__': [],
 '__header__': 'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Sun Oct 16 1
 '__version__': '1.0',
 'y': array([[10],
             [10],
             [10],
             ...,
             [ 9],
             [ 0]
```

```
[ 9],  
[ 9]], dtype=uint8)}
```



Since we're going to need these later (and will use them often), let's create some useful variables up-front.

```
X = data['X']  
y = data['y']  
X.shape, y.shape  
  
((5000L, 400L), (5000L, 1L))
```

We're also going to need to one-hot encode our labels. One-hot encoding turns a class label  $n$  (out of  $k$  classes) into a vector of length  $k$  where index  $n$  is "hot" (1) while the rest are zero. Scikit-learn has a built in utility we can use for this.

```
from sklearn.preprocessing import OneHotEncoder  
encoder = OneHotEncoder(sparse=False)  
y_onehot = encoder.fit_transform(y)  
y_onehot.shape  
  
(5000L, 10L)
```

```
y[0], y_onehot[0,:]  
  
(array([10], dtype=uint8),  
 array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.])))
```

The neural network we're going to build for this exercise has an input layer matching the size of our instance data (400 + the bias unit), a hidden layer with 25 units (26 with the bias unit) and an output layer with 10 units corresponding to our one-hot encoding for the class labels. The first piece we need to implement is a cost function to evaluate the loss for a given set of network parameters. The source mathematical function is in the exercise text, and looks pretty intimidating, but it helps to really break it down into pieces. Here are the functions required to compute the cost.

```
def sigmoid(z):  
    return 1 / (1 + np.exp(-z))
```

```
def forward_propagate(X, theta1, theta2):  
    m = X.shape[0]  
  
    a1 = np.insert(X, 0, values=np.ones(m), axis=1)  
    z2 = a1 * theta1.T  
    a2 = np.insert(sigmoid(z2), 0, values=np.ones(m), axis=1)  
    z3 = a2 * theta2.T  
    h = sigmoid(z3)  
  
    return a1, z2, a2, z3, h
```

```
def cost(params, input_size, hidden_size, num_labels, X, y, learning_rate):  
    m = X.shape[0]  
    X = np.matrix(X)  
    y = np.matrix(y)  
  
    # reshape the parameter array into parameter matrices for each layer  
    theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidd
```

```

theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (n
theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (n

# run the feed-forward pass
a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

# compute the cost
J = 0
for i in range(m):
    first_term = np.multiply(-y[i,:], np.log(h[i,:]))
    second_term = np.multiply((1 - y[i,:]), np.log(1 - h[i,:]))
    J += np.sum(first_term - second_term)

J = J / m

return J

```

We've used the sigmoid function before so that's not new. The forward-propagate function computes the hypothesis for each training instance given the current parameters (in other words, given some current state of the network and a set of inputs, it calculates the outputs at each layer in the network). The shape of the hypothesis vector (denoted by  $\mathbf{h}$ ), which contains the prediction probabilities for each class, should match our one-hot encoding for  $\mathbf{y}$ . Finally, the cost function runs the forward-propagation step and calculates the error of the hypothesis (predictions) vs. the true label for the instance.

We can test this real quick to convince ourselves that it's working as expected. Seeing the output from intermediate steps is also useful to understand what's going on.

```

# initial setup
input_size = 400

```



```
hidden_size = 25
num_labels = 10
learning_rate = 1

# randomly initialize a parameter array of the size of the full network's parameters
params = (np.random.random(size=hidden_size * (input_size + 1) + num_labels * (h

m = X.shape[0]
X = np.matrix(X)
y = np.matrix(y)

# unravel the parameter array into parameter matrices for each layer
theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidden_s
theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (num_labe

theta1.shape, theta2.shape

((25L, 401L), (10L, 26L))
```

```
a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)
a1.shape, z2.shape, a2.shape, z3.shape, h.shape

((5000L, 401L), (5000L, 25L), (5000L, 26L), (5000L, 10L), (5000L, 10L))
```

The cost function, after computing the hypothesis matrix  $h$ , applies the cost equation to compute the total error between  $y$  and  $h$ .

```
cost(params, input_size, hidden_size, num_labels, X, y_onehot, learning_rate)
```

6.8228086634127862



Our next step is to add regularization to the cost function, which adds a penalty term to the cost that scales with the magnitude of the parameters. The equation for this looks pretty ugly, but it can be boiled down to just one line of code added to the original cost function. Just add the following right before the return statement.

```
J += (float(learning_rate) / (2 * m)) * (np.sum(np.power(theta1[:,1:], 2)) + np.
```

Next up is the backpropagation algorithm. Backpropagation computes the parameter updates that will reduce the error of the network on the training data. The first thing we need is a function that computes the gradient of the sigmoid function we created earlier.

```
def sigmoid_gradient(z):  
    return np.multiply(sigmoid(z), (1 - sigmoid(z)))
```

Now we're ready to implement backpropagation to compute the gradients. Since the computations required for backpropagation are a superset of those required in the cost function, we're actually going to extend the cost function to also perform backpropagation and return both the cost and the gradients. If you're wondering why I'm not just calling the existing cost function from within the backprop function to make the design more modular, it's because backprop uses a number of other variables calculated inside the cost function. Here's the full implementation. I skipped ahead and added gradient regularization rather than first create an un-regularized version.

```
def backprop(params, input_size, hidden_size, num_labels, X, y, learning_rate):
```

```
##### this section is identical to the cost function logic we already saw
```

```
m = X.shape[0]
```

```
X = np.matrix(X)
```

```
y = np.matrix(y)
```

```
# reshape the parameter array into parameter matrices for each layer
```

```
theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidden_size, input_size + 1)))
```

```
theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (num_hidden_size, hidden_size)))
```

```
# run the feed-forward pass
```

```
a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)
```

```
# initializations
```

```
J = 0
```

```
delta1 = np.zeros(theta1.shape) # (25, 401)
```

```
delta2 = np.zeros(theta2.shape) # (10, 26)
```

```
# compute the cost
```

```
for i in range(m):
```

```
    first_term = np.multiply(-y[i,:], np.log(h[i,:]))
```

```
    second_term = np.multiply((1 - y[i,:]), np.log(1 - h[i,:]))
```

```
    J += np.sum(first_term - second_term)
```

```
J = J / m
```

```
# add the cost regularization term
```

```
J += (float(learning_rate) / (2 * m)) * (np.sum(np.power(theta1[:,1:], 2)) +
```


```
##### end of cost function logic, below is the new part #####
```

```
# perform backpropagation
```

```
for t in range(m):
```

```
    a1t = a1[t,:] # (1, 401)
```





```

a1t = a1[t,:] # (1, 401)
z2t = z2[t,:] # (1, 25)
a2t = a2[t,:] # (1, 26)
ht = h[t,:] # (1, 10)
yt = y[t,:] # (1, 10)

d3t = ht - yt # (1, 10)

z2t = np.insert(z2t, 0, values=np.ones(1)) # (1, 26)
d2t = np.multiply((theta2.T * d3t.T).T, sigmoid_gradient(z2t)) # (1, 26)

delta1 = delta1 + (d2t[:,1:]).T * a1t
delta2 = delta2 + d3t.T * a2t

delta1 = delta1 / m
delta2 = delta2 / m

# add the gradient regularization term
delta1[:,1:] = delta1[:,1:] + (theta1[:,1:] * learning_rate) / m
delta2[:,1:] = delta2[:,1:] + (theta2[:,1:] * learning_rate) / m

# unravel the gradient matrices into a single array
grad = np.concatenate((np.ravel(delta1), np.ravel(delta2)))

return J, grad

```

There's a lot going on here so let's try to unpack it a bit. The first half of the function is calculating the error by running the data plus current parameters through the "network" (the forward-propagate function) and comparing the output to the true labels. The total error across the whole data set is represented as  $J$ . This is the part we discussed earlier as the cost function.



The rest of the function is essentially answering the question "how can I adjust my parameters to reduce the error the next time I run through the network"? It does this by computing the contributions at each layer to the total error and adjusting appropriately by coming up with a "gradient" matrix (or, how much to change each parameter and in what direction).

The hardest part of the backprop computation (other than understanding WHY we're doing all these calculations) is getting the matrix dimensions right, which is why I annotated some of the calculations with comments showing the resulting shape of the computation. By the way, if you find it confusing when to use  $A * B$  vs. `np.multiply(A, B)`, you're not alone. Basically the former is a matrix multiplication and the latter is an element-wise multiplication (unless A or B is a scalar value, in which case it doesn't matter). I wish there was a more concise syntax for this (maybe there is and I'm just not aware of it).

Anyway, let's test it out to make sure the function returns what we're expecting it to.

```
J, grad = backprop(params, input_size, hidden_size, num_labels, X, y_onehot, lea
J, grad.shape

(6.8281541822949299, (10285L,))
```

We're finally ready to train our network and use it to make predictions. This is roughly similar to the previous exercise with multi-class logistic regression.

```
from scipy.optimize import minimize

# minimize the objective function
fmin = minimize(fun=backprop, x0=params, args=(input_size, hidden_size, num_labe
```



◀
▶

```
[...],  
[ 9],  
[ 9],  
[ 9]], dtype=int64)
```



Finally we can compute the accuracy to see how well our trained network is doing.

```
correct = [1 if a == b else 0 for (a, b) in zip(y_pred, y)]  
accuracy = (sum(map(int, correct)) / float(len(correct)))  
print 'accuracy = {0}%'.format(accuracy * 100)  
  
accuracy = 99.22%
```

And we're done! We've successfully implemented a rudimentary feed-forward neural network with backpropagation and used it to classify images of handwritten digits. It might seem surprising at first that we managed to do this without implementing any classes or data structures resembling a network in any way. Isn't that what neural networks are all about? This was one of the biggest surprises to me when I took the class - how the whole thing basically boils down to a series of matrix multiplications. It turns out that this is by far the most efficient way to solve the problem. In fact if you look at any of the popular deep learning frameworks such as Tensorflow, they're essentially graphs of linear algebra computations. It's a very useful and practical way to think about machine learning algorithms.

That concludes the exercise on neural networks. In the next exercise we'll look at another popular supervised learning algorithm, support vector machines.

---

Follow me on twitter to get new post updates.

Follow @jdwittenauer