Practical 3 : Convolutional Neural Networks

In this practical, we'll build a convolutional neural network to classify handwritten digits. We'll also visualise the convolutional filters learnt in the first convolutional layer, as well as the image patches in the dataset that result in the strongest activation of these filters. For this practical we'll use the MNIST dataset and `tensorflow` to implement neural networks.

Brief Introduction to Tensorflow

You should begin by reading through the Deep MNIST for Experts tutorial on the tensorflow website. (Yes, you are experts by now!)

Go through the examples provided in this tutorial. Your actual task will be to build a somewhat different network and also visualise filters and image patches. It'd be helpful to quickly go through the examples provided in the tutorial, though you don't have to wait for the optimisation to complete.

Convolution and Maxpool in Tensorflow

Read how convolution is implemented in tensorflow here. You should understand how the size of the next layer is computed. Also understand what the difference between `VALID` and `SAME` padding is. You'll need to understand how the sizes are affected by strides, padding, etc. in order to implement the network properly.

Then read how the max_pool layer is implemented.

Visualising the MNIST data

Should you prefer to do so, you may fill in your code in the skeleton jupyter notebook provided on the course website. This notebook also shows how to visualise the MNIST data. In order to obtain the data you should run the following:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

As this will download the dataset, it may take some time. The training data is stored in `mnist.train.images` and the corresponding labels are in `mnist.train.labels`; similarly `mnist.validation.images` and `mnist.test.images`.

You can visualise the images using the `imshow` function in `matplotlib.pyplot`. The following code will result in something like this (see Fig. 1):

```
fig = plt.figure()
for i in range(16):
```

```
ax = fig.add_subplot(4, 4, i + 1)
ax.set_xlim(())
ax.set_ylim(())
ax.imshow(mnist.training.images[i].reshape(28, 28), cmap='Greys_r')
```

Convolutional Neural Network

You should now build a network with the following layers

- The inputs are provided as vectors of size 784, you should reshape them as 28x28x1 images (1 because these are grey scale)
- Add a convolutional layer with 25 filters of size 12x12x1 and the ReLU non-linearity. Use a stride of 2 in both directions and ensure that there is no padding.
- Add a second convolutional layer with 64 filters of size 5x5x25 that maintains the same width and height. Use stride of 1 in both directions and add padding as necessary and use the ReLU non-linearity.
- Add a max_pooling layer with pool size 2x2
- Add a fully connected layer with 1024 units. Each unit in the max_pool should be connected to these 1024 units. Add the ReLU non-linearity to these units.
- Add another fully connected layer to get 10 output units. Don't add any non-linearity to this layer, we'll implement the softmax non-linearity as part of the loss function.

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
- Accuracy 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 what is asked in this practial

Training

You should now train your neural network using minibatches of size 50. Try about 1000-5000 iterations. You may want to start out with fewer iterations to make sure your code is making good progress. Once you are sure your code is correct, you can let it run for more iterations and read up for the rest of the assignment while the code runs. Keep track of the validation accuracy every 100 iterations or so; however, do not touch the test dataset. Once you are sure your optimisation is working properly, you should run the resulting model on the test data and report the test error.

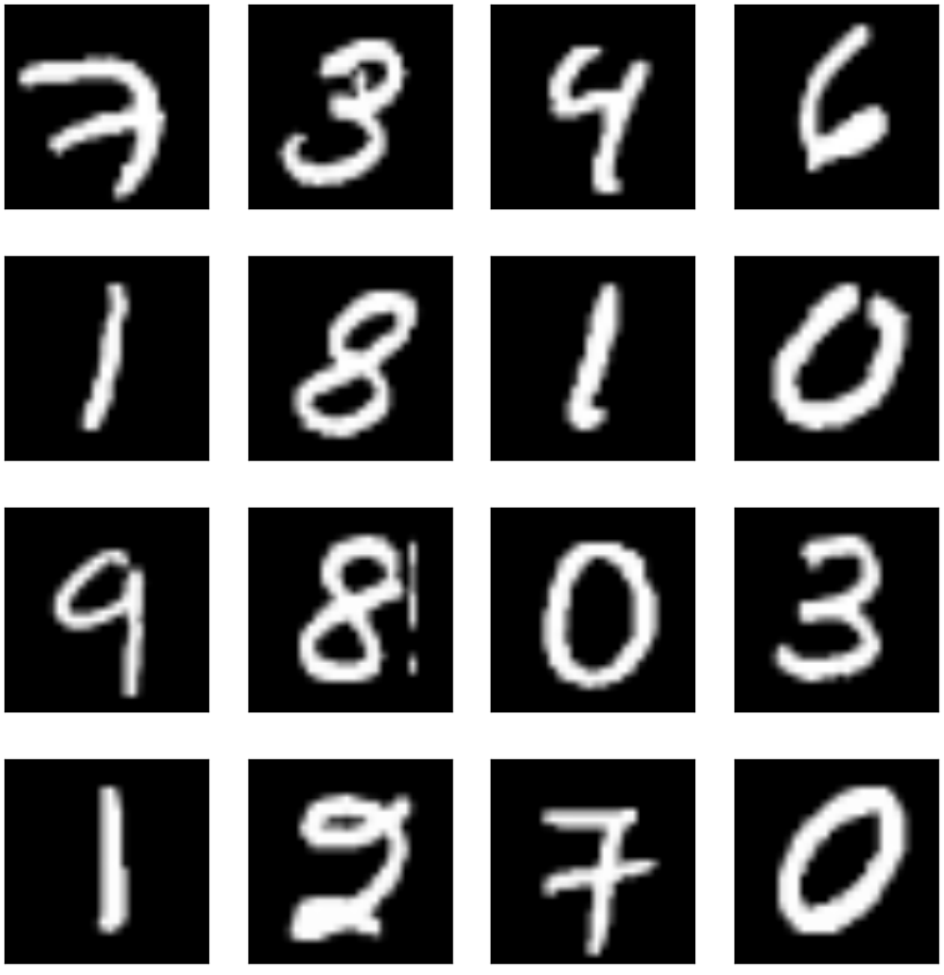

Figure 1: Digits

Visualising Filters

We'll now visualise the filters in the first convolutional layer. There are 25 filters each of size 12x12x1. Thus, the filters themselves can be viewed as 12x12 greyscale images. The size of the images in the dataset is relatively small, so don't expect these filters to have some obvious patterns. Nonetheless, it is helpful to visualise them. Every run will get a different set of filters, so you should not expect to see something exactly like the image shown in Fig. 2, but something along these lines.

In order to get the weights `W_conv1` for the first convolutional layer you can run the following code:

```
with sess.as_default():
    W = W_conv1.eval()
```

Visualising Patches with High Activation

The last task is to visualise the patches in the test dataset (10000 images) that result in the highest activation for each fitler. You can try as many filters as you like, but we recommend trying at least 5. The filters are all from the first convolutional layer.

Once you find the patch that results in the highest activation (these patches will be of size 12x12), display the top 12 patches for some random set of 5 filters in your model. You should expect to see something along the lines of what is shown in Fig. 3. The exact patterns will be different in each case.

To get the activations on the test data run the following:

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

You may find the following numpy functions useful:

```
np.argsort
np.unravel_index
```

Use `np.argsort` with `axis=None`.

Handin

- Report your code defining the model
- Report the accuracy on the test data (you should get about 98% accuracy)
- Report all the visualisations

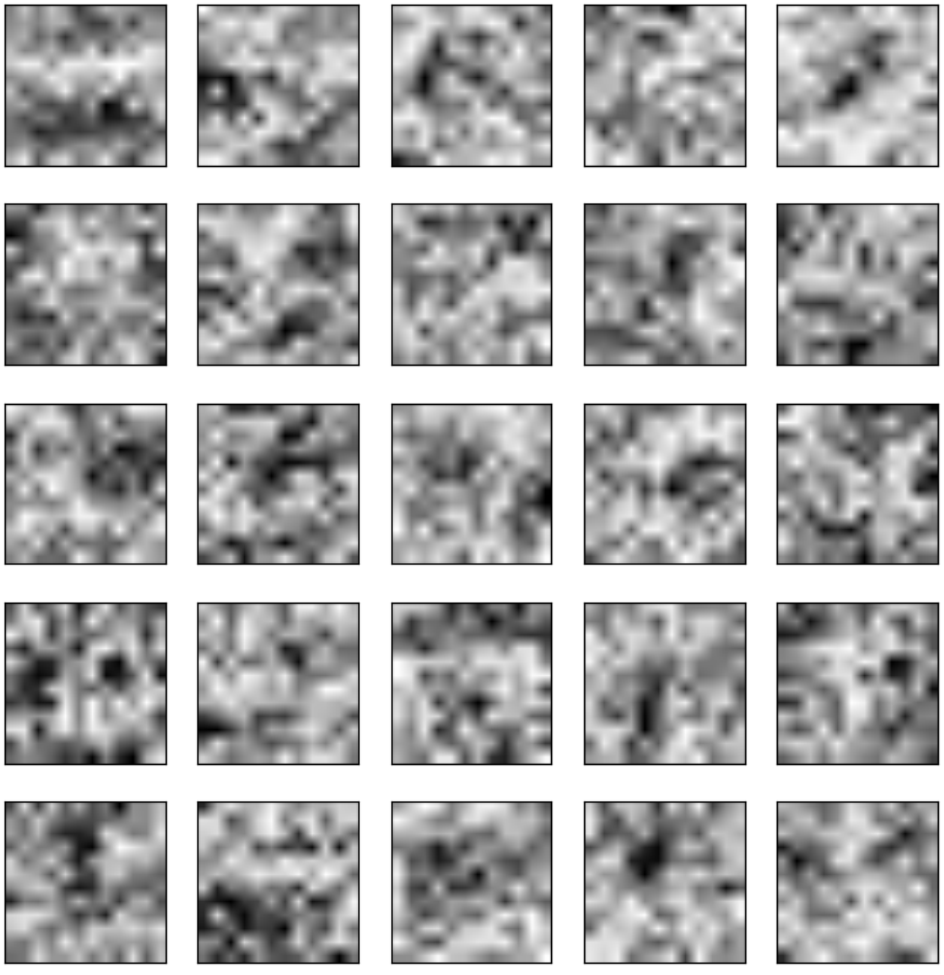

Figure 2: Filters

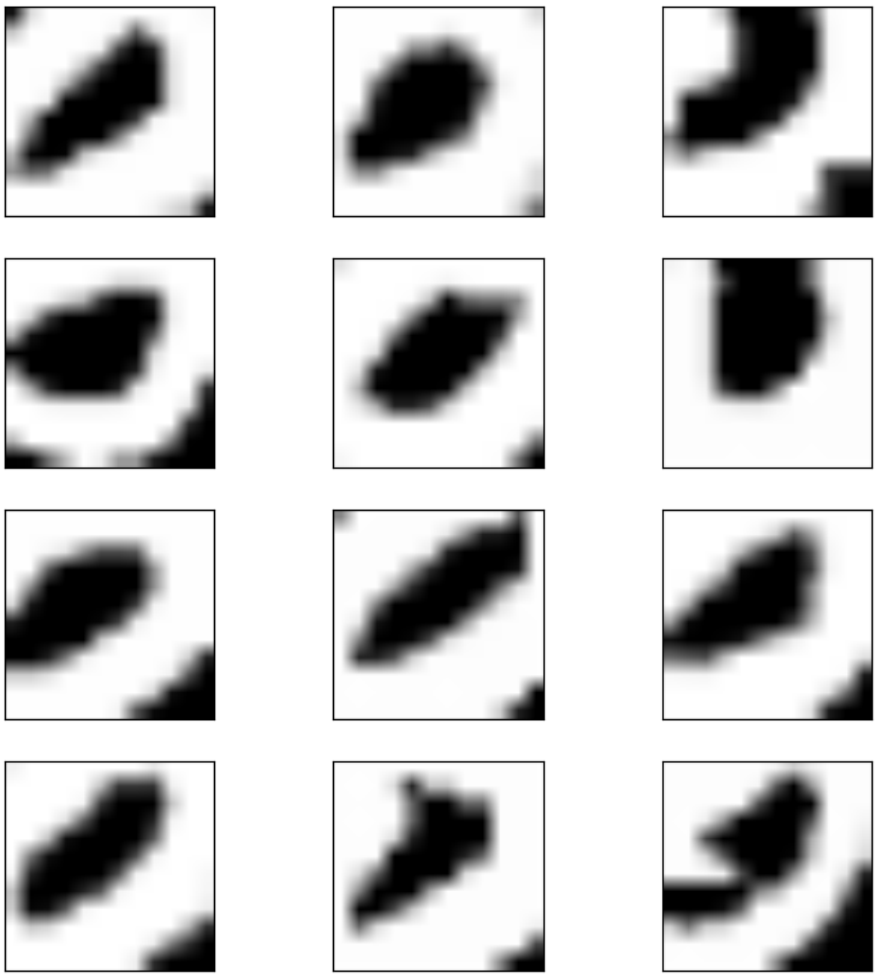

Figure 3: Patches