Deep Learning

Implementation of
Linear Regression and Logistic Regression in Python and TensorFlow
- the assignments -

Exercise 1

Linear Regression

(without Tensorflow)

```
Programming Exercise 1: Linear Regression (without tensorflow)
==============================================================

ACKNOWLEDGMENT: This exercise is based on an exercise in Andrew Ng's course.

In this exercise, you will write the code for the following:
    the PlotData function
    code to process the data
    the ComputeCost function
    the GradientDescent function
```

In [ ]:

```python
# Run this cell to import all the libraries you need for this exercise
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
```

```
======== STEP 1: Read in data points and plot the data ==========
```

In [ ]:

```python
# Let's just make sure you know how produce a scatter plot of a few points first

def PlotData(x, y, xlabel, ylabel):
    #
    # INSTRUCTIONS:
    # Write the code for PlotData(x,y) which is a function
    # that plots the data points x and y in a scatter plot
    # with axes labels xlabel and ylabel.
    #
    # ====================== YOUR CODE HERE ======================
    #




    # ============================================================
    return

PlotData([1,2,3,4], [1,4,9,16], 'Population of City in 10,000s', 'Profit in $10,000s')
```

```
Programming Exercise 1: Linear Regression (without tensorflow)
==============================================================

ACKNOWLEDGMENT: This exercise is based on an exercise in Andrew Ng's course.

In this exercise, you will write the code for the following:
    the PlotData function
    code to process the data
    the ComputeCost function
    the GradientDescent function
```

In [ ]:
```python
# Run this cell to import all the libraries you need for this exercise
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
```

```
======== STEP 1: Read in data points and plot the data ==========
```

In [ ]:
```python
# Let's just make sure you know how produce a scatter plot of a few points first

def PlotData(x, y, xlabel, ylab
    #
    # INSTRUCTIONS:
    # Write the code for PlotDa
    # that plots the data point
    # with axes labels xlabel a
    #
    # ===================== YOUR CODE HERE =====================
    #



    # ============================================================
    return

PlotData([1,2,3,4], [1,4,9,16], 'Population of City in 10,000s', 'Profit in $10,000s')
```

Learn about and use:
matplotlib.pyplot.plot, matplotlib.pyplot.xlabel, matplotlib.pyplot.ylabel, matplotlib.pyplot.show

Secure | https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html

matplotlib

matplotlib.pyplot.plot

`matplotlib.pyplot.plot(*args, **kwargs)`

Plot lines and/or markers to the `Axes`. *args* is a variable length argument, allowing for multiple *x*, *y* pairs with an optional format string. For example, each of the following is legal:

```
plot(x, y)        # plot x and y using default line style and color
plot(x, y, 'bo')  # plot x and y using blue circle markers
plot(y)           # plot y using x as index array 0..N-1
plot(y, 'r+')     # ditto, but with red plusses
```

If *x* and/or *y* is 2-dimensional, then the corresponding columns will be plotted.

If used with labeled data, make sure that the color spec is not included as an element in data, as otherwise the last case `plot("v","r", data={"v":..., "r":...)` can be interpreted as the first case which would do `plot(v, r)` using the default line style and color.

If not used with labeled data (i.e., without a data argument), an arbitrary number of *x*, *y*, *fmt* groups can be specified, as in:

```
a.plot(x1, y1, 'g^', x2, y2, 'g-')
```

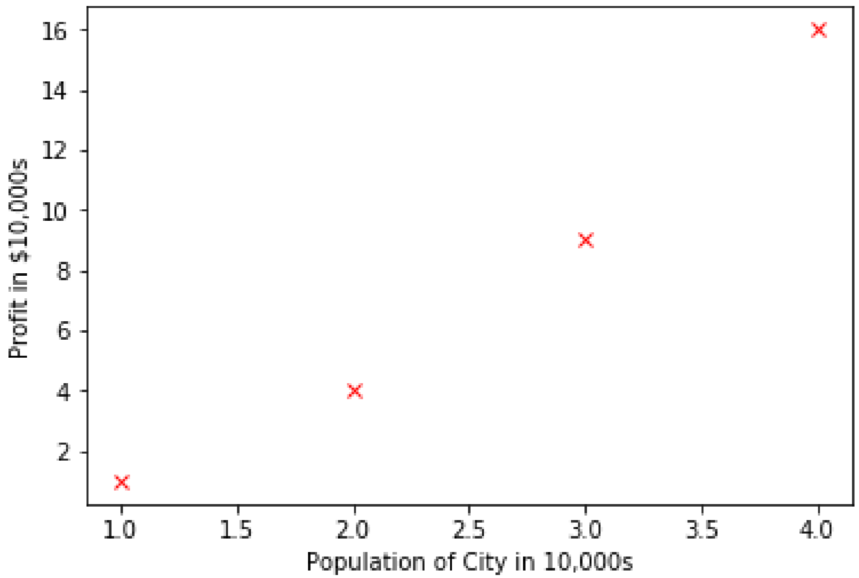
Return value is a list of lines that were added.

By default, each line is assigned a different style specified by a 'style cycle'. To change this behavior, you can edit the axes.prop_cycle rcParam.

The following format string characters are accepted to control the line style or marker:

character	description
`'-'`	solid line style
`'--'`	dashed line style

```
PlotData([1,2,3,4], [1,4,9,16], 'Population of City in 10,000s', 'Profit in $10,000s')
```

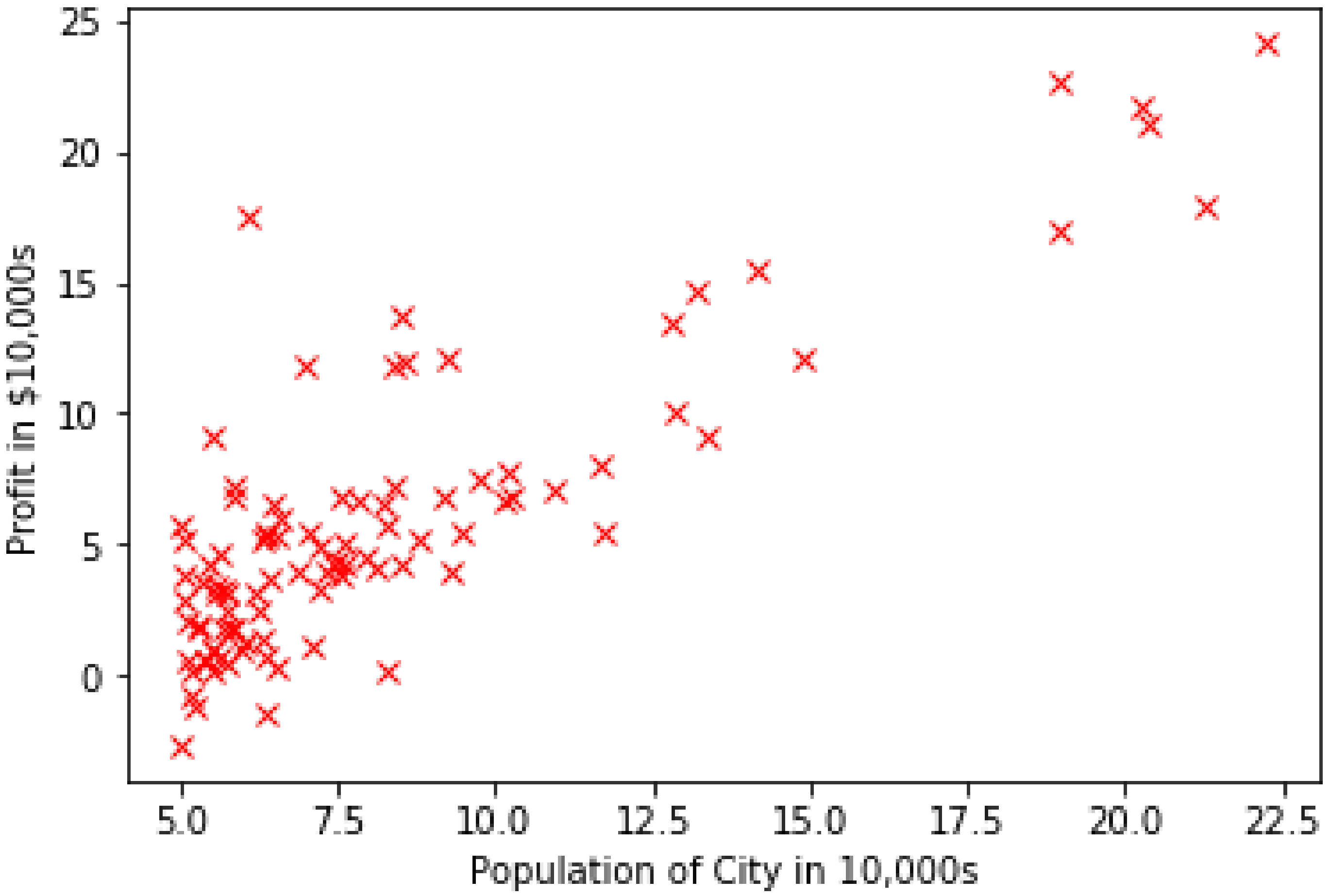


In [ ]:

```
# Let's read in the data from 'ex1data1.txt'
f = open('ex1data1.txt', 'r')

x_list = []
y_list = []

# INSTRUCTIONS:
# Write code to process the data. Each line of the file contains
# two numbers x and y separated by a comma. Put all the x's
# into the x list and the y's into the y list,
# which will later be passed to your function PlotData
# ===================== YOUR CODE HERE ======================
#




# ==============================================================

# Let's see what the data looks like
x = np.array(x_list).reshape(len(x_list),1)
y = np.array(y_list).reshape(len(y_list),1)
PlotData(x, y, 'Population of City in 10,000s', 'Profit in $10,000s')
f.close()
```

```python
# Let's read in the data from 'ex1data1.txt'
f = open('ex1data1.txt', 'r')

x_list = []
y_list = []

# INSTRUCTIONS:
# Write code to process the data. Each line of the file contains
# two numbers x and y separated by a comma. Put all the x's
# into the x list and the y's into the y list,
# which will later be passed to your function PlotData
# ====================== YOUR CODE HERE ======================
#

for line in f:
        x_str, y_str = line.split(',')
        x_list.append(float(x_str))
        y_list.append(float(y_str))

# ============================================================

# Let's see what the data looks like
x = np.array(x_list).reshape(len(x_list),1)
y = np.array(y_list).reshape(len(y_list),1)
PlotData(x, y, 'Population of City in 10,000s', 'Profit in $10,000s')
f.close()
```

Data (one variable): $h_\theta(x) = \theta_0 + \theta_1 x$



```
======== STEP 2: Write the ComputeCost function ==========
```

```
# Before we can apply gradient descent, we need to have in place a cost function

# The code below adds a column of ones to x and initializes theta to all zeros
X = np.c_[np.ones(len(x)),x]
theta = np.zeros((2, 1))

def ComputeCost(X, y, theta):
    #
    # INSTRUCTIONS:
    # Write the code for ComputeCost, which is a function that computes
    # the cost of using theta as the parameter of the linear regression
    # J = COMPUTECOST(X, y, theta) computes the cost of using theta as the
    # parameter for linear regression to fit the data points in x and y.
    #
    # Let m = number of data points (or training examples)
    # Assume that:
    #     x, y have shape (m,1).
    #     X, being x with an additional column of ones, has shape (m,2).
    #     theta has shape (2,1).

    m = len(y)

    # ====================== YOUR CODE HERE ======================
    #

    # ============================================================
    return J

# compute and display initial cost
print(ComputeCost(X, y, theta))
```

```
32.0727338775
```

Cost Function for Gradient Descent
(one variable)

Mean-square error:

$$J(\theta_0, \theta_1) = \frac{1}{M}\sum_{i=1}^{M}(h_\theta(x_i) - y_i)^2$$

Half mean-square error :

$$J(\theta_0, \theta_1) = \frac{1}{2M}\sum_{i=1}^{M}(h_\theta(x_i) - y_i)^2$$

```
======== STEP 2: Write the ComputeCost function ==========
```

```python
# Before we can apply gradient descent, we need to have in place a cost function

# The code below adds a column of ones to x and initializes theta to all zeros
X = np.c_[np.ones(len(x)),x]
theta = np.zeros((2, 1))

def ComputeCost(X, y, theta):
    #
    # INSTRUCTIONS:
    # Write the code for ComputeCost, which is a function that computes
    # the cost of using theta as the parameter of the linear regression
    # J = COMPUTECOST(X, y, theta) computes the cost of using theta as the
    # parameter for linear regression to fit the data points in x and y.
    #
    # Let m = number of data points (or training examples)
    # Assume that:
    #    x, y have shape (m,1).
    #    X, being x with an additional column of ones, has shape (m,2).
    #    theta has shape (2,1).

    m = len(y)

    # ====================== YOUR CODE HERE ======================
    #


    # ============================================================
    return J

# compute and display initial cost
print(ComputeCost(X, y, theta))
```

```
32.0727338775
```

Half mean-square error :

$$J(\theta_0, \theta_1) = \frac{1}{2M}\sum_{i=1}^{M}(h_\theta(x_i) - y_i)^2$$

```python
# Now let's implement the Gradient Descent function

def GradientDescent(X, y, theta, alpha, num_iters):
    #
    # INSTRUCTIONS:
    # Write the code for function gradientDescent(X, y, theta, alpha, num_iters)
    # which performs gradient descent to learn and return theta.
    # It updates theta by taking num_iters gradient steps with learning rate alpha.
    #

    # Initialize m = no. of training examples
    m = len(y)
    J_history = np.zeros((num_iters, 1))

    for iter in range(0, num_iters):
        #
        # ===================== YOUR CODE HERE ======================
        #




        # ============================================================
        # Save the cost J in every iteration
        J_history[iter] = ComputeCost(X, y, theta)

    # Let's just print the first 10 Js to check that the cost is decreasing nicely
    print(J_history[0:10])
    return theta

# Run gradient descent
theta = np.zeros((2, 1))
alpha = 0.01
iterations = 1500
theta = GradientDescent(X, y, theta, alpha, iterations)
```

Taking derivatives (one variable)

$J(\theta_0, \theta_1) = \frac{1}{2M}\sum_{i=1}^{M}(h_\theta(x_i) - y_i)^2$ and $h_\theta(x_i) = \theta_0 + \theta_1 x_i$

$$\theta_0 \leftarrow \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$

$$\theta_1 \leftarrow \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

Repeat until convergence:

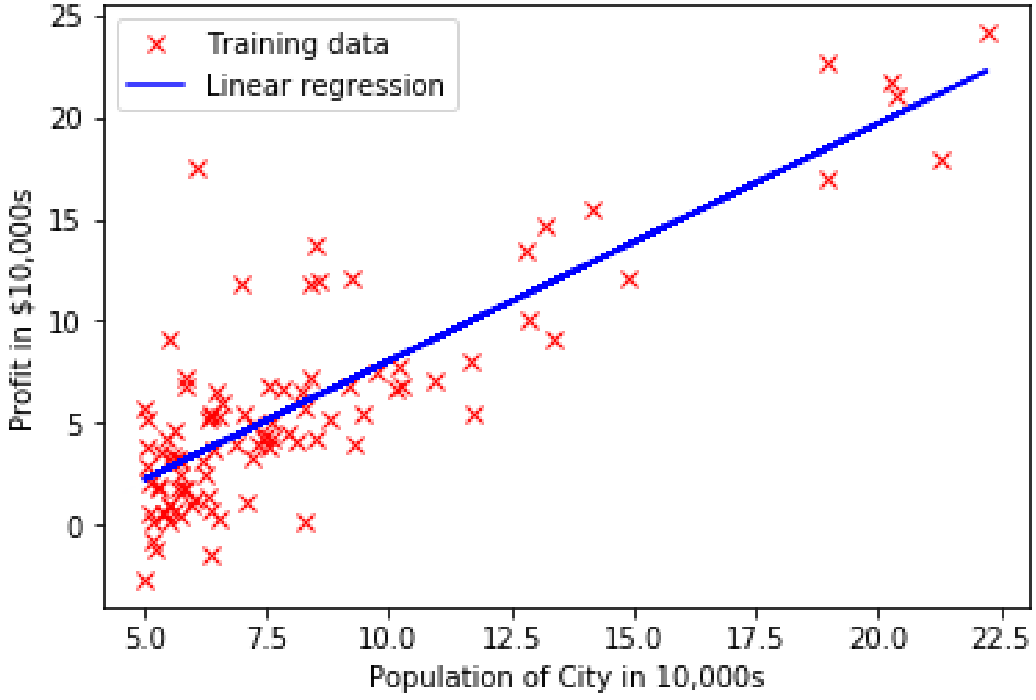
$$\theta_0 \leftarrow \theta_0 - \alpha \frac{1}{M}\sum_{i=1}^{M}(h_\theta(x_i) - y_i)$$

$$\theta_1 \leftarrow \theta_1 - \alpha \frac{1}{M}\sum_{i=1}^{M}(h_\theta(x_i) - y_i)\, x_i$$

```
# Now let's implement the Gradient Descent function

def GradientDescent(X, y, theta, alpha, num_iters):
    #
    # INSTRUCTIONS:
    # Write the code for function gradientDescent(X, y, theta, alpha, num_iters)
    # which performs gradient descent to learn and return theta.
    # It updates theta by taking num_iters gradient steps with learning rate alpha.
    #

    # Initialize m = no. of training examples
    m = len(y)
    J_history = np.zeros((num_iters, 1))

    for iter in range(0, num_iters):
        #
        # ====================== YOUR CODE HERE =========
        #



        # ==========================================
        # Save the cost J in every iteration
        J_history[iter] = ComputeCost(X, y, theta)

    # Let's just print the first 10 Js to check that the cost is decreasing nicely
    print(J_history[0:10])
    return theta

# Run gradient descent
theta = np.zeros((2, 1))
alpha = 0.01
iterations = 1500
theta = GradientDescent(X, y, theta, alpha, iterations)
```

$$\theta_0 \leftarrow \theta_0 - \alpha \frac{1}{M} \sum_{i=1}^{M} (h_\theta(x_i) - y_i)$$

$$\theta_1 \leftarrow \theta_1 - \alpha \frac{1}{M} \sum_{i=1}^{M} (h_\theta(x_i) - y_i)\, x_i$$




```
# Predict values for population sizes of 35,000 and 70,000
predict1 = np.matmul(np.matrix('1 3.5'), theta)
print('For population = 35,000, we predict a profit of:', predict1*10000);
predict2 = np.matmul(np.matrix('1, 7'), theta)
print('For population = 70,000, we predict a profit of:', predict2*10000);
```

```
For population = 35,000, we predict a profit of: [[ 4483.98578098]]
For population = 70,000, we predict a profit of: [[ 45328.60631675]]
```

Exercise 2

Linear Regression

(with Tensorflow)

Introduction to TensorFlow™

What is TensorFlow™?

- **open source** software library
- for numerical computation using **dataflow graphs**
- originally created by **Google** as an **internal** machine learning tool, but became **open sourced** under the Apache 2.0 License in **November 2015**

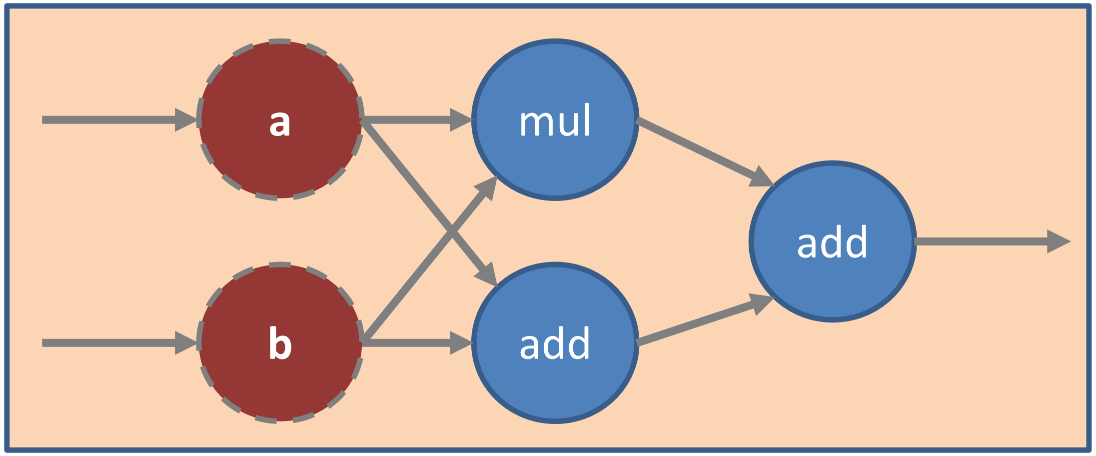
Why TensorFlow?

- [Other such libraries: **Torch**, **Theano**, **Caffe**, …]
- Python API
- Portable (one or more CPUs or GPUs in a desktop, server, or mobile device)
- Flexible (Raspberry Pi, Android, Windows, iOS, Linux,…)
- Visualization (Tensorboard)
- Checkpoints (for managing experiements)
- Auto-differentiation

Tensor

- *n*-dimensional matrices
- 0-d tensor: scalar (number)
- 1-d tensor: vector
- 2-d tensor: matrix

Data Flow Graphs




```
import tensorflow as tf
g = tf.Graph()
with g.as.default():
   a = tf.placeholder(tf.float32)
   b = tf.placeholder(tf.float32)
   op1 = tf.mul(a, b)
   op2 = tf.add(a, b)
   op3 = tf.add(op1, op2)
```

First part of TensorFlow code defines the data flow graph

```
with tf.Session() as sess:
   print sess.run(op3, feed_dict={a: 2, b:5})
   sess.close()
```

Second part of TensorFlow code: session to execute operations in the graph

Constants, Variables and Placeholders

Constants are stored in the graph definition

```
tf.constant(value, dtype=None, shape=None, name=‘Const’, verify_shape=False)
```

```
a = tf.constant([2,2], name=“a”)
b = tf.constant([[0,1], [2,3]], name=“b”)
```

Constants, Variables and Placeholders

Constants: Stored in the graph definition

Variables: What you want to train
NB: You need to initialize variables

```
tf.Variable(value, name='Const')
```

```
a = tf.Variable(2, name="scalar")
b = tf.Variable([2,3], name="vector")
c = tf.Variable([[0, 1], [2, 3]], name="matrix")
d = tf.Variable(tf.zeros([784,10]))
```

```
with tf.Session() as sess:
   sess.run(tf.global_variables_initializer())
```

Constants, Variables and Placeholders

Constants: Stored in the graph definition

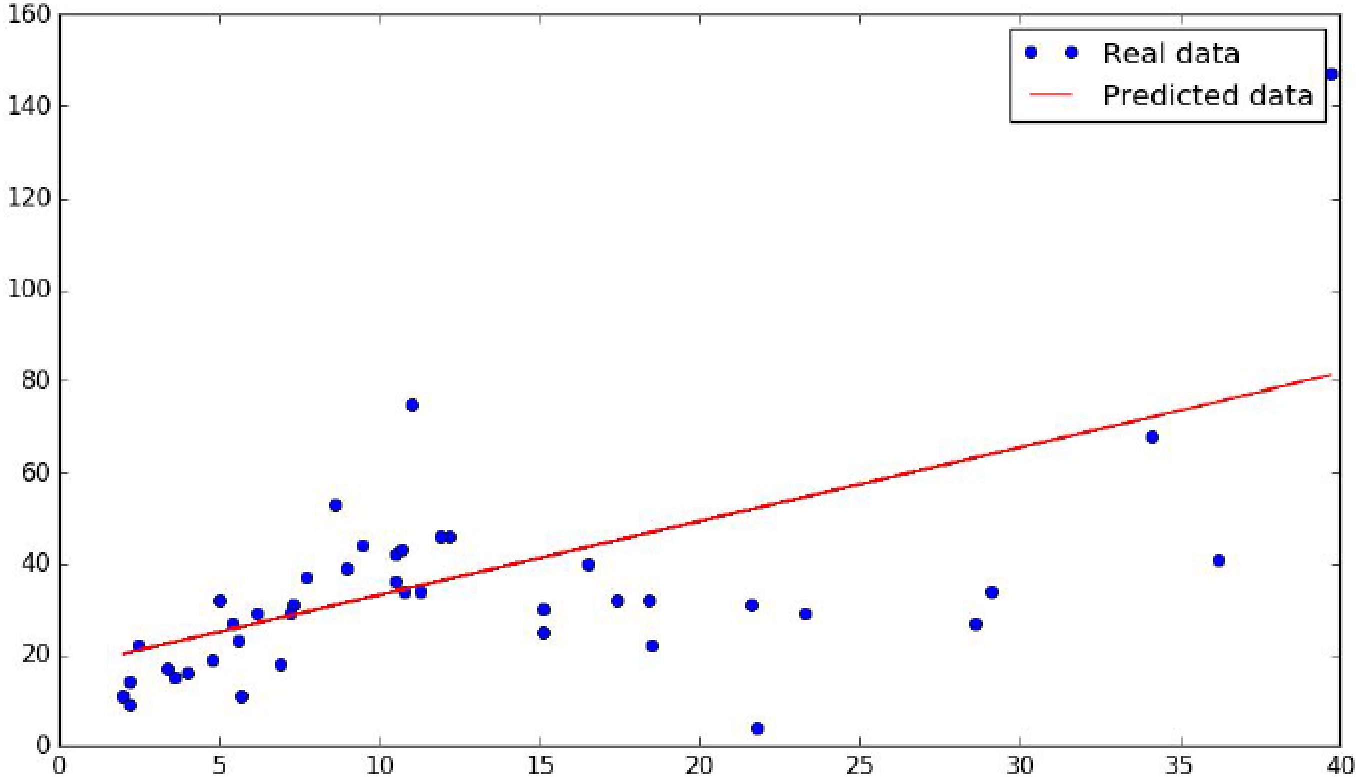
Variables: What you want to train
NB: You need to initialize variables

Placeholders: What you want to input

```
tf.placeholder(dtype, shape=None, name=None)
```

```
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
```

Linear Regression



Linear Regression (reference example)

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import xlrd

DATA_FILE = "data/fire_theft.xls"

# Step 1: read in data from the .xls file
book = xlrd.open_workbook(DATA_FILE, encoding_override="utf-8")
sheet = book.sheet_by_index(0)
data = np.asarray([sheet.row_values(i) for i in range(1, sheet.nrows)])
n_samples = sheet.nrows - 1

# Step 2: create placeholders for input X (number of fire) and label Y (number of
theft)
X = tf.placeholder(tf.float32, name="X")
Y = tf.placeholder(tf.float32, name="Y")

# Step 3: create weight and bias, initialized to 0
w = tf.Variable(0.0, name="weights")
b = tf.Variable(0.0, name="bias")

# Step 4: construct model to predict Y (number of theft) from the number of fire
Y_predicted = X * w + b
```

```python
# Step 2: create placeholders for input X (number of fire) and label Y (number of
theft)
X = tf.placeholder(tf.float32, name="X")
Y = tf.placeholder(tf.float32, name="Y")

# Step 3: create weight and bias, initialized to 0
w = tf.Variable(0.0, name="weights")
b = tf.Variable(0.0, name="bias")

# Step 4: construct model to predict Y (number of theft) from the number of fire
Y_predicted = X * w + b

# Step 5: use the square error as the loss function
loss = tf.square(Y - Y_predicted, name="loss")

# Step 6: using gradient descent with learning rate of 0.01 to minimize loss
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)
```

```python
# Step 4: construct model to predict Y (number of theft) from the number of fire
Y_predicted = X * w + b

# Step 5: use the square error as the loss function
loss = tf.square(Y - Y_predicted, name="loss")

# Step 6: using gradient descent with learning rate of 0.01 to minimize loss
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)

with tf.Session() as sess:
	# Step 7: initialize the necessary variables, in this case, w and b
	sess.run(tf.global_variables_initializer())

	# Step 8: train the model
	for i in range(100): # run 100 epochs
		for x, y in data:
			# Session runs train_op to minimize loss
			sess.run(optimizer, feed_dict={X: x, Y:y})

	# Step 9: output the values of w and b
	w_value, b_value = sess.run([w, b])
```

Programming Exercise 2: Linear Regression (with Tensorflow)
==

You will use tensorflow to solve the prediction problem of Exercise 1.

In [ ]:
```
# Run this cell to import all the libraries you need for this exercise
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

In [ ]:
```
# STEP 1: Read in the data from 'ex1data1.txt'
f = open('ex1data1.txt', 'r')
x_list = []
y_list = []

for line in f:
        x_str, y_str = line.split(',')
        x_list.append(float(x_str))
        y_list.append(float(y_str))

m = len(x_list)
x_data = np.array(x_list).reshape(m,1)
y_data = np.array(y_list).reshape(m,1)
data = np.c_[x_data,y_data]
f.close()
```

In [ ]:
```
# STEP 2: Create placeholders for X (Population of City) and Y (Profit)
#
#  *** FILL IN ***
```

In [ ]:
```
# STEP 3: Create weight and bias, initialized to 0
#
# *** FILL IN ***
```

```
In [ ]: # STEP 4: Construct model to predict Y (Profit) from X (Population of City)
        #
        # *** FILL IN ***
```

```
In [ ]: # STEP 5: Use the square error as the loss function
        #
        # *** FILL IN ***
```

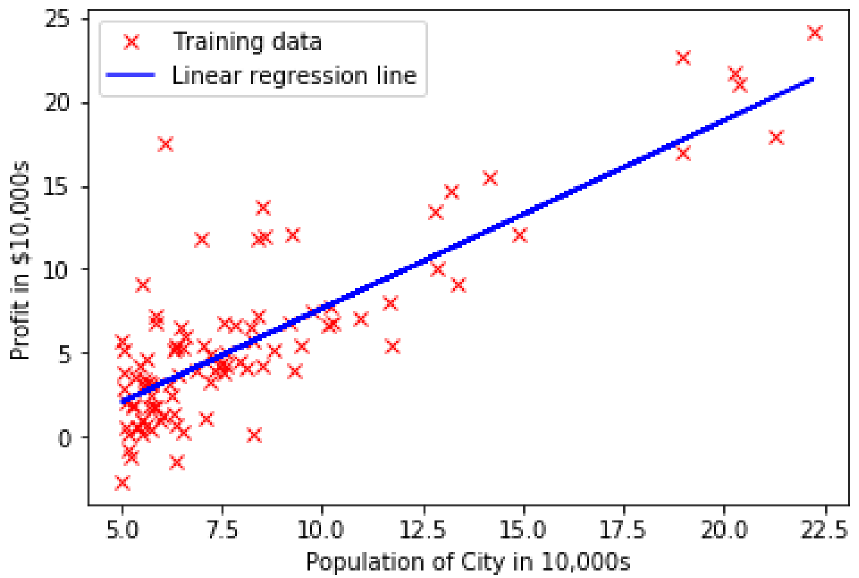
```
In [ ]: # STEP 6: Use gradient descent with learning rate of 0.01 to minimize loss
        optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(cost)
```

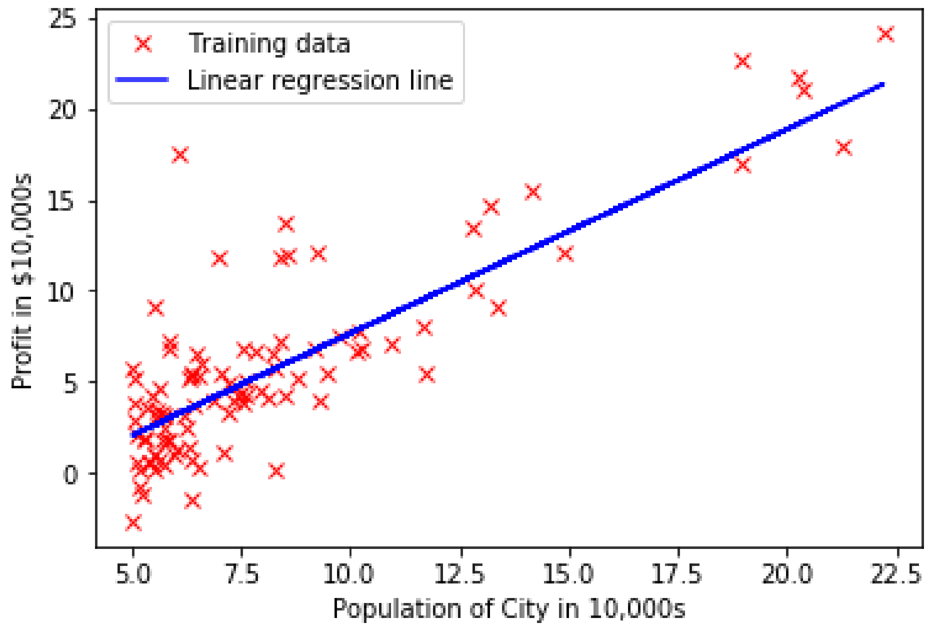
```
In [ ]: with tf.Session() as sess:

            # STEP 7: Initialize variables w and b
            sess.run(tf.global_variables_initializer())

            # STEP 8: Train the model
            for i in range(0,1500):
                for x, y in data:
                    sess.run(optimizer, feed_dict={X:x,Y:y})

            # STEP 9: Obtain the values of w and b, cost and Y_predicted
            w_value, b_value = sess.run([w,b])
            cost_value = sess.run(cost, feed_dict={X: x_data, Y: y_data})
            y_predicted = sess.run(Y_predicted, feed_dict={X: x_data, Y: y_data})
```

```python
# print w_value and b_value and cost_value
print('b_value (previously theta_0) found by gradient descent: ', b_value)
print('w_value (previously theta_1) found by gradient descent: ', w_value)
print('Cost: ', cost_value)
```

```
b_value (previously theta_0) found by gradient descent:  -3.58837
w_value (previously theta_1) found by gradient descent:  1.12367
Cost:  4.54607
```

```python
# Plot the linear fit
plt.xlabel('Population of City in 10,000s')
plt.ylabel('Profit in $10,000s')
plt.plot(x_data,y_data,marker='x',lw=0,color='r',label='Training data')
plt.plot(x_data,y_predicted,linestyle='-',color='b',label='Linear regression line')
plt.legend()
plt.show()
```






```
# Predict values for population sizes of 35,000 and 70,000
predict1 = w_value * 3.5 + b_value
print('For population = 35,000, we predict a profit of:', predict1*10000);
predict2 = w_value * 7 + b_value
print('For population = 70,000, we predict a profit of:', predict2*10000);
```

```
For population = 35,000, we predict a profit of: 3444.57507133
For population = 70,000, we predict a profit of: 42772.8533745
```

```
CONGRATULATIONS - YOU HAVE COMPLETED Exercise 2 !!!!
```

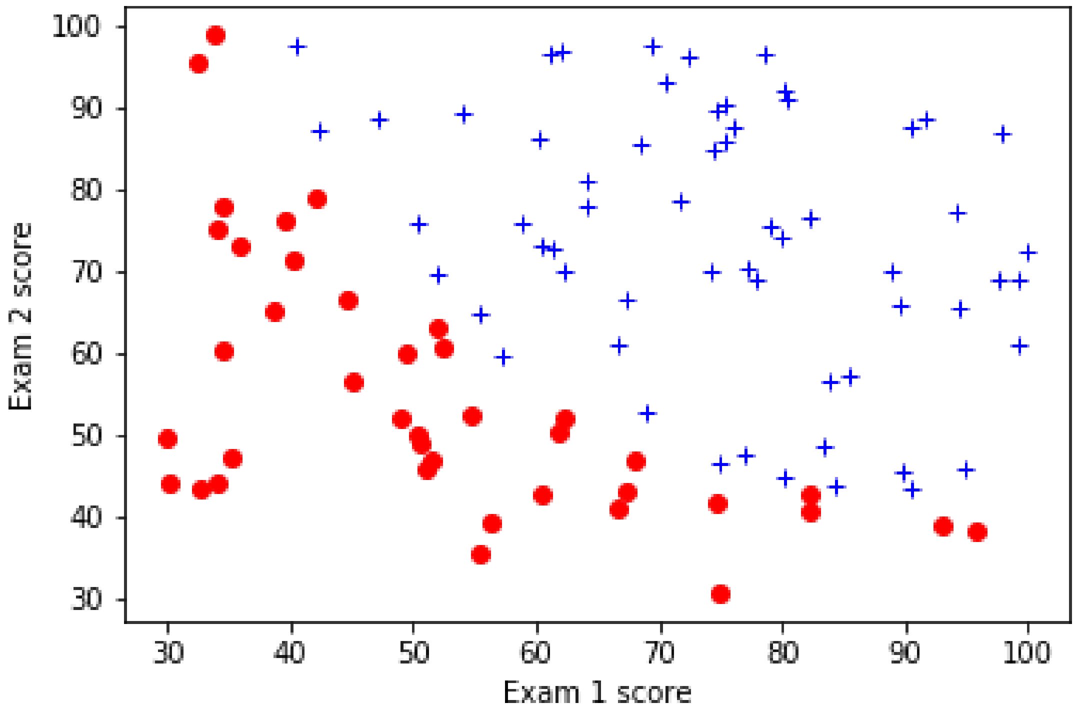
Exercise 3

Logistic Regression

(without Tensorflow)

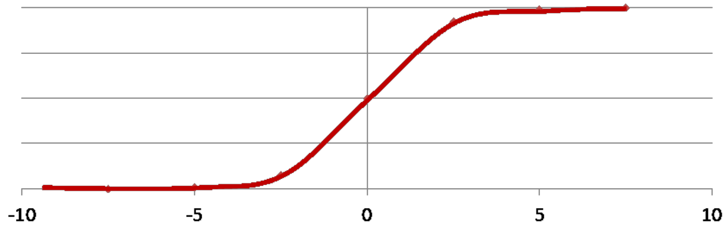
Data

```
Plotting data with + indicating (y = 1) examples and o indicating (y = 0) examples
```



```
def sigmoid(z):
    #
    # INSTRUCTIONS:
    # Write the code for sigmoid
    #
    # ====================== YOUR CODE HERE ======================
    #
```

$$\sigma(s) = \frac{e^s}{1+e^s} = \frac{1}{1+e^{-s}}$$



```
def CostFunction(theta, X, y):
    #
    # INSTRUCTIONS:
    # Write the code for CostFunction, which is a function that returns
    # the cost of using theta as the parameter of the logistic regression
    #
    # ====================== YOUR CODE HERE ======================
    #
```

$$h(\boldsymbol{x}) = \sigma(\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{x})$$

$$J(\boldsymbol{\theta}) = -\frac{1}{M}\sum_{i=1}^{M}\left[y_i \log h(\boldsymbol{x}_i) + (1 - y_i)\log(1 - h(\boldsymbol{x}_i))\right]$$

```
    # =============================================================
    return J
```

```python
def Gradient(theta, X, y):
    #
    # INSTRUCTIONS:
    # Write the code for Gradient, which is a function that computes
    # the gradient  of the cost with respect to the parameters
    # using theta as the parameter of the logistic regression
    #
    # ====================== YOUR CODE HERE ======================
    #
```

Compute for all j:

$$\text{grad[j]} = \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j} = -\frac{1}{M}\sum_{i=1}^{M}\big(y_i - h(\boldsymbol{x}_i)\big)x_i^{(j)}$$

```python
    # =============================================================
    return grad
```

```
======== STEP 3: Run Optimization Function ==========
```

```
# Run optimize function
initial_theta = np.zeros(3)
result = opt.minimize(fun=CostFunction,x0=initial_theta,args=(X,y),method='TNC',jac=Gradient)
print('Cost at theta: ', CostFunction(result.x,X,y))
print('theta: ')
print(result.x)
```

```
Cost at theta:  0.203497701589
theta:
[-25.16131858   0.20623159   0.20147149]
```

```
======== STEP 4: Make Predictions ==========
```

```
# Predict probability that a student with score 45 on exam 1
# and score 85 on exam 2 will be admitted
theta_min = np.matrix(result.x).reshape(3,1)
prob = sigmoid(np.matmul(np.matrix('1 45 85'),theta_min))
print('For a student with scores 45 and 85, we predict an admission probability of ', prob)
```

```
For a student with scores 45 and 85, we predict an admission probability of  [[ 0.77629062]]
```

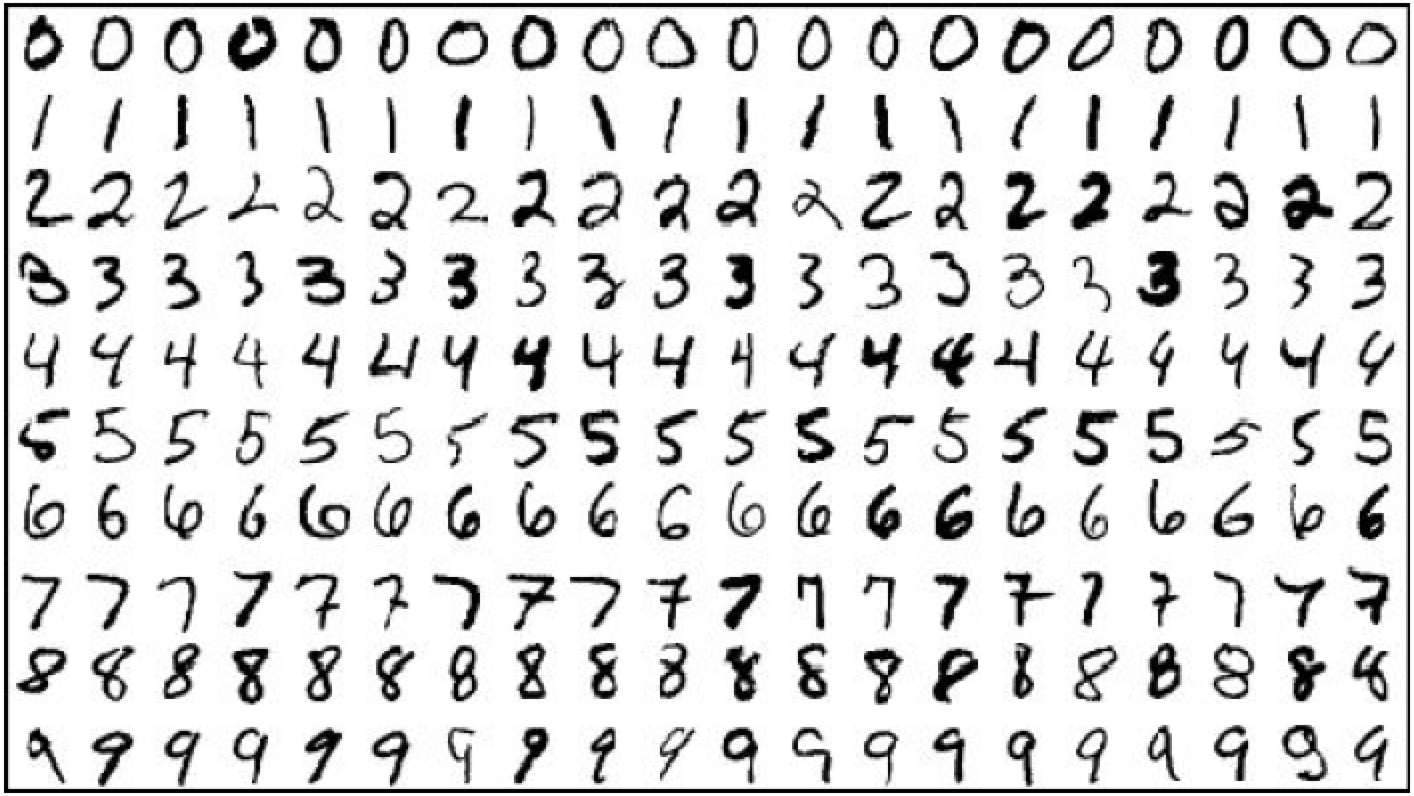
Exercise 4

Logistic Regression

(with Tensorflow)

Logistic Regression

MNIST (Mixed National Institute of Standards and Technology) database



```python
import time
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

# Step 1: Read in data
# using TF Learn's built in function to load MNIST data to the folder data/mnist
MNIST = input_data.read_data_sets("/data/mnist", one_hot=True)

# Step 2: Define parameters for the model
learning_rate = 0.01
batch_size = 128
n_epochs = 25

# Step 3: create placeholders for features and labels
# each image in the MNIST data is of shape 28*28 = 784
# therefore, each image is represented with a 1x784 tensor
# there are 10 classes for each image, corresponding to digits 0 - 9.
# each label is one hot vector.
X = tf.placeholder(tf.float32, [batch_size, 784])
Y = tf.placeholder(tf.float32, [batch_size, 10])

# Step 4: create weights and bias
# w is initialized to random variables with mean of 0, stddev of 0.01
# b is initialized to 0
# shape of w depends on the dimension of X and Y so that Y = tf.matmul(X, w)
# shape of b depends on Y
w = tf.Variable(tf.random_normal(shape=[784, 10], stddev=0.01), name="weights")
b = tf.Variable(tf.zeros([1, 10]), name="bias")
```

```python
# Step 5: predict Y from X and w, b
# the model that returns probability distribution of possible label of the image
# through the softmax layer
# a batch_size x 10 tensor that represents the possibility of the digits
logits = tf.matmul(X, w) + b

# Step 6: define loss function
# use softmax cross entropy with logits as the loss function
# compute mean cross entropy, softmax is applied internally
entropy = tf.nn.softmax_cross_entropy_with_logits(logits, Y)
loss = tf.reduce_mean(entropy) # computes the mean over examples in the batch

# Step 7: define training op
# using gradient descent with learning rate of 0.01 to minimize cost
optimizer =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(loss)

init = tf.global_variables_initializer()

with tf.Session() as sess:
	sess.run(init)
	n_batches = int(MNIST.train.num_examples/batch_size)
	for i in range(n_epochs): # train the model n_epochs times
		for _ in range(n_batches):
			X_batch, Y_batch = MNIST.train.next_batch(batch_size)
			sess.run([optimizer, loss], feed_dict={X: X_batch, Y:Y_batch})
```

```
# test the model
        n_batches = int(MNIST.test.num_examples/batch_size)
        total_correct_preds = 0
        for i in range(n_batches):
                X_batch, Y_batch = MNIST.test.next_batch(batch_size)
                _, loss_batch, logits_batch = sess.run([optimizer, loss, logits],
feed_dict={X: X_batch, Y:Y_batch})
                preds = tf.nn.softmax(logits_batch)
                correct_preds = tf.equal(tf.argmax(preds, 1), tf.argmax(Y_batch, 1))
                accuracy = tf.reduce_sum(tf.cast(correct_preds, tf.float32)) # similar
to numpy.count_nonzero(boolarray) :(
                total_correct_preds += sess.run(accuracy)

        print "Accuracy {0}".format(total_correct_preds/MNIST.test.num_examples)
```