In [1]:

```
import keras
keras.__version__
```

```
Using TensorFlow backend.
```

Out[1]:

```
'2.2.4'
```

5.1 - Introduction to convnets

This notebook contains the code sample found in Chapter 5, Section 1 of [Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff)](https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff). Note that the original text features far more content, in particular further explanations and figures: in this notebook, you will only find source code and related comments.

First, let's take a practical look at a very simple convnet example. We will use our convnet to classify MNIST digits, a task that you've already been through in Chapter 2, using a densely-connected network (our test accuracy then was 97.8%). Even though our convnet will be very basic, its accuracy will still blow out of the water that of the densely-connected model from Chapter 2.

The 6 lines of code below show you what a basic convnet looks like. It's a stack of `Conv2D` and `MaxPooling2D` layers. We'll see in a minute what they do concretely. Importantly, a convnet takes as input tensors of shape `(image_height, image_width, image_channels)` (not including the batch dimension). In our case, we will configure our convnet to process inputs of size `(28, 28, 1)`, which is the format of MNIST images. We do this via passing the argument `input_shape=(28, 28, 1)` to our first layer.

In [2]:

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

```
WARNING:tensorflow:From /Users/jlee/anaconda3/lib/python3.7/site-packa
ges/tensorflow/python/framework/op_def_library.py:263: colocate_with (
from tensorflow.python.framework.ops) is deprecated and will be remove
d in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

Let's display the architecture of our convnet so far:

In [3]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928     
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
_________________________________________________________________
```

You can see above that the output of every `Conv2D` and `MaxPooling2D` layer is a 3D tensor of shape `(height, width, channels)`. The width and height dimensions tend to shrink as we go deeper in the network. The number of channels is controlled by the first argument passed to the `Conv2D` layers (e.g. 32 or 64).

The next step would be to feed our last output tensor (of shape `(3, 3, 64)`) into a densely-connected classifier network like those you are already familiar with: a stack of `Dense` layers. These classifiers process vectors, which are 1D, whereas our current output is a 3D tensor. So first, we will have to flatten our 3D outputs to 1D, and then add a few `Dense` layers on top:

In [4]:

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

We are going to do 10-way classification, so we use a final layer with 10 outputs and a softmax activation. Now here's what our network looks like:

In [5]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928     
_________________________________________________________________
flatten_1 (Flatten)          (None, 576)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 64)                36928     
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650       
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
```

As you can see, our `(3, 3, 64)` outputs were flattened into vectors of shape `(576,)`, before going through two `Dense` layers.

Now, let's train our convnet on the MNIST digits. We will reuse a lot of the code we have already covered in the MNIST example from Chapter 2.

In [6]:

```
from keras.datasets import mnist
from keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

In [23]:

```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_images, train_labels,
                    batch_size=64,
                    epochs=5,
                    validation_data=(test_images, test_labels)
                   )
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 72s 1ms/step - loss: 0.
0082 - acc: 0.9976 - val_loss: 0.0589 - val_acc: 0.9903
Epoch 2/5
60000/60000 [==============================] - 376s 6ms/step - loss: 0
.0087 - acc: 0.9978 - val_loss: 0.0423 - val_acc: 0.9927
Epoch 3/5
60000/60000 [==============================] - 70s 1ms/step - loss: 0.
0071 - acc: 0.9983 - val_loss: 0.0447 - val_acc: 0.9928
Epoch 4/5
60000/60000 [==============================] - 69s 1ms/step - loss: 0.
0068 - acc: 0.9983 - val_loss: 0.0471 - val_acc: 0.9913
Epoch 5/5
60000/60000 [==============================] - 72s 1ms/step - loss: 0.
0052 - acc: 0.9986 - val_loss: 0.0608 - val_acc: 0.9905
```

Let's evaluate the model on the test data:

In [26]:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
```

```
10000/10000 [==============================] - 5s 494us/step
```

In [27]:

```
test_acc
```

Out[27]:

```
0.9905
```

While our densely-connected networks had a test accuracy of 97.8%, our basic convnet has a test accuracy of 99.3%: we decreased our error rate by 68% (relative). Not bad!

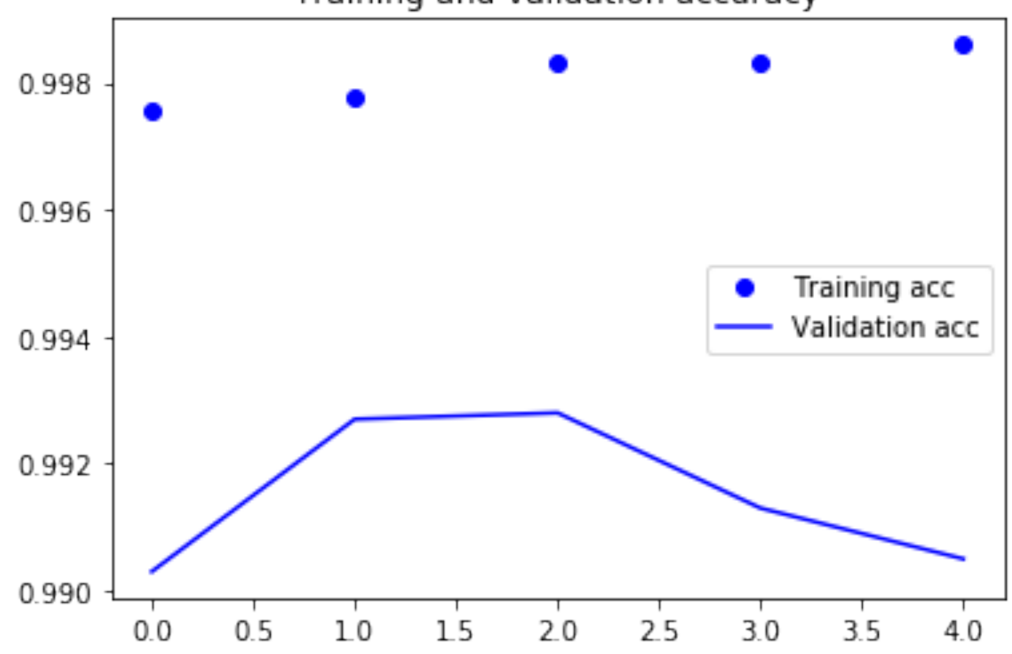
Problem 1

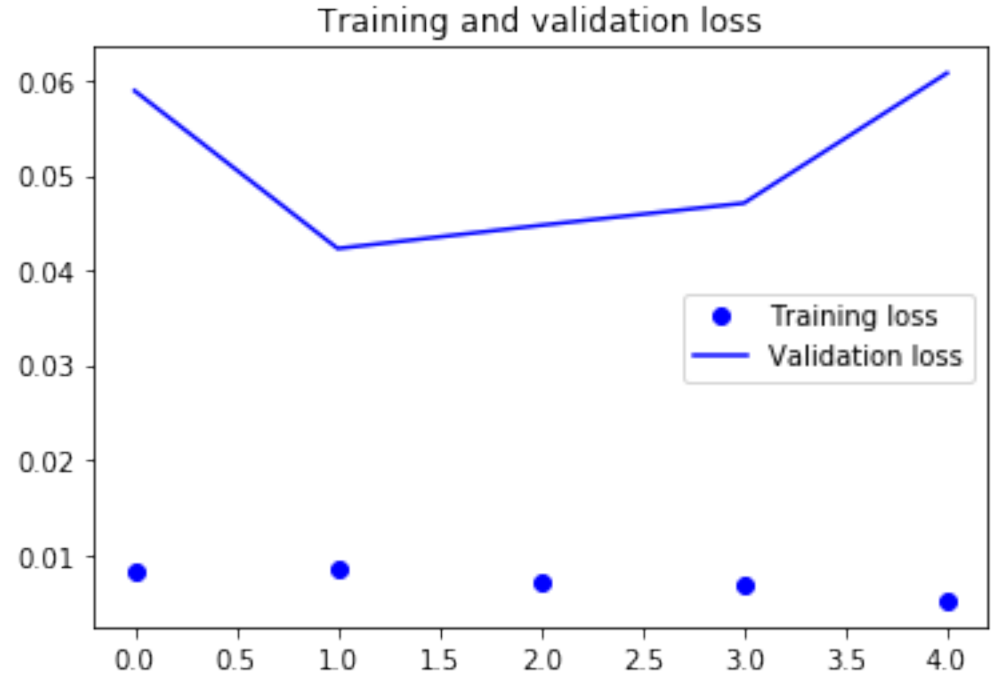
Execute all cells of the notebook 5.1-introduction-toconvnets.ipynb. Modify cell #7 and capture history object so that you could plot training and validation accuracy. Add a new cell and repeat training with L2 regularization. Use regularization parameter l=0.05 and l=0.01. Report on effect on overfitting, if any, and accuracy. Do not search for the optimal values for l and the number of epochs. Just report what you observe. Submit the Jupyter notebook 5.1_yourname.ipynb as well as the PDF image of that notebook. (10%)

In [25]:

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

Training and validation accuracy





Before any regularizers are added, we can see clear discrepancies between the training and validation accuracies and losses, the graphs show that the model is quite overfit.

In [36]:

```
from keras import regularizers

#Original Model
l2_model = models.Sequential()
l2_model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
l2_model.add(layers.MaxPooling2D((2, 2)))
l2_model.add(layers.Conv2D(64, (3, 3), activation='relu'))
l2_model.add(layers.MaxPooling2D((2, 2)))
l2_model.add(layers.Conv2D(64, (3, 3), activation='relu'))

#Add Level 2 Regularization
l2_model.add(layers.Dense(10, kernel_regularizer=regularizers.l2(0.05), activation=
l2_model.add(layers.Dense(10, kernel_regularizer=regularizers.l2(0.01), activation=

#Flatten and Compile
l2_model.add(layers.Flatten())
l2_model.add(layers.Dense(64, activation='relu'))
l2_model.add(layers.Dense(10, activation='softmax'))
l2_model.compile(optimizer='rmsprop',loss='categorical_crossentropy',metrics=['acc'
```

In [37]:

```
l2_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_13 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 3, 3, 64)          36928
_________________________________________________________________
dense_20 (Dense)             (None, 3, 3, 10)          650
_________________________________________________________________
dense_21 (Dense)             (None, 3, 3, 10)          110
_________________________________________________________________
flatten_5 (Flatten)          (None, 90)                0
_________________________________________________________________
dense_22 (Dense)             (None, 64)                5824
_________________________________________________________________
dense_23 (Dense)             (None, 10)                650
=================================================================
Total params: 62,978
Trainable params: 62,978
Non-trainable params: 0
_________________________________________________________________
```

In [38]:

```
l2_history = l2_model.fit(train_images, train_labels,
                    batch_size=64,
                    epochs=5,
                    validation_data=(test_images, test_labels)
                   )

test_loss, test_acc = l2_model.evaluate(test_images, test_labels)
test_acc
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 73s 1ms/step - loss: 0.
5229 - acc: 0.9025 - val_loss: 0.2153 - val_acc: 0.9582
Epoch 2/5
60000/60000 [==============================] - 70s 1ms/step - loss: 0.
1575 - acc: 0.9694 - val_loss: 0.1073 - val_acc: 0.9805
Epoch 3/5
60000/60000 [==============================] - 70s 1ms/step - loss: 0.
1061 - acc: 0.9784 - val_loss: 0.0781 - val_acc: 0.9839
Epoch 4/5
60000/60000 [==============================] - 70s 1ms/step - loss: 0.
0831 - acc: 0.9824 - val_loss: 0.0760 - val_acc: 0.9829
Epoch 5/5
60000/60000 [==============================] - 70s 1ms/step - loss: 0.
0699 - acc: 0.9846 - val_loss: 0.1190 - val_acc: 0.9693
10000/10000 [==============================] - 5s 519us/step
```

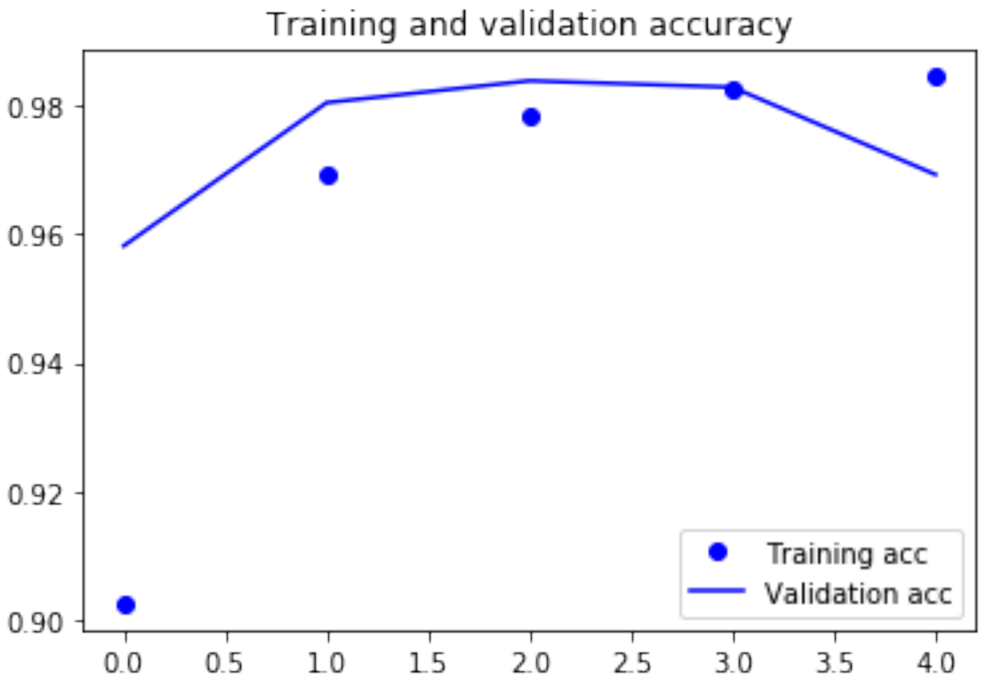
Out[38]:

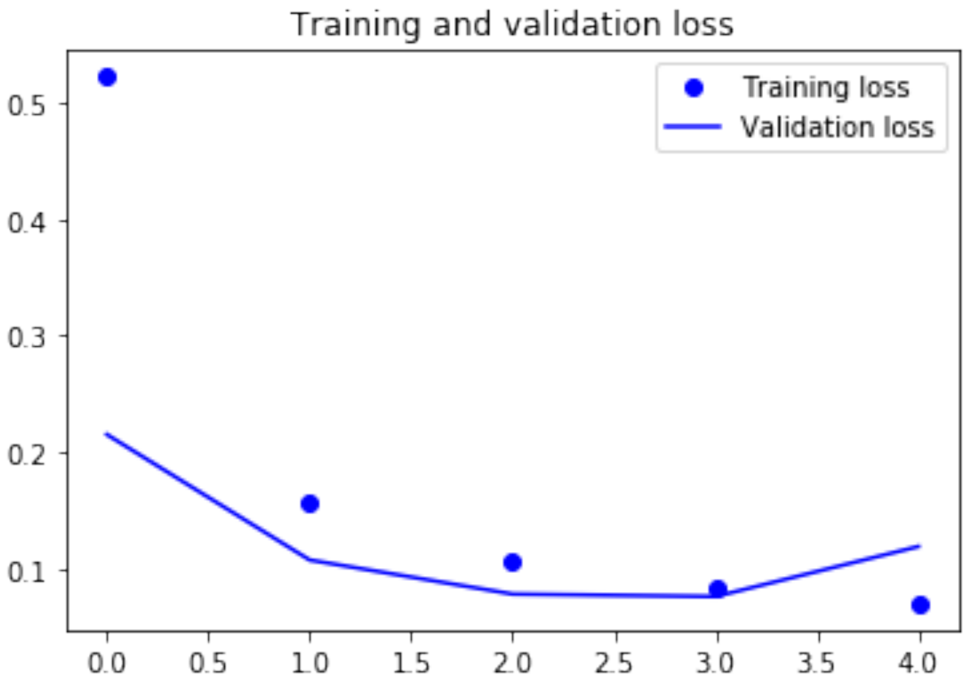
```
0.9693
```

In [39]:

```
l2_acc = l2_history.history['acc']
l2_val_acc = l2_history.history['val_acc']
l2_loss = l2_history.history['loss']
l2_val_loss = l2_history.history['val_loss']

l2_epochs = range(len(acc))

plt.plot(l2_epochs, l2_acc, 'bo', label='Training acc')
plt.plot(l2_epochs, l2_val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(l2_epochs, l2_loss, 'bo', label='Training loss')
plt.plot(l2_epochs, l2_val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
```

```
plt.show()
```





Interestingly enough, after adding additional L2 Regularization layers, the accuracy of the model goes down from 0.9905 to 0.9693. However, from what we can see in the graphs of the training and validation accuracy, the model is an overall better fit compared to the previous model as the two plots are much closer in comparison. Only after the third epoch, does there became a degree of increased loss / decreased accuracy (overfitting).

In [1]:

```
```

```
Using TensorFlow backend.
```

Out[1]:

```
'2.2.4'
```

5.1 - Introduction to convnets

This notebook contains the code sample found in Chapter 5, Section 1 of Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff). Note that the original text features far more content, in particular further explanations and figures: in this notebook, you will only find source code and related comments.

First, let's take a practical look at a very simple convnet example. We will use our convnet to classify MNIST digits, a task that you've already been through in Chapter 2, using a densely-connected network (our test accuracy then was 97.8%). Even though our convnet will be very basic, its accuracy will still blow out of the water that of the densely-connected model from Chapter 2.

The 6 lines of code below show you what a basic convnet looks like. It's a stack of `Conv2D` and `MaxPooling2D` layers. We'll see in a minute what they do concretely. Importantly, a convnet takes as input tensors of shape `(image_height, image_width, image_channels)` (not including the batch dimension). In our case, we will configure our convnet to process inputs of size `(28, 28, 1)`, which is the format of MNIST images. We do this via passing the argument `input_shape=(28, 28, 1)` to our first layer.

In [2]:

```
WARNING:tensorflow:From /Users/jlee/anaconda3/lib/python3.7/site-packa
ges/tensorflow/python/framework/op_def_library.py:263: colocate_with (
from tensorflow.python.framework.ops) is deprecated and will be remove
d in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

Let's display the architecture of our convnet so far:

In [3]:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928     
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
_________________________________________________________________
```

You can see above that the output of every `Conv2D` and `MaxPooling2D` layer is a 3D tensor of shape `(height, width, channels)`. The width and height dimensions tend to shrink as we go deeper in the network. The number of channels is controlled by the first argument passed to the `Conv2D` layers (e.g. 32 or 64).

The next step would be to feed our last output tensor (of shape `(3, 3, 64)`) into a densely-connected classifier network like those you are already familiar with: a stack of `Dense` layers. These classifiers process vectors, which are 1D, whereas our current output is a 3D tensor. So first, we will have to flatten our 3D outputs to 1D, and then add a few `Dense` layers on top:

In [4]:

We are going to do 10-way classification, so we use a final layer with 10 outputs and a softmax activation. Now here's what our network looks like:

In [5]:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928     
_________________________________________________________________
flatten_1 (Flatten)          (None, 576)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 64)                36928     
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650       
=================================================================
```

As you can see, our `(3, 3, 64)` outputs were flattened into vectors of shape `(576,)`, before going through two `Dense` layers.

Now, let's train our convnet on the MNIST digits. We will reuse a lot of the code we have already covered in the MNIST example from Chapter 2.

In [6]:

In [23]:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 72s 1ms/step - loss: 0.
0082 - acc: 0.9976 - val_loss: 0.0589 - val_acc: 0.9903
Epoch 2/5
60000/60000 [==============================] - 376s 6ms/step - loss: 0
.0087 - acc: 0.9978 - val_loss: 0.0423 - val_acc: 0.9927
Epoch 3/5
60000/60000 [==============================] - 70s 1ms/step - loss: 0.
0071 - acc: 0.9983 - val_loss: 0.0447 - val_acc: 0.9928
Epoch 4/5
60000/60000 [==============================] - 69s 1ms/step - loss: 0.
0068 - acc: 0.9983 - val_loss: 0.0471 - val_acc: 0.9913
Epoch 5/5
60000/60000 [==============================] - 72s 1ms/step - loss: 0.
0052 - acc: 0.9986 - val_loss: 0.0608 - val_acc: 0.9905
```

Let's evaluate the model on the test data:

In [26]:

```
10000/10000 [==============================] - 5s 494us/step
```

In [27]:

Out[27]:

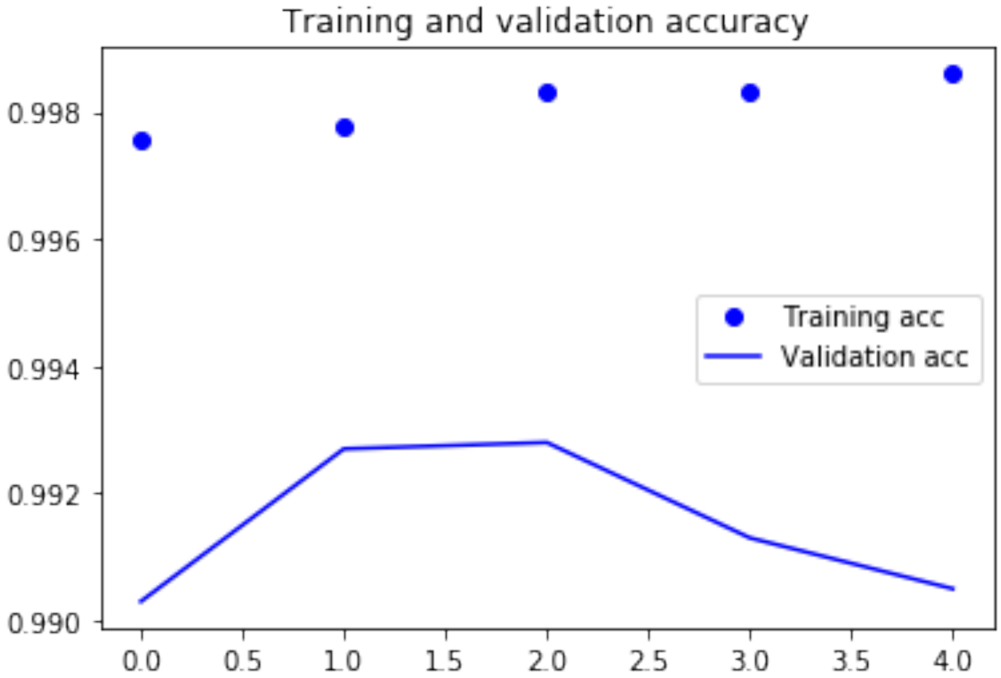
```
0.9905
```

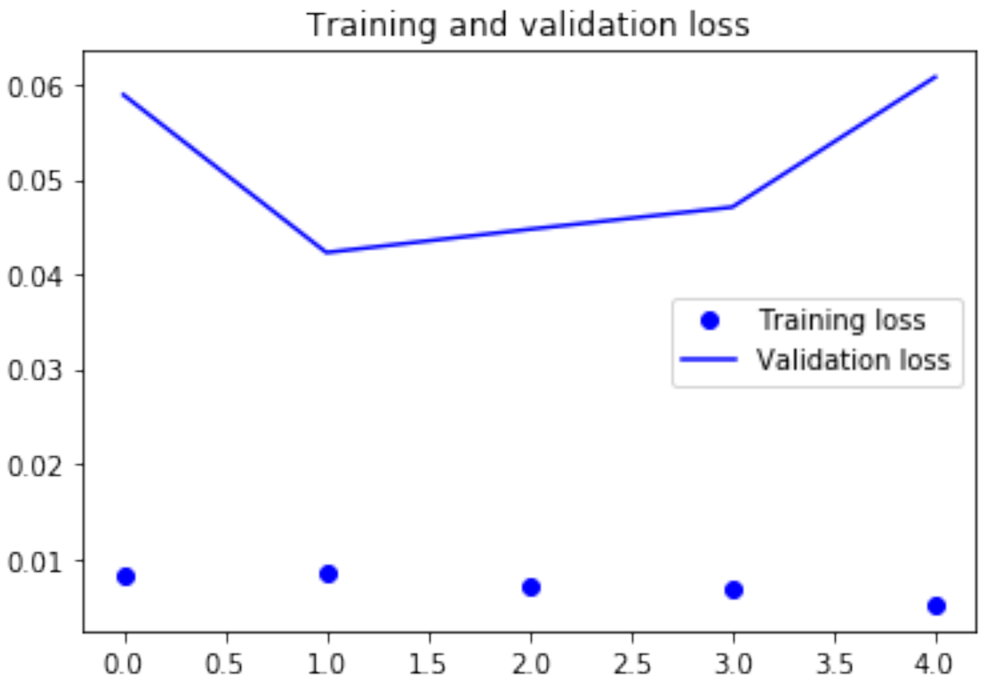
While our densely-connected networks had a test accuracy of 97.8%, our basic convnet has a test accuracy of 99.3%: we decreased our error rate by 68% (relative). Not bad!

Problem 1

Execute all cells of the notebook 5.1-introduction-toconvnets.ipynb. Modify cell #7 and capture history object so that you could plot training and validation accuracy. Add a new cell and repeat training with L2 regularization. Use regularization parameter l=0.05 and l=0.01. Report on effect on overfitting, if any, and accuracy. Do not search for the optimal values for l and the number of epochs. Just report what you observe. Submit the Jupyter notebook 5.1_yourname.ipynb as well as the PDF image of that notebook. (10%)

In [25]:





Before any regularizers are added, we can see clear discrepancies between the training and validation accuracies and losses, the graphs show that the model is quite overfit.

In [36]:

In [37]:

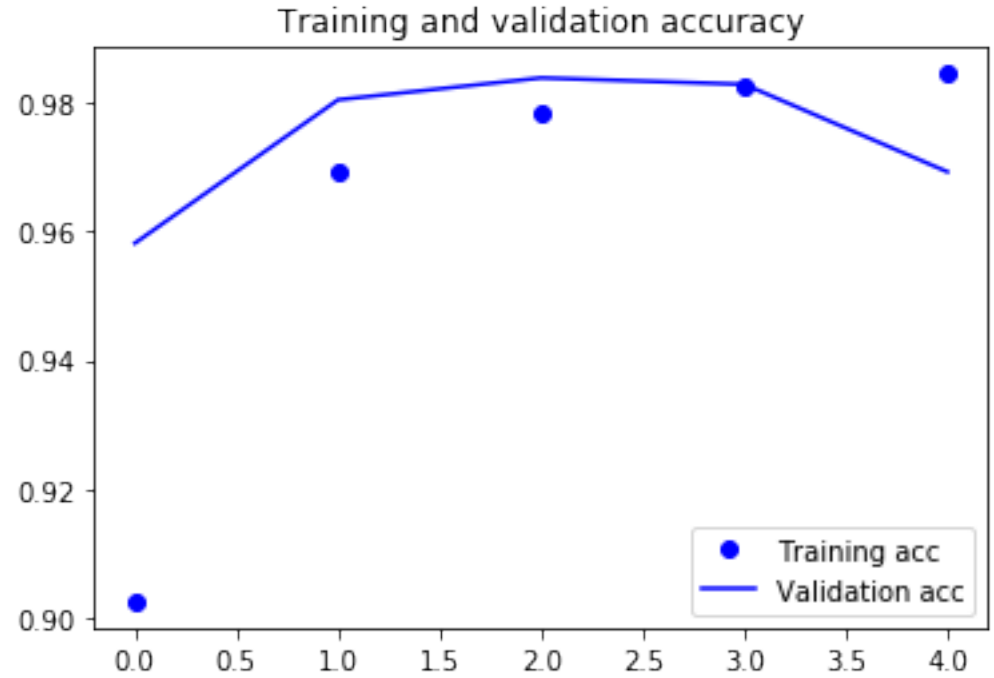
```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_13 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 3, 3, 64)          36928
_________________________________________________________________
dense_20 (Dense)             (None, 3, 3, 10)          650
_________________________________________________________________
dense_21 (Dense)             (None, 3, 3, 10)          110
_________________________________________________________________
flatten_5 (Flatten)          (None, 90)                0
_________________________________________________________________
```

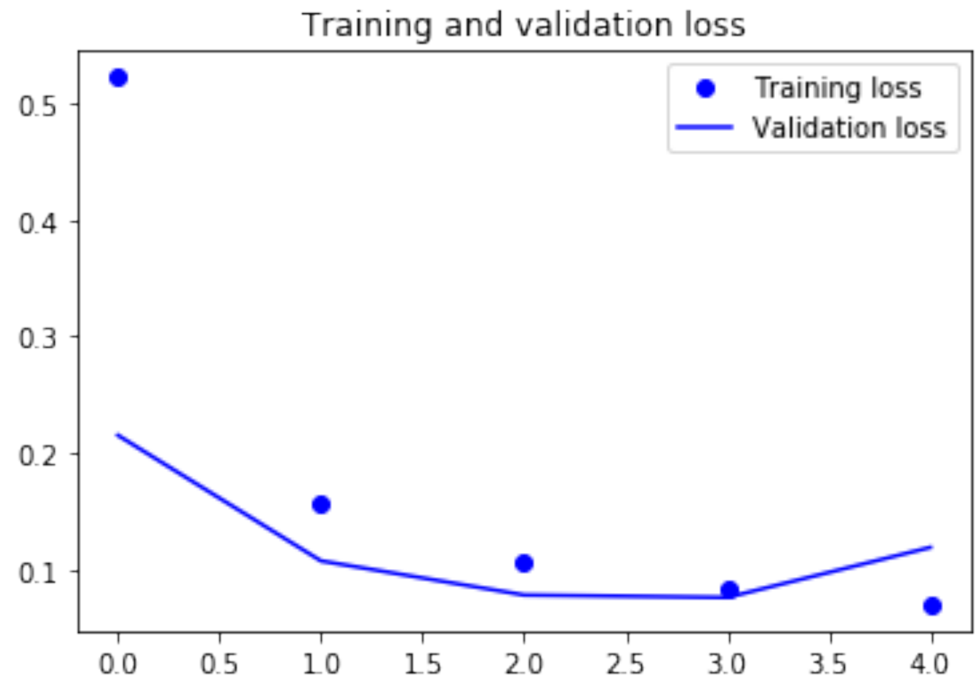
In [38]:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 73s 1ms/step - loss: 0.
5229 - acc: 0.9025 - val_loss: 0.2153 - val_acc: 0.9582
Epoch 2/5
60000/60000 [==============================] - 70s 1ms/step - loss: 0.
1575 - acc: 0.9694 - val_loss: 0.1073 - val_acc: 0.9805
Epoch 3/5
60000/60000 [==============================] - 70s 1ms/step - loss: 0.
1061 - acc: 0.9784 - val_loss: 0.0781 - val_acc: 0.9839
Epoch 4/5
60000/60000 [==============================] - 70s 1ms/step - loss: 0.
0831 - acc: 0.9824 - val_loss: 0.0760 - val_acc: 0.9829
Epoch 5/5
60000/60000 [==============================] - 70s 1ms/step - loss: 0.
0699 - acc: 0.9846 - val_loss: 0.1190 - val_acc: 0.9693
10000/10000 [==============================] - 5s 519us/step
```

Out[38]:

In [39]:





Interestingly enough, after adding additional L2 Regularization layers, the accuracy of the model goes down from 0.9905 to 0.9693. However, from what we can see in the graphs of the training and validation accuracy, after the third epoch, there became a degree of increased loss / decreased accuracy (overfitting).