The following is the tutorial provided by Martin Görner *et al*. adopted and edited for the lab work on CNN Machine Learning class. The original content can be found in the GOOGLE's developers branch.

CNN lab work

Part 2

A.Sarsembayev , N.Duzbayev

IITU, 2018

Outline

Intro

It is assumed that you have already finished writing the code for the 1-layer NN. Your goal was to write it and thoroughly understand it in order to improve it later.

The goal of the presented work is to add additional layers to the network. The final accuracy of text recognition should be ~97%.

For this work you will need

- Python 2 or 3 (Python 3 recommended)
- TensorFlow
- Matplotlib (Python visualisation library)
- The baseline code (the code which you have written already)

Let's check what we have up till now

```
import tensorflow as tf

X = tf.placeholder(tf.float32, [None, 28, 28, 1])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

init = tf.initialize_all_variables()
```

First we define TensorFlow variables and placeholders. Variables are all the parameters that you want the training algorithm to determine for you. In our case, our weights and biases.

Placeholders are parameters that will be filled with actual data during training, typically training images. The shape of the tensor holding the training images is [None, 28, 28, 1] which stands for:

28, 28, 1: our images are 28x28 pixels x 1 value per pixel (grayscale). The last number would be 3 for color images and is not really necessary here.

None: this dimension will be the number of images in the mini-batch. It will be known at training time.

```
# model
Y = tf.nn.softmax(tf.matmul(tf.reshape(X, [-1, 784]), W) + b)
# placeholder for correct labels
```

```
Y_ = tf.placeholder(tf.float32, [None, 10])

# loss function
cross_entropy = -tf.reduce_sum(Y_ * tf.log(Y))

# % of correct answers found in batch
is_correct = tf.equal(tf.argmax(Y,1), tf.argmax(Y_,1))
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
```

The first line is the model for our 1-layer neural network. The formula is the one we established in the previous theory section. The tf.reshape command transforms our 28x28 images into single vectors of 784 pixels. The "-1" in the reshape command means "computer, figure it out, there is only one possibility". In practice it will be the number of images in a mini-batch.

We then need an additional placeholder for the training labels that will be provided alongside training images.

Now, we have model predictions and correct labels so we can compute the cross-entropy. tf.reduce_sum sums all the elements of a vector.

The last two lines compute the percentage of correctly recognised digits.

```
optimizer = tf.train.GradientDescentOptimizer(0.003)
train_step = optimizer.minimize(cross_entropy)
```

This where the TensorFlow magic happens. You select an optimiser (there are many available) and ask it to minimise the cross-entropy loss. In this step, TensorFlow computes the partial derivatives of the loss function relatively to all the weights and all the biases (the gradient). This is a formal derivation, not a numerical one which would be far too time-consuming.

The gradient is then used to update the weights and biases. 0.003 is the learning rate.

Finally, it is time to run the training loop. All the TensorFlow instructions up to this point have been preparing a computation graph in memory but nothing has been computed yet.

The computation requires actual data to be fed into the placeholders you have defined in your TensorFlow code. This is supplied in the form of a Python dictionary where the keys are the names of the placeholders.

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

for i in range(1000):
    # load batch of images and correct answers
    batch_X, batch_Y = mnist.train.next_batch(100)
    train_data={X: batch_X, Y_: batch_Y}

    # train
    sess.run(train_step, feed_dict=train_data)
```

The train_step that is executed here was obtained when we asked TensorFlow to minimise out cross-entropy. That is the step that computes the gradient and updates weights and biases.

Finally, we also need to compute a couple of values for display so that we can follow how our model is performing.

The accuracy and cross entropy are computed on training data using this code in the training loop (every 10 iterations for example):

```
# success ?
a,c = sess.run([accuracy, cross_entropy], feed_dict=train_data)
```

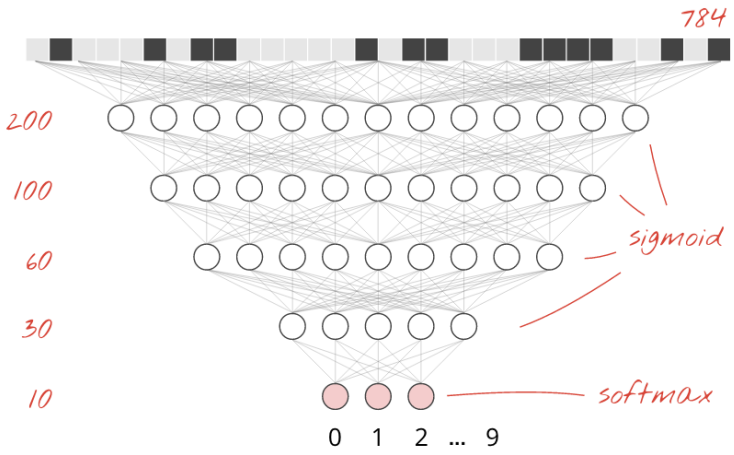
The same can be computed on test data by supplying test instead of training data in the feed dictionary (do this every 100 iterations for example. There are 10,000 test digits so this takes some CPU time):

```
# success on test data ?
test_data={X: mnist.test.images, Y_: mnist.test.labels}
a,c = sess.run([accuracy, cross_entropy], feed=test_data)
```

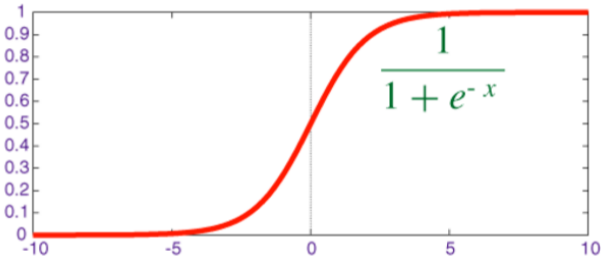
This simple model already recognises 92% of the digits. Not bad, but you will now improve this significantly.

Adding layers.

To improve the recognition accuracy we will add more layers to the neural network. The neurons in the second layer, instead of computing weighted sums of pixels will compute weighted sums of neuron outputs from the previous layer. Here is for example a 5-layer fully connected neural network:



We keep softmax as the activation function on the last layer because that is what works best for classification. On intermediate layers however we will use the the most classical activation function: the sigmoid:



Your task in this section is to add one or two intermediate layers to your model to increase its performance.

To add a layer, you need an additional weights matrix and an additional bias vector for the intermediate layer:

*W1 = tf.Variable(tf.truncated_normal([28*28, 200] ,stddev=0.1))*
B1 = tf.Variable(tf.zeros([200]))

W2 = tf.Variable(tf.truncated_normal([200, 10], stddev=0.1))
B2 = tf.Variable(tf.zeros([10]))

The shape of the weights matrix for a layer is [N, M] where N is the number of inputs and M of outputs for the layer. In the code above, we use 200 neurons in the intermediate layer and still 10 neurons in the last layer.

*as you go deep, it becomes important to initialise weights with random values. The optimiser can get stuck in its initial position if you do not. tf.truncated_normal is a TensorFlow function that produces random values following the normal (Gaussian) distribution between -2*stddev and +2*stddev.*

And now change your 1-layer model into a 2-layer model:

*XX = tf.reshape(X, [-1, 28*28])*

Y1 = tf.nn.sigmoid(tf.matmul(XX, W1) + B1)
Y = tf.nn.softmax(tf.matmul(Y1, W2) + B2)

Don't forget to adjust your visualization methods. The following is the code for 5-layers network. If you have 2 or 3 layers you should adjust it respectively:

allweights = tf.concat([tf.reshape(W1, [-1]), tf.reshape(W2, [-1]), tf.reshape(W3, [-1]), tf.reshape(W4, [-1]), tf.reshape(W5, [-1])], 0)
allbiases = tf.concat([tf.reshape(B1, [-1]), tf.reshape(B2, [-1]), tf.reshape(B3, [-1]), tf.reshape(B4, [-1]), tf.reshape(B5, [-1])], 0)

That's it. You should now be able to push your network above 97% accuracy with 2 intermediate layer with for example 200 and 100 neurons.

What if we will add 5 layers? What if 7? Check it out.