In [ ]:

```
%matplotlib notebook

from matplotlib import pylab
pylab.rcParams['figure.figsize'] = (10.0, 10.0)

from tensorflow.examples.tutorials.mnist import input_data

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

Initialisations

Let us write some helper functions to initialise weights and biases. We'll initialise weights as Gaussian random variables with mean 0 and variance 0.0025. For biases we'll initialise everything with a constant 0.1. This is because we're mainly going to be using ReLU non-linearities.

In [ ]:

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.05)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

Model

Let's define the model. The model is defined as follows:

- An input that is 728 dimensional vector.
- Reshape the input as 28x28x1 images (only 1 because they are grey scale)
- A convolutional layer with 25 filters of shape 12x12x1 and a ReLU non-linearity (with stride (2, 2) and no padding)
- A convolutional layer with 64 filters of shape 5x5x25 and a ReLU non-linearity (with stride (1, 2) and padding to maintain size)
- A max_pooling layer of shape 2x2
- A fully connected layer taking all the outputs of the max_pooling layer to 1024 units and ReLU nonlinearity
- A fully connected layer taking 1024 units to 10 no activation function (the softmax non-linearity will be included in the loss function rather than in the model)

In [ ]:

```
x = tf.placeholder(tf.float32, shape=[None, 784])
x_ = tf.reshape(x, [-1, 28, 28, 1])
y_ = tf.placeholder(tf.float32, shape=[None, 10])

# Define the first convolution layer here
# TODO
# W_conv1 =
# b_conv1 =
# h_conv1 =

# Define the second convolution layer here
# W_conv2 =
# b_conv2 =
# h_conv2 =

# Define maxpooling
# h_pool2 =

# All subsequent layers will be fully connected ignoring geometry so we'll flatten the
 layer
# Flatten the h_pool2_layer (as it has a multidimensiona shape)
# h_pool2_flat =

# Define the first fully connected layer here
# W_fc1 =
# b_fc1 =
# h_fc1 =

# Use dropout for this layer (should you wish)
# keep_prob = tf.placeholder(tf.float32)
# h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# The final fully connected layer
# W_fc2 =
# b_fc2 =
# y_conv =
```

Loss Function, Accuracy and Training Algorithm

- We'll use the cross entropy loss function. The loss function is called `tf.nn.cross_entropy_with_logits` in tensorflow
- Accuray is simply defined as the fraction of data correctly classified
- For training you should use the AdamOptimizer (read the documentation) and set the learning rate to be 1e-4. You are welcome, and in fact encouraged, to experiment with other optimisation procedures and learning rates.
- (Optional): You may even want to use different filter sizes once you are finished with experimenting with what is asked in this practial

In [ ]:

```
# We'll use the cross entropy loss function
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y_conv, y_))

# And classification accuracy
correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# And the Adam optimiser
train_step = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(cross_entropy)
```

In [ ]:

```
# Load the mnist data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

In [ ]:

```
# Let us visualise the first 16 data points from the MNIST training data

fig = plt.figure()
for i in range(16):
    ax = fig.add_subplot(4, 4, i + 1)
    ax.set_xticks(())
    ax.set_yticks(())
    ax.imshow(mnist.train.images[i].reshape(28, 28), cmap='Greys_r')
```

In [ ]:

```
# Start a tf session and run the optimisation algorithm
sess = tf.Session()
sess.run(tf.initialize_all_variables())

for i in range(3000):
    batch = mnist.train.next_batch(50)
    #TODO
    # Write the optimisation code here
```

In [ ]:

```
# Print accuracy on the test set
# print ('Test accuracy: %g' % sess.run(accuracy, feed_dict={x: mnist.test.images, y_:
 mnist.test.labels, keep_prob: 1.0}))
```

Visualising the Filters

We'll now visualise all the 32 filters in the first convolution layer. As they are each of shape 12x12x1, they may themselves be viewed as greyscale images. Visualising filters in further layers is more complicated and involves modifying the neural network. See the paper (http://www.matthewzeiler.com/pubs/arxive2013/arxive2013.pdf) by Matt Zeiler and Rob Fergus if you are interested.

In [ ]:

```
# Visualise the filters in the first convolutional layer
with sess.as_default():
    W = W_conv1.eval()

# Add code to visualise fitlers here
```

Identifying image patches that activate the filters

For this part you'll find the 12 patches in the test-set that activate each of the first 5 filters that maximise the activation for that filter.

In [ ]:

```
H =  sess.run(h_conv1, feed_dict={x: mnist.test.images})

# Add code to visualise patches in the test set that find the most result in
# the highest activations for filters 0, ... 4
```