DeepLearning1_Supervised_mbochenek

May 16, 2022

1 1000-719bMSB Modeling of Complex Biological Systems

2 Deep Neural Network: Supervised Learning

2.0.1 Basic python and pandas

https://www.kaggle.com/lavanyashukla01/pandas-numpy-python-cheatsheet

https://www.utc.fr/~jlaforet/Suppl/python-cheatsheets.pdf

List comprehensions are a concise way to create new lists from existing ones.

[1]:
```
list1 = list(range(0,10))
print(list1)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[2]:
```
list1[0] # A vector in R starts with an index of 1. In Python, 0.
```

[2]:
```
0
```

[3]:
```
list1[2:5]
```

[3]:
```
[2, 3, 4]
```

[4]:
```
list2 = []
for i in list1:
    list2.append(i+1)

print(list2)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

[5]:
```
list3 = [i+1 for i in list1]

print(list3)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

3 Classification of MNIST using densely connected layers

We are going to use the Keras library to implement a neural network that can classify handwritten digits - in just a few lines of code.

First we load and inspect the data. The dataset is split into training and test data.

[6]:
```
import numpy as np
import tensorflow.keras as keras
import matplotlib.pyplot as plt
```

[7]:
```
import tensorflow as tf
print(tf.__version__)
tf.compat.v1.disable_eager_execution()
```

```
2.8.0
```

[8]:
```
(train_images, train_labels), (test_images, test_labels) = keras.datasets.mnist.
↪load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

[9]:
```
train_images.shape
```

[9]: (60000, 28, 28)

[10]:
```
train_labels.shape
```

[10]: (60000,)

[11]:
```
test_images.shape
```

[11]: (10000, 28, 28)

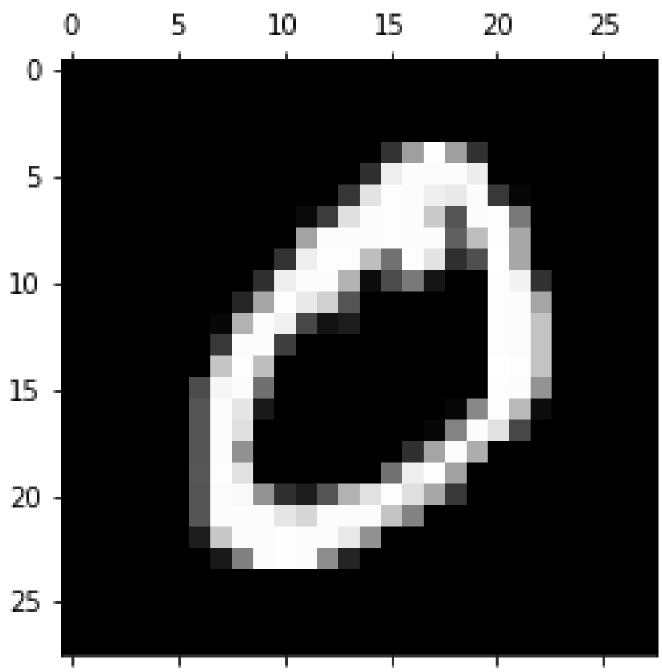
[12]:
```
test_labels.shape
```

[12]: (10000,)

Let's plot one of the digits and the corresponding label.

[16]:
```
print('Label of element 0:',train_labels[1])
plt.matshow(train_images[1], cmap='gray')
plt.show()
```

```
Label of element 0: 0
```



In this step we define the neural network. ReLu is an activation function defined as f(x) = max(0,x).

Softmax activation function is normalized such that the sum of all outputs is equal 1.

[17]:
```
from tensorflow.keras import layers
from tensorflow.keras import models
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
#units (here, 512) - Positive integer, dimensionality of the output space.
model.add(layers.Dense(10, activation='softmax'))
```

With compile we tell the network which optimizer and loss function to use. Optimizer specifies the particular implementation of the gradient-descent, e.g. how it adapts the learning rate. 'Metrics' specifies the output during the training.

[18]:
```
model.compile(optimizer='rmsprop',
loss='mean_squared_error',
metrics=['accuracy'])
```

[19]:
```
model.summary()
```

```
Model: "sequential"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 512)               401920
```

```
 dense_1 (Dense)             (None, 10)                5130

=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

We are using a densely connected network, so we have to flatten the images.
Input values should be in the range (0,1) for fast convergence.

[21]:
```python
#reshaping training dataset
train_images_flat = train_images.reshape((60000, 28 * 28))
#normalization - finding max value in the train dataset
maxValue = np.amax(train_images_flat)
print(maxValue)
#now normalizing
train_images_flat = train_images_flat.astype('float32') / 255

#reshaping testing dataset
test_images_flat = test_images.reshape((10000, 28 * 28))
#normalization - dividing by max value
test_images_flat = test_images_flat.astype('float32') / 255
```

```
255
```

Convert the labels to a 'one-hot' coding.

[22]:
```python
from tensorflow.keras.utils import to_categorical
print(f"Train labels before 'one hot coding': {train_labels[:10]}")
train_labels = to_categorical(train_labels)
print(f"Train labels after 'one hot coding': {train_labels[:10]}")
test_labels = to_categorical(test_labels)
```

```
Train labels before 'one hot coding': [5 0 4 1 9 2 1 3 1 4]
Train labels after 'one hot coding': [[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

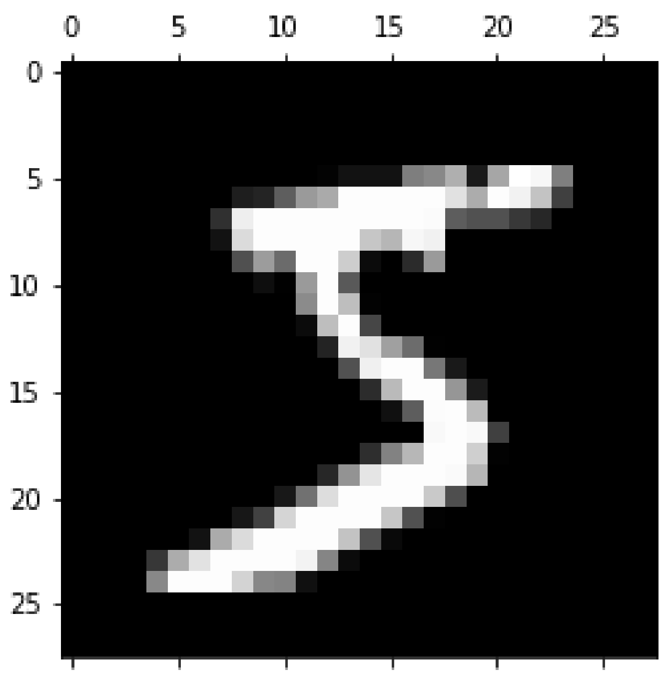
[23]:
```python
train_images.reshape((60000,28*28)).shape
```

[23]: (60000, 784)

[24]:
```
train_labels[0]
```

[24]: array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)

[25]:
```
plt.matshow(train_images[0], cmap='gray')
plt.show()
```



[26]:
```
model.fit(train_images_flat, train_labels, epochs=5, batch_size=128)
```

```
Train on 60000 samples
Epoch 1/5
60000/60000 [==============================] - 4s 61us/sample - loss: 0.0121 -
accuracy: 0.9226
Epoch 2/5
60000/60000 [==============================] - 1s 19us/sample - loss: 0.0053 -
accuracy: 0.9671
Epoch 3/5
60000/60000 [==============================] - 1s 19us/sample - loss: 0.0037 -
accuracy: 0.9771
Epoch 4/5
60000/60000 [==============================] - 1s 19us/sample - loss: 0.0028 -
accuracy: 0.9826
Epoch 5/5
```

```
60000/60000 [==============================] - 1s 19us/sample - loss: 0.0022 -
accuracy: 0.9861
```

[26]:
```
<keras.callbacks.History at 0x7f8633576150>
```

Let's check the performance on the test set. If the accuracy is less than the training accuracy, then we might be overfitting!

[27]:
```python
test_loss, test_acc = model.evaluate(test_images_flat, test_labels)
print('test_acc:', test_acc)
```

```
/usr/local/lib/python3.7/dist-packages/keras/engine/training_v1.py:2057:
UserWarning: `Model.state_updates` will be removed in a future version. This
property should not be used in TensorFlow 2.0, as `updates` are applied
automatically.
  updates = self.state_updates
```

```
test_acc: 0.9805
```

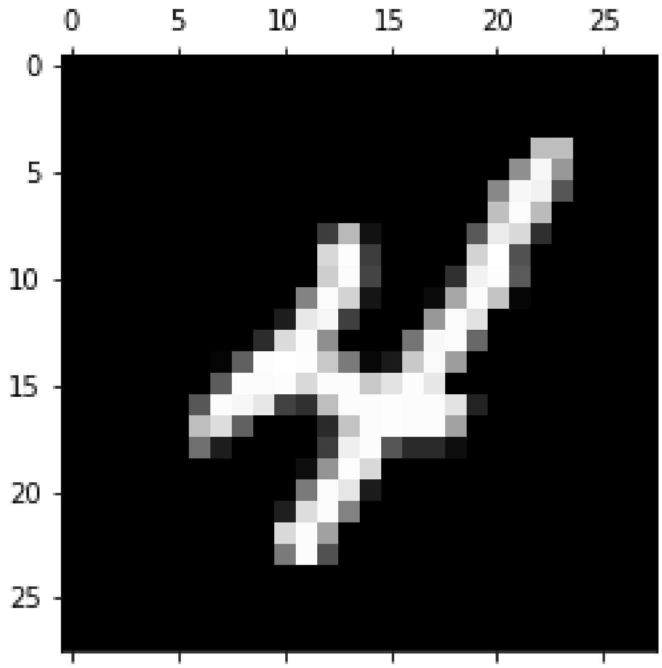
We can also find the predictions for a selection of input images.

[28]:
```python
predictions = model.predict(train_images_flat[:10])
```

```
/usr/local/lib/python3.7/dist-packages/keras/engine/training_v1.py:2079:
UserWarning: `Model.state_updates` will be removed in a future version. This
property should not be used in TensorFlow 2.0, as `updates` are applied
automatically.
  updates=self.state_updates,
```

[31]:
```python
img_num = 9
print(predictions[img_num])
print(train_labels[img_num])
plt.matshow(train_images[img_num], cmap='gray')
plt.show()
```

```
[1.8075972e-06 3.0627989e-09 2.8879535e-06 9.8233635e-08 9.9992228e-01
 7.7069035e-06 6.0323819e-05 4.0500558e-06 3.0447688e-07 5.6681574e-07]
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
```



[32]:
```
train_labels
```

[32]:
```
array([[0., 0., 0., …, 0., 0., 0.],
       [1., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       …,
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 1., 0.]], dtype=float32)
```

4 Classification of MNIST using convolutional layers

We have build a classifier for handwritten images only using densely connected layers. Let's see if we can do better using convolutional layers!

First define the convolutional layers.

[33]:
```
model2 = models.Sequential()
model2.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28,␣
 ↪1)))
model2.add(layers.MaxPooling2D((2, 2)))
model2.add(layers.Conv2D(64, (3, 3), activation='relu'))
model2.add(layers.MaxPooling2D((2, 2)))
model2.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

[34]:
```
model2.summary()
```

```
Model: "sequential_1"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 64)          36928

=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
_________________________________________________________________
```

Now add a classifier on top of the convnet.

[35]:
```
model2.add(layers.Flatten()) #flattening layer
model2.add(layers.Dense(64, activation='relu')) #dense layer
model2.add(layers.Dense(10, activation='softmax')) #dense layer
```

[36]:
```
model2.summary()
```

```
Model: "sequential_1"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 64)          36928
```

```
 flatten (Flatten)           (None, 576)               0

 dense_2 (Dense)             (None, 64)                36928

 dense_3 (Dense)             (None, 10)                650

=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
```

[37]:

```
train_images_conv = train_images.reshape((60000, 28, 28, 1))
train_images_conv = train_images_conv.astype('float32') / 255
test_images_conv = test_images.reshape((10000, 28, 28, 1))
test_images_conv = test_images_conv.astype('float32') / 255
```

[38]:

```
model2.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

[39]:

```
model2.fit(train_images_conv, train_labels, epochs=15, batch_size=64)
```

```
Train on 60000 samples
Epoch 1/15
60000/60000 [==============================] - 13s 220us/sample - loss: 0.1779 -
accuracy: 0.9438
Epoch 2/15
60000/60000 [==============================] - 4s 74us/sample - loss: 0.0479 -
accuracy: 0.9851
Epoch 3/15
60000/60000 [==============================] - 4s 64us/sample - loss: 0.0330 -
accuracy: 0.9899
Epoch 4/15
60000/60000 [==============================] - 3s 57us/sample - loss: 0.0253 -
accuracy: 0.9921
Epoch 5/15
60000/60000 [==============================] - 4s 74us/sample - loss: 0.0193 -
accuracy: 0.9941
Epoch 6/15
60000/60000 [==============================] - 4s 66us/sample - loss: 0.0158 -
accuracy: 0.9950
Epoch 7/15
60000/60000 [==============================] - 4s 68us/sample - loss: 0.0132 -
accuracy: 0.9961
Epoch 8/15
60000/60000 [==============================] - 4s 61us/sample - loss: 0.0098 -
accuracy: 0.9969
```

```
Epoch 9/15
60000/60000 [==============================] - 4s 60us/sample - loss: 0.0099 -
accuracy: 0.9971
Epoch 10/15
60000/60000 [==============================] - 4s 61us/sample - loss: 0.0086 -
accuracy: 0.9978
Epoch 11/15
60000/60000 [==============================] - 4s 60us/sample - loss: 0.0069 -
accuracy: 0.9980
Epoch 12/15
60000/60000 [==============================] - 4s 65us/sample - loss: 0.0069 -
accuracy: 0.9981
Epoch 13/15
60000/60000 [==============================] - 4s 64us/sample - loss: 0.0058 -
accuracy: 0.9984
Epoch 14/15
60000/60000 [==============================] - 3s 57us/sample - loss: 0.0049 -
accuracy: 0.9987
Epoch 15/15
60000/60000 [==============================] - 4s 59us/sample - loss: 0.0041 -
accuracy: 0.9989
```

[39]:
```
<keras.callbacks.History at 0x7f85b45d6f10>
```

[40]:
```python
test_loss, test_acc = model2.evaluate(test_images_conv, test_labels)
print(f"Accuracy on the test dataset = {test_acc}")
```

```
/usr/local/lib/python3.7/dist-packages/keras/engine/training_v1.py:2057:
UserWarning: `Model.state_updates` will be removed in a future version. This
property should not be used in TensorFlow 2.0, as `updates` are applied
automatically.
  updates = self.state_updates
```

```
Accuracy on the test dataset = 0.9925000071525574
```

4.1 Introducing Fashion MNIST (Homework dataset)

The MNIST dataset is not too demanding, let's try something a little more difficult - Fashion MNIST.

LINK TO IMAGE

Check out labels on GitHub

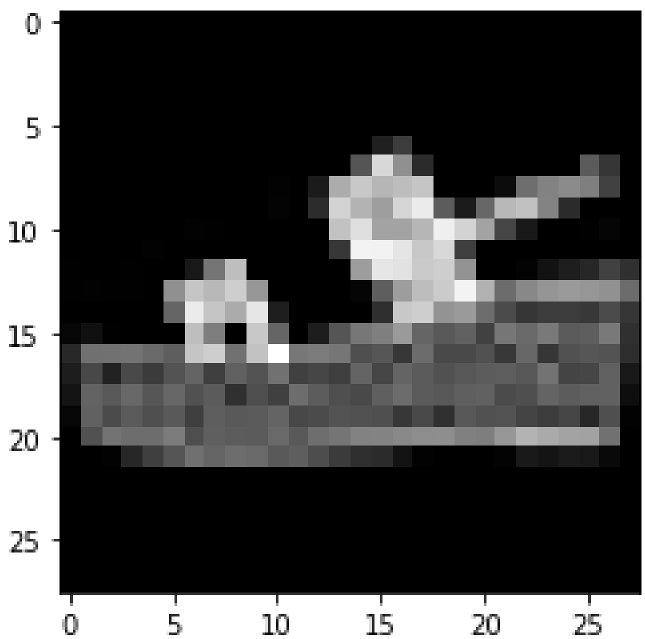
[63]:
```python
(train_imgs_fash, train_labels_fash), (test_imgs_fash, test_labels_fash) =␣
↪keras.datasets.fashion_mnist.load_data()
```

[47]:
```
print(train_imgs_fash.shape)
print(test_imgs_fash.shape)
```

```
(60000, 28, 28)
(10000, 28, 28)
```

[48]:
```
plt.imshow(train_imgs_fash[12],  cmap=plt.get_cmap('gray'))
```

[48]: <matplotlib.image.AxesImage at 0x7f85b43a5d10>



5 HOMEWORK 1

Build a classifier for fashion MNIST.

1. Use exactly the same architectures (both densely connected layers and from convolutional layers) as the above MNIST e.g., replace the dataset. Save the Jupyter Notebook in its original format and output a PDF file after training, testing, and validation. Make sure to write down how do they perform (training accuracny, testing accuracy).
2. Improve the architecture. Experiment with different numbers of layers, size of layers, number of filters, size of filters. You are required to make those adjustment to get the highest accuracy. Watch out for overfitting -- we want the highest testing accuracy! Please provide a PDF file of the result, the best test accuracy and the architecture (different numbers of layers, size of layers, number of filters, size of filters)

5.0.1 Data preprocessing

[64]:
```
#reshaping
train_imgs_fash = train_imgs_fash.reshape((60000, 28, 28, 1))
test_imgs_fash = test_imgs_fash.reshape((10000, 28, 28, 1))

#normalizing data
#finding max value in the dataset
maxValue = np.amax(train_imgs_fash)
print(f"max value = {maxValue}")

train_imgs_fash = train_imgs_fash.astype('float32') / maxValue
test_imgs_fash = test_imgs_fash.astype('float32') / maxValue

#encoding labels
print(f"Train labels before 'one hot coding': {train_labels_fash[:10]}")
train_labels_fash = to_categorical(train_labels_fash)
print(f"Train labels after 'one hot coding': {train_labels_fash[:10]}")
test_labels_fash = to_categorical(test_labels_fash)
```

```
max value = 255
Train labels before 'one hot coding': [9 0 0 3 0 2 7 2 5 5]
Train labels after 'one hot coding': [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]]
```

5.0.2 First model creation

[53]:
```
#creating the model
model_fash1 = models.Sequential()
model_fash1.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28,␣
 ↪28, 1)))
model_fash1.add(layers.MaxPooling2D((2, 2)))
model_fash1.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_fash1.add(layers.MaxPooling2D((2, 2)))
model_fash1.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_fash1.add(layers.Flatten()) #flattening layer
model_fash1.add(layers.Dense(64, activation='relu')) #dense layer
model_fash1.add(layers.Dense(10, activation='softmax')) #dense layer
```

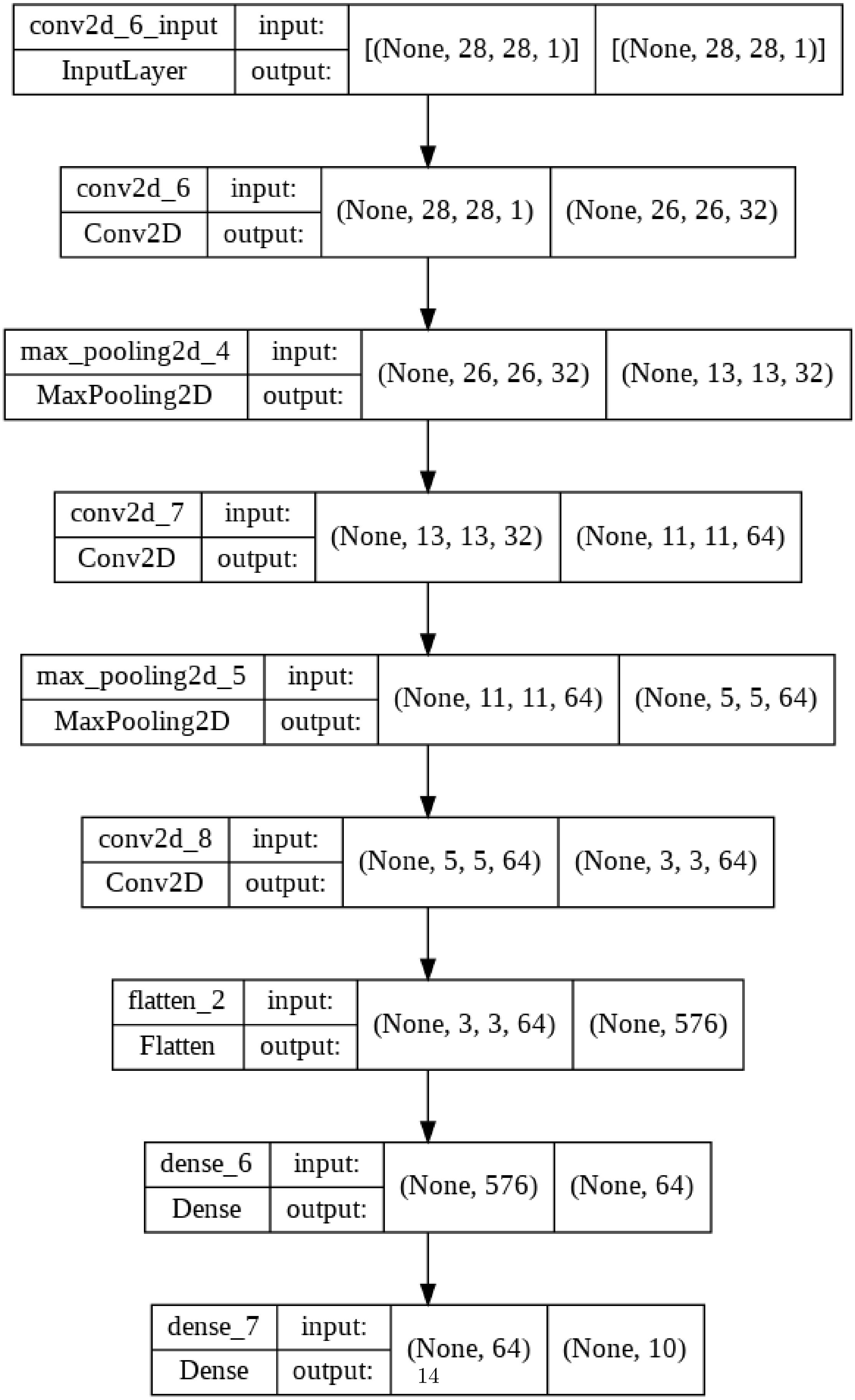
```python
#compiling
model_fash1.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

[59]:
```python
model_fash1.summary()
```

```
Model: "sequential_3"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 26, 26, 32)        320

 max_pooling2d_4 (MaxPooling  (None, 13, 13, 32)       0
 2D)

 conv2d_7 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_5 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 conv2d_8 (Conv2D)           (None, 3, 3, 64)          36928

 flatten_2 (Flatten)         (None, 576)               0

 dense_6 (Dense)             (None, 64)                36928

 dense_7 (Dense)             (None, 10)                650

=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
```

[147]:
```python
tf.keras.utils.plot_model(model_fash1, 'model_fash1.png', show_shapes=True)
```

[147]:


conv2d_6_input
InputLayer
input:
output:
[(None, 28, 28, 1)]
[(None, 28, 28, 1)]
conv2d_6
Conv2D
input:
output:
(None, 28, 28, 1)
(None, 26, 26, 32)
max_pooling2d_4
MaxPooling2D
input:
output:
(None, 26, 26, 32)
(None, 13, 13, 32)
conv2d_7
Conv2D
input:
output:
(None, 13, 13, 32)
(None, 11, 11, 64)
max_pooling2d_5
MaxPooling2D
input:
output:
(None, 11, 11, 64)
(None, 5, 5, 64)
conv2d_8
Conv2D
input:
output:
(None, 5, 5, 64)
(None, 3, 3, 64)
flatten_2
Flatten
input:
output:
(None, 3, 3, 64)
(None, 576)
dense_6
Dense
input:
output:
(None, 576)
(None, 64)
dense_7
Dense
input:
output:
(None, 64)
(None, 10)

[54]:
```
model_fash1.fit(train_imgs_fash, train_labels_fash, epochs=15, batch_size=64)
```

```
Train on 60000 samples
Epoch 1/15
60000/60000 [==============================] - 4s 62us/sample - loss: 0.5396 -
accuracy: 0.8001
Epoch 2/15
60000/60000 [==============================] - 4s 62us/sample - loss: 0.3333 -
accuracy: 0.8794
Epoch 3/15
60000/60000 [==============================] - 3s 58us/sample - loss: 0.2788 -
accuracy: 0.8985
Epoch 4/15
60000/60000 [==============================] - 3s 58us/sample - loss: 0.2480 -
accuracy: 0.9093
Epoch 5/15
60000/60000 [==============================] - 3s 58us/sample - loss: 0.2249 -
accuracy: 0.9168
Epoch 6/15
60000/60000 [==============================] - 3s 58us/sample - loss: 0.2072 -
accuracy: 0.9239
Epoch 7/15
60000/60000 [==============================] - 4s 59us/sample - loss: 0.1902 -
accuracy: 0.9300
Epoch 8/15
60000/60000 [==============================] - 4s 59us/sample - loss: 0.1760 -
accuracy: 0.9355
Epoch 9/15
60000/60000 [==============================] - 4s 58us/sample - loss: 0.1645 -
accuracy: 0.9396
Epoch 10/15
60000/60000 [==============================] - 3s 58us/sample - loss: 0.1530 -
accuracy: 0.9427
Epoch 11/15
60000/60000 [==============================] - 3s 57us/sample - loss: 0.1429 -
accuracy: 0.9479
Epoch 12/15
60000/60000 [==============================] - 3s 57us/sample - loss: 0.1337 -
accuracy: 0.9507
Epoch 13/15
60000/60000 [==============================] - 3s 58us/sample - loss: 0.1256 -
accuracy: 0.9536
Epoch 14/15
60000/60000 [==============================] - 3s 58us/sample - loss: 0.1179 -
accuracy: 0.9563
```

```
Epoch 15/15
60000/60000 [==============================] - 3s 58us/sample - loss: 0.1118 -
accuracy: 0.9585
```

[54]:
```
<keras.callbacks.History at 0x7f85b4212a90>
```

[55]:
```
test_loss, test_acc = model_fash1.evaluate(test_imgs_fash, test_labels_fash)
print(f"Accuracy on the test dataset = {test_acc}")
```

```
/usr/local/lib/python3.7/dist-packages/keras/engine/training_v1.py:2057:
UserWarning: `Model.state_updates` will be removed in a future version. This
property should not be used in TensorFlow 2.0, as `updates` are applied
automatically.
  updates = self.state_updates
```

```
Accuracy on the test dataset = 0.9110000133514404
```

5.0.3 Improve the architecture

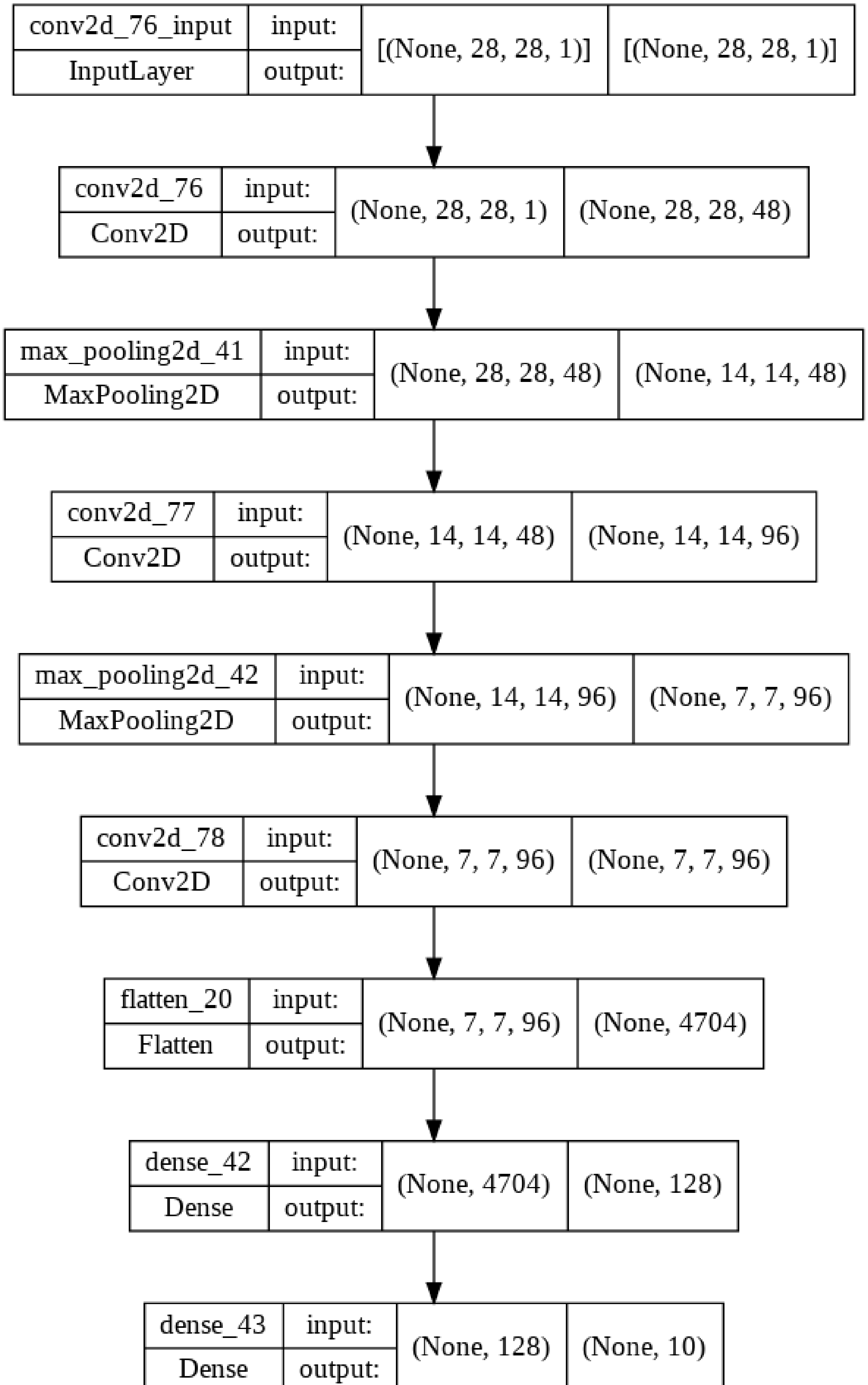
[127]:
```
#creating the model
model_fash2 = models.Sequential()
model_fash2.add(layers.Conv2D(48, (3, 3), padding='same', activation='relu',␣
 ↪input_shape=(28, 28, 1)))
model_fash2.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2,
                                    padding='valid'))
model_fash2.add(layers.Conv2D(96, (3, 3), padding='same', activation='relu'))
model_fash2.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2,
                                    padding='valid'))
model_fash2.add(layers.Conv2D(96, (3, 3), padding='same', activation='relu'))

model_fash2.add(layers.Flatten()) #flattening layer
model_fash2.add(layers.Dense(128, activation='relu')) #dense layer
model_fash2.add(layers.Dense(10, activation='softmax')) #dense layer

#compiling
model_fash2.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])

model_fash2.summary()
```

```
Model: "sequential_21"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_76 (Conv2D)          (None, 28, 28, 48)        480
```

```
 max_pooling2d_41 (MaxPoolin  (None, 14, 14, 48)       0
 g2D)

 conv2d_77 (Conv2D)          (None, 14, 14, 96)        41568

 max_pooling2d_42 (MaxPoolin  (None, 7, 7, 96)         0
 g2D)

 conv2d_78 (Conv2D)          (None, 7, 7, 96)          83040

 flatten_20 (Flatten)        (None, 4704)              0

 dense_42 (Dense)            (None, 128)               602240

 dense_43 (Dense)            (None, 10)                1290

=================================================================
Total params: 728,618
Trainable params: 728,618
Non-trainable params: 0
_________________________________________________________________
```

[148]:
```
tf.keras.utils.plot_model(model_fash2, 'model_fash2.png', show_shapes=True)
```

[148]:


conv2d_76_input
InputLayer
input:
output:
[(None, 28, 28, 1)]
[(None, 28, 28, 1)]
conv2d_76
Conv2D
input:
output:
(None, 28, 28, 1)
(None, 28, 28, 48)
max_pooling2d_41
MaxPooling2D
input:
output:
(None, 28, 28, 48)
(None, 14, 14, 48)
conv2d_77
Conv2D
input:
output:
(None, 14, 14, 48)
(None, 14, 14, 96)
max_pooling2d_42
MaxPooling2D
input:
output:
(None, 14, 14, 96)
(None, 7, 7, 96)
conv2d_78
Conv2D
input:
output:
(None, 7, 7, 96)
(None, 7, 7, 96)
flatten_20
Flatten
input:
output:
(None, 7, 7, 96)
(None, 4704)
dense_42
Dense
input:
output:
(None, 4704)
(None, 128)
dense_43
Dense
input:
output:
(None, 128)
(None, 10)

[128]:
```
model_fash2.fit(train_imgs_fash, train_labels_fash, epochs=15, batch_size=64)
```

```
Train on 60000 samples
Epoch 1/15
60000/60000 [==============================] - 5s 86us/sample - loss: 0.4165 -
accuracy: 0.8475
Epoch 2/15
60000/60000 [==============================] - 5s 81us/sample - loss: 0.2543 -
accuracy: 0.9079
Epoch 3/15
60000/60000 [==============================] - 5s 82us/sample - loss: 0.2126 -
accuracy: 0.9216
Epoch 4/15
60000/60000 [==============================] - 5s 82us/sample - loss: 0.1840 -
accuracy: 0.9335
Epoch 5/15
60000/60000 [==============================] - 5s 82us/sample - loss: 0.1622 -
accuracy: 0.9407
Epoch 6/15
60000/60000 [==============================] - 5s 82us/sample - loss: 0.1458 -
accuracy: 0.9474
Epoch 7/15
60000/60000 [==============================] - 5s 82us/sample - loss: 0.1315 -
accuracy: 0.9532
Epoch 8/15
60000/60000 [==============================] - 5s 82us/sample - loss: 0.1223 -
accuracy: 0.9574
Epoch 9/15
60000/60000 [==============================] - 5s 81us/sample - loss: 0.1139 -
accuracy: 0.9599
Epoch 10/15
60000/60000 [==============================] - 5s 83us/sample - loss: 0.1042 -
accuracy: 0.9628
Epoch 11/15
60000/60000 [==============================] - 5s 83us/sample - loss: 0.0975 -
accuracy: 0.9657
Epoch 12/15
60000/60000 [==============================] - 5s 81us/sample - loss: 0.0946 -
accuracy: 0.9677
Epoch 13/15
60000/60000 [==============================] - 5s 81us/sample - loss: 0.0911 -
accuracy: 0.9681
Epoch 14/15
60000/60000 [==============================] - 5s 81us/sample - loss: 0.0862 -
accuracy: 0.9699
```

```
Epoch 15/15
60000/60000 [==============================] - 5s 81us/sample - loss: 0.0837 -
accuracy: 0.9708
```

[128]:
```
<keras.callbacks.History at 0x7f8517268310>
```

[129]:
```
test_loss, test_acc = model_fash2.evaluate(test_imgs_fash, test_labels_fash)
print(f"Accuracy on the test dataset = {test_acc}")
```

```
/usr/local/lib/python3.7/dist-packages/keras/engine/training_v1.py:2057:
UserWarning: `Model.state_updates` will be removed in a future version. This
property should not be used in TensorFlow 2.0, as `updates` are applied
automatically.
  updates = self.state_updates
```

```
Accuracy on the test dataset = 0.9185000061988831
```

6 Visualizing Filter Response

We use gradient descent in input space to display the visual pattern each filter is maximally responsive to.
To this end we take a VGG19 convnet pretrained on the ImageNet dataset.

Very Deep Convolutional Networks for Large-Scale Image Recognition Karen Simonyan, Andrew Zisserman

DL Architecture

[105]:
```
from tensorflow.keras.applications import VGG19
from tensorflow.keras import backend as K
import numpy as np

import matplotlib.pyplot as plt

#Load pretrained model
#we omit the densely connected layers of the network
model = VGG19(weights='imagenet', include_top=False)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5
80142336/80134624 [==============================] - 0s 0us/step
80150528/80134624 [==============================] - 0s 0us/step
```

[106]:
```
model.summary()
```

```
Model: "vgg19"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
```

```
 input_1 (InputLayer)        [(None, None, None, 3)]   0

 block1_conv1 (Conv2D)       (None, None, None, 64)    1792

 block1_conv2 (Conv2D)       (None, None, None, 64)    36928

 block1_pool (MaxPooling2D)  (None, None, None, 64)    0

 block2_conv1 (Conv2D)       (None, None, None, 128)   73856

 block2_conv2 (Conv2D)       (None, None, None, 128)   147584

 block2_pool (MaxPooling2D)  (None, None, None, 128)   0

 block3_conv1 (Conv2D)       (None, None, None, 256)   295168

 block3_conv2 (Conv2D)       (None, None, None, 256)   590080

 block3_conv3 (Conv2D)       (None, None, None, 256)   590080

 block3_conv4 (Conv2D)       (None, None, None, 256)   590080

 block3_pool (MaxPooling2D)  (None, None, None, 256)   0

 block4_conv1 (Conv2D)       (None, None, None, 512)   1180160

 block4_conv2 (Conv2D)       (None, None, None, 512)   2359808

 block4_conv3 (Conv2D)       (None, None, None, 512)   2359808

 block4_conv4 (Conv2D)       (None, None, None, 512)   2359808

 block4_pool (MaxPooling2D)  (None, None, None, 512)   0

 block5_conv1 (Conv2D)       (None, None, None, 512)   2359808

 block5_conv2 (Conv2D)       (None, None, None, 512)   2359808

 block5_conv3 (Conv2D)       (None, None, None, 512)   2359808

 block5_conv4 (Conv2D)       (None, None, None, 512)   2359808

 block5_pool (MaxPooling2D)  (None, None, None, 512)   0

=================================================================
Total params: 20,024,384
Trainable params: 20,024,384
Non-trainable params: 0
```

----

[107]:
```
#Specify filter you want to visualize and get its output
layer_name = 'block5_conv3'
filter_index = 3
layer_output = model.get_layer(layer_name).output

#Loss is the averaged activation of the chosen filter
loss = K.mean(layer_output[:, :, :, filter_index])
```

[108]:
```
#Gradients of loss with respect to the input
#upgrading to 2.x: tf.gradients is no longer supported
#requiring tf.compat.v1.disable_eager_execution()
grads = K.gradients(loss, model.input)[0]

#A trick is to normalize the gradients by their L2 norm
#This ensures that the magnitude of the gradients is always in the same range
#and leads to a smooth descent process
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
```

[109]:
```
#The tensors defined so far (loss, grads) were symbolic
#To obtain values we need to feed an input via K.function

iterate = K.function([model.input], [loss, grads])
loss_value, grads_value = iterate([np.zeros((1, 150, 150, 3))])
```

[110]:
```
# print(grads)
# print(grads_value)
```

[111]:
```
#Implement the actual gradient descent
#Initial input is a grey image with some noise

input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128.
step = 1.
for i in range(40):
    loss_value, grads_value = iterate([input_img_data])
    input_img_data += grads_value * step
```

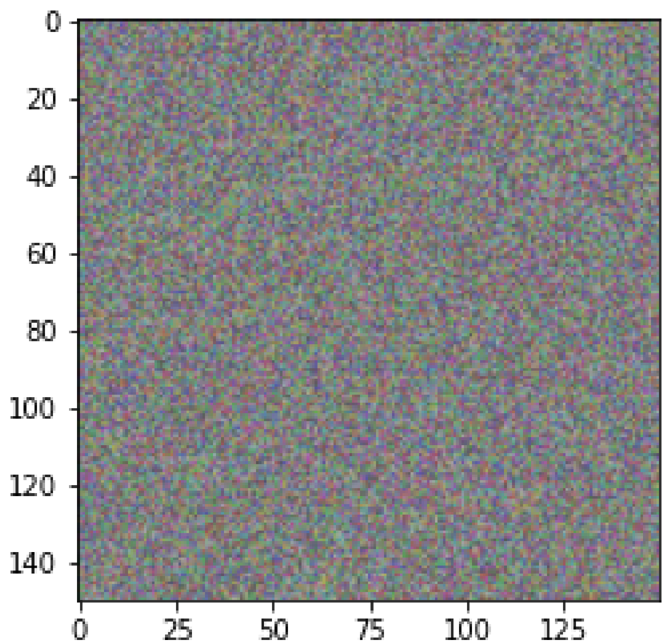
[112]:
```
# print(grads_value)
```

[116]:
```
#Postprocess to turn into displayable image

def deprocess_image(x):
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1
```

```
    x += 0.5
    x = np.clip(x, 0, 1)

    x *= 255
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

[117]:
```
plt.imshow(deprocess_image(input_img_data[0]))
```

[117]: <matplotlib.image.AxesImage at 0x7f85177af250>



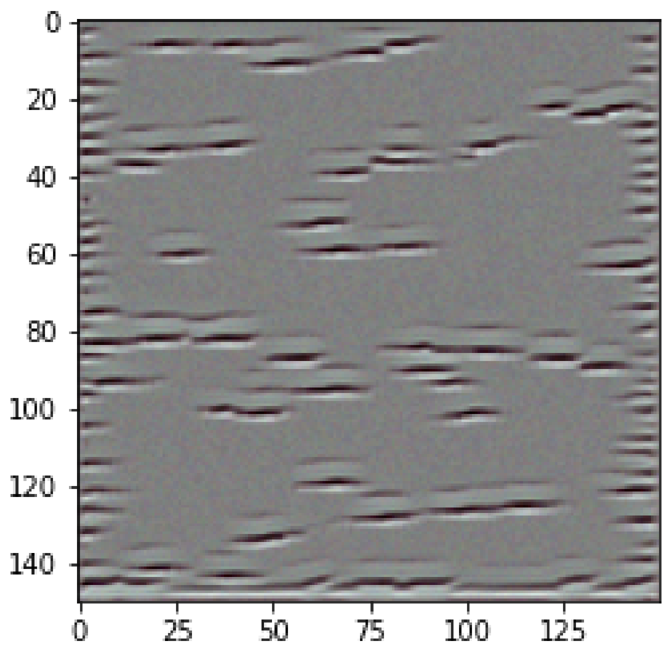
HOMEWORK 2:

Write a function that takes as arguments the name of the layer and filter index and outputs the displayable filter response.
Then you can choose different filters and visualize which patterns they are responsive too! Submit the code (as Notebook) and at least 3 filter responses (a PDF file).

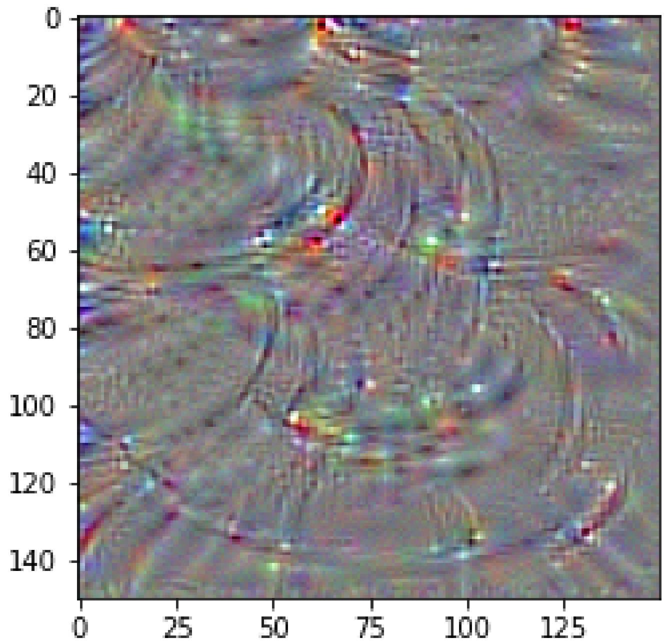
[143]:
```
def visualize_layer(model, layer_name, filter_index):
  layer_output = model.get_layer(layer_name).output
  loss = K.mean(layer_output[:, :, :, filter_index])

  grads = K.gradients(loss, model.input)[0]
  #normalizing gradients by L2 norm
  grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
```

```
iterate = K.function([model.input], [loss, grads])
loss_value, grads_value = iterate([np.zeros((1, 150, 150, 3))])

input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128.
step = 1.
for i in range(40):
    loss_value, grads_value = iterate([input_img_data])
    input_img_data += grads_value * step

plt.imshow(deprocess_image(input_img_data[0]))
filename = layer_name + "_" + str(filter_index) + ".pdf"
plt.imsave(filename, deprocess_image(input_img_data[0]))
```

[144]:
```
visualize_layer(model, 'block1_conv2', 63)
```



[145]:
```
visualize_layer(model, 'block5_conv3', 500)
```



[146]:
```
visualize_layer(model, 'block3_conv4', 200)
```