I have been very fascinated by how Convolution Neural  Networks have been able to, so efficiently,  do image classification and image recognition CNN’s have been very successful in in both these tasks.

In their paper they show how by passing the feature map through a deconvnet ,which does the reverse process of the convnet, they can reconstruct what input pattern originally caused a given activation in the feature map

In the paper they say “A deconvnet can be thought of as a convnet model that uses the same components (filtering, pooling) but in reverse, so instead of mapping pixels to features, it does the opposite. An input image is presented to the CNN and features  activation computed throughout the layers. To examine a given convnet activation, we set all other activations in the layer to zero and pass the feature maps as input to the attached deconvnet layer. Then we successively (i) unpool, (ii) rectify and (iii) filter to reconstruct the activity in the layer beneath that gave rise to the chosen activation. This is then repeated until input pixel space is reached.”

I started to scout the internet to see how I can implement this reverse process of Convolution to understand what really happens under the hood of a CNN.

This post takes VGG16 as the pre-trained network and then uses this network to display the intermediate visualizations.  While this post was very informative and also the visualizations of the various images were very clear, I wanted to simplify the problem for my own understanding.

Hence I decided to take the MNIST digit classification as my base problem. I created a simple 3 layer CNN which gives close to 99.1% accuracy and decided to see if I could do the visualization.

As mentioned in the above post, there are 3 major visualisations

1. Feature activations at the layer
2. Visualisation of the filters
3. Visualisation of the class outputs

**Feature Activation** – This visualization the feature activation at the 3 different layers for a given input image. It can be seen that first layer  activates based on the edge of the image. Deeper layers activate in a more abstract way.

**Visualization of the filters**: This visualization shows what patterns the filters respond maximally to. To do this the following is repeated in a loop:

* Choose a loss function that maximizes the value of a convnet filter activation
* Do gradient ascent (maximization) in input space that increases the filter activation

**Visualizing Class Outputs of the MNIST Convnet:** This process is similar to determining the filter activation. Here the convnet is made to generate an image that represents the category maximally.

import numpy as np

import pandas as pd

import os

import tensorflow as tf

import matplotlib.pyplot as plt

from keras.layers import Dense, Dropout, Flatten

from keras.layers import Conv2D, MaxPooling2D, Input

from keras.models import Model

from sklearn.model\_selection import train\_test\_split

from keras.utils import np\_utils

Using TensorFlow backend.

In [0]:

mnist=tf.keras.datasets.mnist

# Set training and test data and labels

(training\_images,training\_labels),(test\_images,test\_labels)=mnist.load\_data()

In [0]:

#Normalize training data

X =np.array(training\_images).reshape(training\_images.shape[0],28,28,1)

# Normalize the images by dividing by 255.0

X = X/255.0

X.shape

# Use one hot encoding for the labels

Y = np\_utils.to\_categorical(training\_labels, 10)

Y.shape

# Split training data into training and validation data in the ratio of 80:20

X\_train, X\_validation, y\_train, y\_validation = train\_test\_split(X,Y,test\_size=0.20, random\_state=42)

In [4]:

# Normalize test data

X\_test =np.array(test\_images).reshape(test\_images.shape[0],28,28,1)

X\_test=X\_test/255.0

#Use OHE for the test labels

Y\_test = np\_utils.to\_categorical(test\_labels, 10)

X\_test.shape

Out[4]:

(10000, 28, 28, 1)

**Display data**

Display the training data and the corresponding labels

In [5]:

print(training\_labels[0:10])

f, axes = plt.subplots(1, 10, sharey=True,figsize=(10,10))

for i,ax in enumerate(axes.flat):

ax.axis('off')

ax.imshow(X[i,:,:,0],cmap="gray")



**Create a Convolutional Neural Network**

The CNN consists of 3 layers

* Conv2D of size 28 x 28 with 24 filters
* Perform Max pooling
* Conv2D of size 14 x 14 with 48 filters
* Perform max pooling
* Conv2d of size 7 x 7 with 64 filters
* Flatten
* Use Dense layer with 128 units
* Perform 25% dropout
* Perform categorical cross entropy with softmax activation function

In [0]:

num\_classes=10

inputs = Input(shape=(28,28,1))

x = Conv2D(24,kernel\_size=(3,3),padding='same',activation="relu")(inputs)

x = MaxPooling2D(pool\_size=(2, 2))(x)

x = Conv2D(48, (3, 3), padding='same',activation='relu')(x)

x = MaxPooling2D(pool\_size=(2, 2))(x)

x = Conv2D(64, (3, 3), padding='same',activation='relu')(x)

x = MaxPooling2D(pool\_size=(2, 2))(x)

x = Flatten()(x)

x = Dense(128, activation='relu')(x)

x = Dropout(0.25)(x)

output = Dense(num\_classes,activation="softmax")(x)

model = Model(inputs,output)

model.compile(loss='categorical\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

**Summary of CNN**

Display the summary of CNN

In [7]:

model.summary()

Model: "model\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) (None, 28, 28, 1) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_1 (Conv2D) (None, 28, 28, 24) 240

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2 (None, 14, 14, 24) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 14, 14, 48) 10416

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_2 (MaxPooling2 (None, 7, 7, 48) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_3 (Conv2D) (None, 7, 7, 64) 27712

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_3 (MaxPooling2 (None, 3, 3, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_1 (Flatten) (None, 576) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 128) 73856

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 10) 1290

=================================================================

Total params: 113,514

Trainable params: 113,514

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Perform Gradient descent and validate with the validation data**

In [8]:

epochs = 20

batch\_size=256

history = model.fit(X\_train,y\_train,

epochs=epochs,

batch\_size=batch\_size,

validation\_data=(X\_validation,y\_validation))

————————————————

acc = history.history[ ‘accuracy’ ]

val\_acc = history.history[ ‘val\_accuracy’ ]

loss = history.history[ ‘loss’ ]

val\_loss = history.history[‘val\_loss’ ]

epochs = range(len(acc)) # Get number of epochs

#————————————————

# Plot training and validation accuracy per epoch

#————————————————

plt.plot ( epochs, acc,label=”training accuracy” )

plt.plot ( epochs, val\_acc, label=’validation acuracy’ )

plt.title (‘Training and validation accuracy’)

plt.legend()

plt.figure()

#————————————————

# Plot training and validation loss per epoch

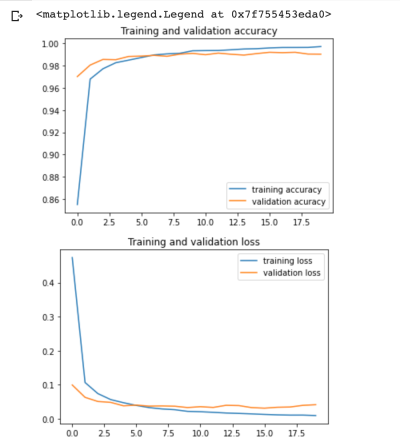
#————————————————

plt.plot ( epochs, loss , label=”training loss”)

plt.plot ( epochs, val\_loss,label=”validation loss” )

plt.title (‘Training and validation loss’ )

plt.legend()



**Test model on test data**

f, axes = plt.subplots(1, 10, sharey=True,figsize=(10,10))

for i,ax in enumerate(axes.flat):

ax.axis(‘off’)

ax.imshow(X\_test[i,:,:,0],cmap=”gray”)



l=[]

for i in range(10):

x=X\_test[i].reshape(1,28,28,1)

y=model.predict(x)

m = np.argmax(y, axis=1)

print(m)

[7]

[2]

[1]

[0]

[4]

[1]

[4]

[9]

[5]

[9]

**Generate the filter activations at the intermediate CNN layers**

In [12]:

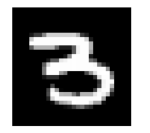
img = test\_images[51].reshape(1,28,28,1)

fig = plt.figure(figsize=(5,5))

print(img.shape)

plt.imshow(img[0,:,:,0],cmap="gray")

plt.axis('off')



**Display the activations at the intermediate layers**

This displays the intermediate activations as the image passes through the filters and generates these feature maps

In [13]:

layer\_names = ['conv2d\_4', 'conv2d\_5', 'conv2d\_6']

layer\_outputs = [layer.output for layer in model.layers if 'conv2d' in layer.name]

activation\_model = Model(inputs=model.input,outputs=layer\_outputs)

intermediate\_activations = activation\_model.predict(img)

images\_per\_row = 8

max\_images = 8

for layer\_name, layer\_activation in zip(layer\_names, intermediate\_activations):

print(layer\_name,layer\_activation.shape)

n\_features = layer\_activation.shape[-1]

print("features=",n\_features)

n\_features = min(n\_features, max\_images)

print(n\_features)

size = layer\_activation.shape[1]

print("size=",size)

n\_cols = n\_features // images\_per\_row

display\_grid = np.zeros((size \* n\_cols, images\_per\_row \* size))

for col in range(n\_cols):

for row in range(images\_per\_row):

channel\_image = layer\_activation[0,:, :, col \* images\_per\_row + row]

channel\_image -= channel\_image.mean()

channel\_image /= channel\_image.std()

channel\_image \*= 64

channel\_image += 128

channel\_image = np.clip(channel\_image, 0, 255).astype('uint8')

display\_grid[col \* size : (col + 1) \* size,

row \* size : (row + 1) \* size] = channel\_image

scale = 2. / size

plt.figure(figsize=(scale \* display\_grid.shape[1],

scale \* display\_grid.shape[0]))

plt.axis('off')

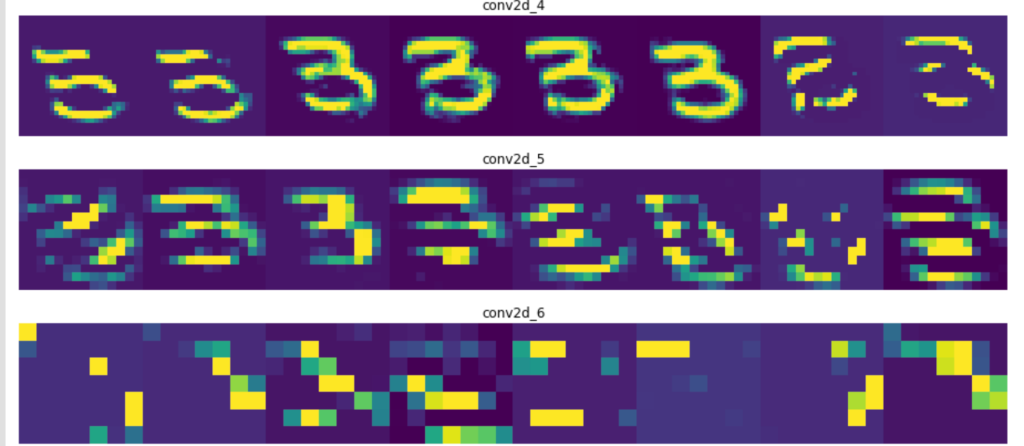
plt.title(layer\_name)

plt.grid(False)

plt.imshow(display\_grid, aspect='auto', cmap='viridis')

plt.show()

It can be seen that at the higher layers only abstract features of the input image are captured



# To fix the ImportError: cannot import name 'imresize' in the next cell. Run this cell. Then comment and restart and run all

#!pip install scipy==1.1.0

**Visualize the pattern that the filters respond to maximally**

* Choose a loss function that maximizes the value of the CNN filter in a given layer
* Start from a blank input image.
* Do gradient ascent in input space. Modify input values so that the filter activates more
* Repeat this in a loop.

In [14]:

from vis.visualization import visualize\_activation, get\_num\_filters

from vis.utils import utils

from vis.input\_modifiers import Jitter

max\_filters = 24

selected\_indices = []

vis\_images = [[], [], [], [], []]

i = 0

selected\_filters = [[0, 3, 11, 15, 16, 17, 18, 22],

[8, 21, 23, 25, 31, 32, 35, 41],

[2, 7, 11, 14, 19, 26, 35, 48]]

# Set the layers

layer\_name = ['conv2d\_4', 'conv2d\_5', 'conv2d\_6']

# Set the layer indices

layer\_idx = [1,3,5]

for layer\_name,layer\_idx in zip(layer\_name,layer\_idx):

# Visualize all filters in this layer.

if selected\_filters:

filters = selected\_filters[i]

else:

# Randomly select filters

filters = sorted(np.random.permutation(get\_num\_filters(model.layers[layer\_idx]))[:max\_filters])

selected\_indices.append(filters)

# Generate input image for each filter.

# Loop through the selected filters in each layer and generate the activation of these filters

for idx in filters:

img = visualize\_activation(model, layer\_idx, filter\_indices=idx, tv\_weight=0.,

input\_modifiers=[Jitter(0.05)], max\_iter=300)

vis\_images[i].append(img)

# Generate stitched image palette with 4 cols so we get 2 rows.

stitched = utils.stitch\_images(vis\_images[i], cols=4)

plt.figure(figsize=(20, 30))

plt.title(layer\_name)

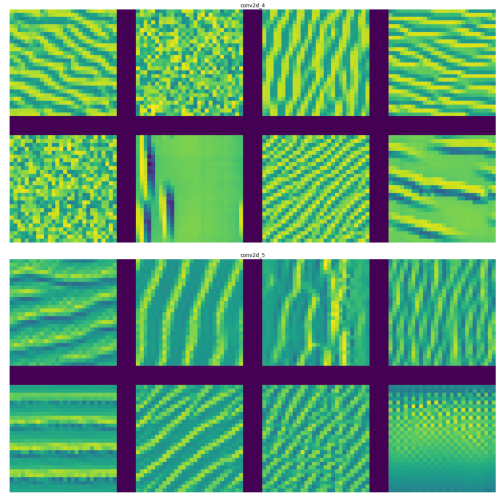
plt.axis('off')

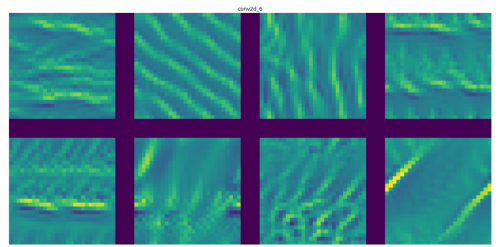
stitched = stitched.reshape(1,61,127,1)

plt.imshow(stitched[0,:,:,0])

plt.show()

i += 1





from vis.utils import utils

new\_vis\_images = [[], [], [], [], []]

i = 0

layer\_name = ['conv2d\_4', 'conv2d\_5', 'conv2d\_6']

layer\_idx = [1,3,5]

for layer\_name,layer\_idx in zip(layer\_name,layer\_idx):

# Generate input image for each filter.

for j, idx in enumerate(selected\_indices[i]):

img = visualize\_activation(model, layer\_idx, filter\_indices=idx,

seed\_input=vis\_images[i][j], input\_modifiers=[Jitter(0.05)], max\_iter=300)

#img = utils.draw\_text(img, 'Filter {}'.format(idx))

new\_vis\_images[i].append(img)

stitched = utils.stitch\_images(new\_vis\_images[i], cols=4)

plt.figure(figsize=(20, 30))

plt.title(layer\_name)

plt.axis('off')

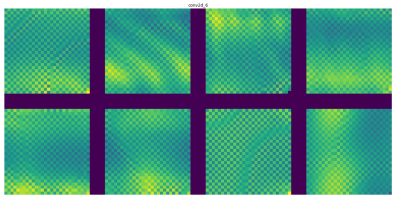
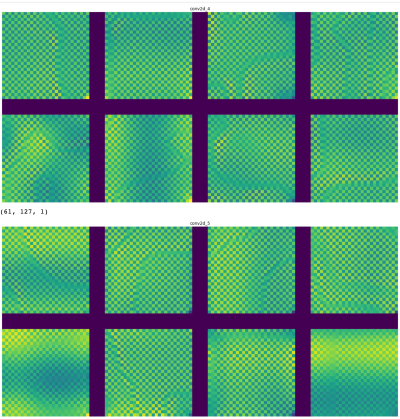
print(stitched.shape)

stitched = stitched.reshape(1,61,127,1)

plt.imshow(stitched[0,:,:,0])

plt.show()

i += 1



**Visualizing Class Outputs**

Here the CNN will generate the image that maximally represents the category. Each of the output represents one of the digits as can be seen below

In [16]:

from vis.utils import utils

from keras import activations

codes = '''

zero 0

one 1

two 2

three 3

four 4

five 5

six 6

seven 7

eight 8

nine 9

'''

layer\_idx=10

initial = []

images = []

tuples = []

# Swap softmax with linear for better visualization

model.layers[layer\_idx].activation = activations.linear

model = utils.apply\_modifications(model)

for line in codes.split('\n'):

if not line:

continue

name, idx = line.rsplit(' ', 1)

idx = int(idx)

img = visualize\_activation(model, layer\_idx, filter\_indices=idx,

tv\_weight=0., max\_iter=300, input\_modifiers=[Jitter(0.05)])

initial.append(img)

tuples.append((name, idx))

i = 0

for name, idx in tuples:

img = visualize\_activation(model, layer\_idx, filter\_indices=idx,

seed\_input = initial[i], max\_iter=300, input\_modifiers=[Jitter(0.05)])

#img = utils.draw\_text(img, name) # Unable to display text on gray scale image

i += 1

images.append(img)

stitched = utils.stitch\_images(images, cols=4)

plt.figure(figsize=(20, 20))

plt.axis('off')

stitched = stitched.reshape(1,94,127,1)

plt.imshow(stitched[0,:,:,0])

plt.show()

In the grid below the class outputs show the MNIST digits to which output responds to maximally. We can see the ghostly outline  
of digits 0 – 9. We can clearly see the outline if 0,1, 2,3,4,5 (yes, it is there!),6,7, 8 and 9. If you look at this from a little distance the digits are clearly visible. Isn’t that really cool!!

