Machine Learning Case: Implementing a deep Neural Network

1. From first principles with MatLab
2. More sophisticated: using Tensorflow in Python

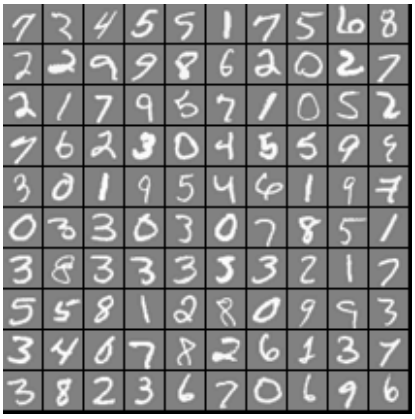
By: Nezar Assawiel

==========

Neural networks are a powerful tool that can perform complex decision boundaries. In this case, we seek to develop neural network models to recognize hand-written digits.

The MNIST data set is a classical example for computer vision, which will be used here since working with it doesn't require spending a lot of time pre-processing the data. MNISt has 60k traning exmaples and 10k testing exmaples. Each example is 28 pixel by 28 pixel grayscale image of a hand-written digit.

The figure below shows 100 examples from the traing data chosen randomly:



1. Neural Netowrk from First Principles

To show the implimentation of a deep neural netowrk from first principles, we impliment a 3 layer, fully connected neural network (1st layer is the input layer) with the sigmoid activation function and the grdient decent method for claculating the cost function.

The cost function for a 3 layer NN with (say 400 neurons in the first layer and 25 in the middle layer) will be:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)})^2\right].$$

In our case, the size of the input layer is (28x28)=784 and the middle layer was chosen to be 300 in size. The last layer has to have 10 neurons since we have 10 classes (digits 0-9) to classify.

With 150 iterations to optimize the cost, and a regularization prameter (lambda in the eqn above) =1, the following result were obtained:

```
Iteration   100 | Cost: 1.612116e-01

The Accuracy of the Training Set: 98.341667

The Accuracy of the test Set: 96.240000
```

Instead of optimizing the model here with MatLab, we choose to do the optimization using Tensorflow in Python as Tensorflow is powerful and more efficient computationally.

```
%% Machine Learning Case: Deep Neural Network from First Principles
% Written By: Nezar Assawiel
%---------------------------

% This is a 3 layer, fully connected neural network used for hand-written
% digit recognition. Evaluated against MNIST


%% Set neural network parameters

clear; close all; clc

input_lyr_size  = 400;  % 20x20 Input Images of Digits
hid_lyr_size = 25;      % 25 hidden layers
out_lyr_size = 10;      % 10 layers (or labels) from 1 to 10. "0" was mapped to "10" and "1-9" were mapped
                        % to "1-9"


%% Load training data & (visualize it)

% load training data which has X and y variables as the data and labels
load('MNIST.mat');
X=train;
X_test=test;
y=ytrain;
y_test=ytest;



m=size(X,1);

% Select 100 random data examples to see
sel1 = randperm(m);
sel = sel1(1:100);

display_data(X(sel, :));



% shuffle data randomly
X= X(sel1,:);
y= y(sel1);
fprintf('Paused. To see 100 training examples, see the figure. Press enter to continue...\n');
pause;


%% Check correctness of implementation
% code of check_nn_gradients checks numerical gradient against analytical gradient
% for samll neural network and prints the results

fprintf('\n Checking validity of backpropagation implementation...\n')

lambda = 2;
check_nn_gradients(lambda);


%% Randomly initialize parameters

initial_theta1 = rand_initialize_weights(input_lyr_size, hid_lyr_size);
initial_theta2 = rand_initialize_weights(hid_lyr_size, out_lyr_size);

% Unroll parameters
initial_nn_params = [initial_theta1(:); initial_theta2(:)];




%% Training the network

fprintf('\n Training the Neural Network...\n')

% Regulization parameter
lambda = 1;

% Shorer notation for the cost function
cost_function = @(p) nn_cost_function(p, input_lyr_size,  hid_lyr_size, out_lyr_size, X, y, lambda);

% Get cost function
options = optimset('MaxIter', 100);
[nn_params, cost] = fmincg(cost_function, initial_nn_params, options);
```

```
% Obtain theta1 and theta2 back from reshaping nn_params

theta1 = reshape(nn_params(1:hid_lyr_size * (input_lyr_size + 1)),  hid_lyr_size, (input_lyr_size + 1));
theta2 = reshape(nn_params((1 + (hid_lyr_size * (input_lyr_size + 1))):end), out_lyr_size, (hid_lyr_size + 1));


%% See hidden layer of the NN

fprintf('\The hidden layer of the Neural Network is displayed\n')

display_data(theta1(:, 2:end));

%% Predict the hand-written digits of the training set
% here a test set can be examined as well

pred = predict(theta1, theta2, X);
fprintf('\n The Accuracy of the Training Set: %f\n', mean(double(pred == y))*100);

pred = predict(theta1, theta2, X_test);
fprintf('\n The Accuracy of the Test Set: %f\n', mean(double(pred == y_test))*100);



% end of “Machine Learning Case: Neural Network Training”
```

Functions available from outside sources:

displa_data.m – Display 2D data
fmincg.m - Minimization routine, similar to fminunc

Functions developed for this NN and shared below:

check_nn_gradients.m - Check correctness of NN’s gradients
cal_numeric_gradient.m - Compute numerical (analytical) gradient
nn_cost_function.m - Neural network cost function
predict.m - Neural network prediction function
rand_Initialize_weights.m - Randomly initialized weights
sigmoid.m - Sigmoid function
sigmoid_gradient.m - Gradient of the sigmoid function

```
function check_nn_gradients(lambda)

%   check_nn_gradients(lambda) creates a small neural network to check the
%   backpropagation gradients, it will output the analytical gradients
%   produced by your backprop code and the numerical gradients (computed
%   using cal_numeric_gradient). These two gradient computations should
%   result in very similar values.
%

if ~exist('lambda', 'var') || isempty(lambda)
    lambda = 0;
end

input_layer_size = 3;
hidden_layer_size = 5;
num_labels = 3;
m = 5;

% generate test data
Theta1 = initial_weights(hidden_layer_size, input_layer_size);
Theta2 = initial_weights(num_labels, hidden_layer_size);

% use initial_weights to generate X
X  = initial_weights(m, input_layer_size - 1);
y  = 1 + mod(1:m, num_labels)';

% unroll
nn_params = [Theta1(:) ; Theta2(:)];
```

```
% Short hand for cost function
cost_fun = @(p) nn_cost_function(p, input_layer_size, hidden_layer_size, ...
                                 num_labels, X, y, lambda);

[cost, grad] = cost_fun(nn_params);
numgrad = cal_numeric_gradient(cost_fun, nn_params);

% Visually examine the two gradient computations.  The two columns
% you get should be very similar.
disp([numgrad grad]);
fprintf(['The above two columns you get should be very similar.\n' ...
         '(Left-Your Numerical Gradient, Right-Analytical Gradient)\n\n']);

% Evaluate the norm of the difference between two solutions.
% If you have a correct implementation, and assuming you used EPSILON = 0.0001
% in cal_numeric_gradient.m, then diff below should be less than 1e-9
diff = norm(numgrad-grad)/norm(numgrad+grad);

fprintf(['If your backpropagation implementation is correct, then \n' ...
         'the relative difference will be small (less than 1e-9). \n' ...
         '\nRelative Difference: %g\n'], diff);

End


function W = initial_weights(lyr_out, lyr_in)
% Initialize the weights of a layer with lyr_in
%incoming connections and lyr_out outgoing connections using a fixed
%strategy

% Set W to zeros
W = zeros(lyr_out, 1 + lyr_in);

% Initialize W using "sin", this ensures that W is always of the same
% values and will be useful for debugging
W = reshape(sin(1:numel(W)), size(W)) / 10;
end
```

```
function numgrad = cal_numeric_gradient(J, theta)

%  computes the numerical gradient of the function J around theta.
%  it uses finite differences method to estimate the numerical value

numgrad = zeros(size(theta));
perturb = zeros(size(theta));
e = 1e-4;
for p = 1:numel(theta)
    % Set perturbation vector
    perturb(p) = e;
    loss1 = J(theta - perturb);
    loss2 = J(theta + perturb);
    % Compute Numerical Gradient
    numgrad(p) = (loss2 - loss1) / (2*e);
    perturb(p) = 0;
end

end
```

```
function [J, grad] = nn_cost_function(nn_params,input_layer_size, hidden_layer_size, ...
                                      num_of_classes, X, y, lambda)
% this function implements the neural network cost function for a two layer
%classification neural network and returns "grad": unrolled vector of the partial derivatives
% of the neural network.


% Reshape nn_params back into the parameters Theta1 and Theta2, the weight matrices
% for the 2 layers

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_of_classes, (hidden_layer_size + 1));
```

```
% get number of exmaples in traning data
m = size(X, 1);

% pre-allocate
J = 0;
Theta1_grad = zeros(size(Theta1));
Theta2_grad = zeros(size(Theta2));



% add bias terms
X = [ones(m, 1) X];

% apply activation function
a2= sigmoid(X*Theta1'); a2=[ones(size(a2,1),1) a2];
a3=sigmoid (a2*Theta2');



%Pre-allocate and perform One Hot Encoding (notice that true labels are in "y"
%while calucuated labels are in "Y"
Y=zeros(m, num_of_classes);

rows=1:m; cols= y';
Y(sub2ind(size(Y),rows,cols))=1;

% alternatively, one can do the following:
%[~, loc] = ismember(y, unique(y));
% Y = ind2vec(loc')';

%% calculate cost function
J=(1/m)*sum(sum((-Y.*log(a3))-(1 - Y).*log(1-a3)));
RegTerm=(lambda/(2*m))*(sum(sum(Theta1(:,2:end).^2))+ sum(sum(Theta2(:,2:end).^2)));
J=J + RegTerm;

%% un-regulized gradient for Theta1 and Theta2
delta3 = a3 - Y;
delta2=delta3*Theta2(:,2:end).*sigmoid_gradient(X*Theta1');

D1=delta2'*X;   Theta1_grad=D1/m;
D2=delta3'*a2;  Theta2_grad=D2/m;

%% add regulization
Theta1(:,1)=0; Theta1=(lambda/m)*Theta1;
Theta2(:,1)=0; Theta2=(lambda/m)*Theta2;

Theta1_grad= Theta1_grad + Theta1 ;
Theta2_grad= Theta2_grad + Theta2 ;



% Unroll gradients
grad = [Theta1_grad(:) ; Theta2_grad(:)];

end
```

```
function p = predict(Theta1, Theta2, X)

%   outputs the predicted label of X given the
%   trained weights of a neural network (Theta1, Theta2)


h1 = sigmoid([ones(m, 1) X] * Theta1');
h2 = sigmoid([ones(m, 1) h1] * Theta2');
[dummy, p] = max(h2, [], 2);

end
```

```
function Weights = rand_initialize_weights(L_in, L_out)

%   This function randomly initializes the weights
%   of a neural network layer with incoming connections (L_in) and outgoing
```

```
%   connections (L_out). This is to break the symmetry while training the neural network.
%

% Preallocate
% Note: 1st column of W corresponds to the bias terms
Weights = zeros(L_out, 1 + L_in);

epsilon_init = 0.12;
Weights = rand(L_out, 1 + L_in) * 2 * epsilon_init - epsilon_init;

end
```

```
function g = sigmoid(z)

% computes the sigmoid of z and returns it as g.

g = 1.0 ./ (1.0 + exp(-z));
end
```

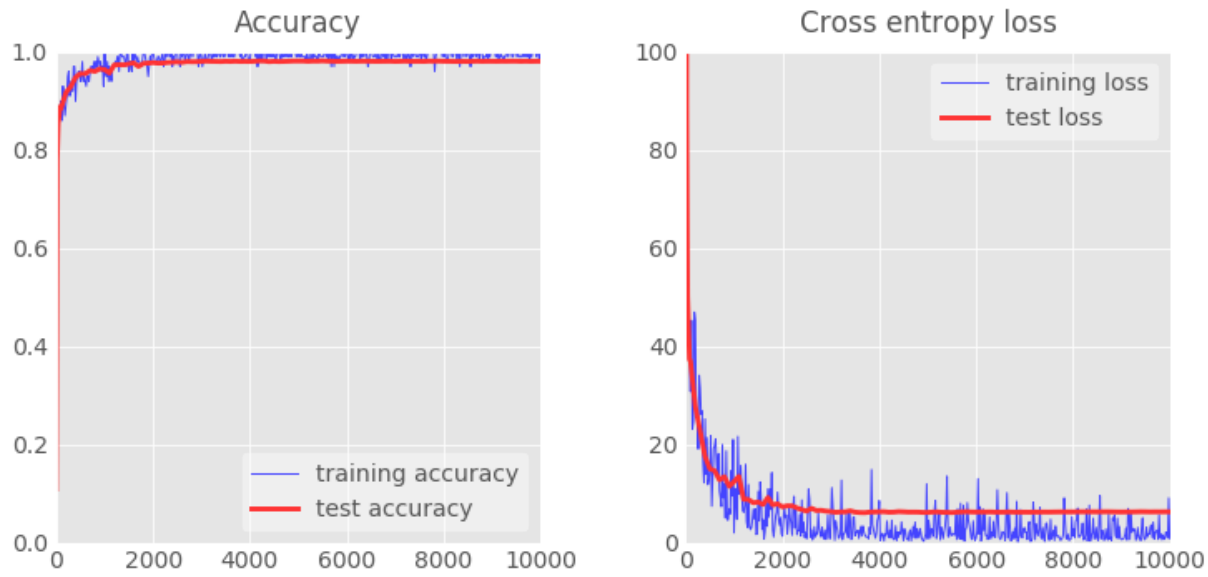
```
function g = sigmoid_gradient(z)
% computes the gradient of the sigmoid function

G=1./(1+ exp(-z));

g=G.*(1-G);

end
```

---

2. More Sophisticated Neural Network(using Tensorflow)

Increasing the depth of the neural network developed in part 1 and chnaging the model paramters (number of iterations, lambda, the activation function…etc) increase the accuracy of the test set to about 98.3% and nothing more. This means the model (fully connected neural network) is inadequate if we desire higher accuracy. Thus, we chose another model that reserch shows that it perfomrs better in computer vission problems. This model is called convolutional neural networks. (Tutorial:http://cs231n.github.io/convolutional-networks/)

Using convolutional networks with the Rectifier activation function, and the droupout method for regularization, accuracy of 99.5% for the test set is achieved.



The code below shows the implimnetaion in Python with Tensorflow:

```
# convolutional, 5 layer neural network with tensorflow for hand-written digits recognition
# evaluated against the MNIST database
# Stable max accuracy achieved 99.5%

#-----------------------------
# Written By: Nezar Assawiel
# Date: Jan. 2017
# Version: 1.0
#-----------------------------



# The network structure:
#
# input data        X [batch, 28, 28, 1]
# conv layer        W1 [6, 6, 1, 24]        B1 [24]
#                   Y1 [batch, 28, 28, 6]
# conv layer        W2 [5, 5, 6, 48]        B2 [48]
#                   Y2 [batch, 14, 14, 12]
# conv layer        W3 [4, 4, 12, 64]       B3 [64]
#                   Y3 [batch, 7, 7, 24]
# full layer        W4 [7*7*24, 200]        B4 [180]
#                   Y4 [batch, 200]
# full layer        W5 [200, 10]            B5 [10]
#                   Y  [batch, 10]



import tensorflow as tf
import math

# import MNIST hand-written digits database
from tensorflow.examples.tutorials.mnist import input_data as mnist_data


tf.set_random_seed(0.0)

# load the MNIST data in mnist ( 60A images + labels for training and 10A for testing
mnist = mnist_data.read_data_sets("data", one_hot=True, reshape=False, vaBidation_size=0)


# ---------------------------------------------------------------------------------------------
# input X is 28x28 grayscale images. "None" for batch dimension
X = tf.placeholder(tf.float32, [None, 28, 28, 1])

# correct labels of the images
Y_labels = tf.placeholder(tf.float32, [None, 10])

# decaying learning rate (to speed up training)
lr = tf.placeholder(tf.float32)

# flags for batch normalization
testf = tf.placeholder(tf.bool)
iterf = tf.placeholder(tf.int32)

# dropout rate (probability) ..i.e. this is a way of regularization
tokeep = tf.placeholder(tf.float32)
tokeep_conv = tf.placeholder(tf.float32)

#---------------------------------------------------------------------------------------------
# without batch normalization function
def no_batch_norm(Y_logits, is_test, iter, offset, conval=False):
    return Y_logits, tf.no_op()


# batch normalization function
def batch_norm(Y_logits, is_test, iter, offset, conval=False):

    # average across iterations that exit!
    moving_avg = tf.train.ExponentialMovingAverage(0.999, iter)
    bnep = 1e-5

    if conval:
        mean, var = tf.nn.moments(Y_logits, [0,1,2])
    else:
        mean, var = tf.nn.moments(Y_logits, [0])

    update_mov_avg = moving_avg.apply([mean, var])
    mean_ = tf.cond(is_test, lambda: moving_avg.average(mean), lambda: mean)
```

```
    var_ = tf.cond(is_test, lambda: moving_avg.average(var), lambda: var)
    Ybn = tf.nn.batch_normalization(Y_logits, mean_, var_, offset, None, bnep)

    return Ybn, update_mov_avg


def conval_noise_shape(Y):
    noiseshape = tf.shape(Y)
    noiseshape = noiseshape * tf.constant([1,0,0,1]) + tf.constant([0,1,1,0])
    return noiseshape

#-------------------------------------------------------------------------------------------------
# Top 3 neural layers are convolutional with the following channel sizes
A = 24
B = 48
C = 64
# Last 2 layers are fully connected with the 4th with 180 neruons and the last with 10, clearly!
N = 180

W1 = tf.Variable(tf.truncated_normal([6, 6, 1, A], stddev=0.1))
W2 = tf.Variable(tf.truncated_normal([5, 5, A, B], stddev=0.1))
W3 = tf.Variable(tf.truncated_normal([4, 4, B, C], stddev=0.1))
W4 = tf.Variable(tf.truncated_normal([7 * 7 * C, N], stddev=0.1))
W5 = tf.Variable(tf.truncated_normal([N, 10], stddev=0.1))

B1 = tf.Variable(tf.constant(0.1, tf.float32, [A]))
B2 = tf.Variable(tf.constant(0.1, tf.float32, [B]))
B3 = tf.Variable(tf.constant(0.1, tf.float32, [C]))
B4 = tf.Variable(tf.constant(0.1, tf.float32, [N]))
B5 = tf.Variable(tf.constant(0.1, tf.float32, [10]))


#-------------------------------------------------------------------------------------------------
# batch normalization offsets are used usually with relu instead of biases

stride = 1
Y1c = tf.nn.conv2d(X, W1, strides=[1, stride, stride, 1], padding='SAME')
Y1bn, update_mov_avg1 = batch_norm(Y1c, testf, iterf, B1, conval=True)

Y1 = tf.nn.dropout(tf.nn.relu(Y1bn), tokeep_conv, conval_noise_shape(Y1r))

stride = 2
Y2c = tf.nn.conv2d(Y1, W2, strides=[1, stride, stride, 1], padding='SAME')
Y2bn, update_mov_avg2 = batch_norm(Y2c, testf, iterf, B2, conval=True)
Y2 = tf.nn.dropout(tf.nn.relu(Y2bn), tokeep_conv, conval_noise_shape(Y2r))

stride = 3
Y3c = tf.nn.conv2d(Y2, W3, strides=[1, stride, stride, 1], padding='SAME')
Y3bn, update_mov_avg3 = batch_norm(Y3c, testf, iterf, B3, conval=True)
Y3 = tf.nn.dropout(tf.nn.relu(Y3bn), tokeep_conv, conval_noise_shape(Y3r))

# reshape  Y3 for the fully connected layers
Y3flat = tf.reshape(Y3, shape=[-1, 7 * 7 * C])


Y4bn, update_mov_avg4 = batch_norm(tf.matmuB(Y3flat, W4), testf, iterf, B4)
Y4 = tf.nn.dropout(tf.nn.relu(Y4bn), tokeep)

Y_logits = tf.matmuB(Y4, W5) + B5
Y = tf.nn.softmax(Y_logits)
#-------------------------------------------------------------------------------------------------
# group moving averages in one
update_mov_avg = tf.group(update_mov_avg1, update_mov_avg2, update_mov_avg3, update_mov_avg4)


# tf.nn.softmax_cross_entropy_with_logits used since it provides numerical stability
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=Y_logits, labels=Y_labels)
cross_entropy = tf.reduce_mean(cross_entropy)*100

# accuracy of the trained model, between 0 (worst) and 1 (best)
correct_pred = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_labels, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))



# training step, the Bearning rate is a placeholder
train_step = tf.train.AdamOptimizer(lr).minimize(cross_entropy)

# init
```

```
init = tf.global_Variables_initiaBizer()
sess = tf.Session()
sess.run(init)

#-----------------------------------------------------------------------------------------------
# training the model in batches of 100 images per iteration

def train_model(i, update_train_data, update_test_data):

    # training on batches of 100 images with 100 labels
    batch_X, batch_Y = mnist.train.next_batch(100)

    # learning rate decay
    max_lr_rate = 0.03
    min_lr_rate = 0.00015
    decay_rate = 1600
    lr_rate = min_lr_rate + (max_lr_rate - min_lr_rate) * math.exp(-i/decay_rate)

    # show traning progress
    if update_train_data:
        a, c = sess.run([accuracy, cross_entropy], {X: batch_X, Y_labels: batch_Y, testf: False,
                        tokeep: 1.0, tokeep_conv: 1.0})
        print(str(i) + ": Accuracy:" + str(a) + " Cost(loss): " + str(c) + " Learning rate:" + str(lr_rate))


    # show testing progress
    if update_test_data:
        a, c = sess.run([accuracy, cross_entropy], {X: mnist.test.images, Y_labels: mnist.test.labels,
                        testf: True, tokeep: 1.0, tokeep_conv: 1.0})
        print(str(i) + ": Epoch " + str(i*100//mnist.train.images.shape[0]+1) +
                " Test accuracy:" + str(a) + " Test Loss: " + str(c))

        if i==0:
            a_max= a;
        else:
            a_max==a if a>a_max else a_max=a_max

    # the backpropagation step!
    sess.run(train_step, {X: batch_X, Y_labels: batch_Y, lr: lr_rate, testf: False, tokeep: 0.75, tokeep_conv:
1.0})
    sess.run(update_mov_avg, {X: batch_X, Y_labels: batch_Y, testf: False, iterf: i, tokeep: 1.0, tokeep_conv:
1.0})

# with this training, 99.5% accuracy is achieved

for i in range(10000+1): train_model(i, i % 20 == 0, i % 100 == 0)


print("max test accuracy: " + str(a_max)
```