GR 5242 Final Project
Feature Visualization of Convolutional Neural Network

Chenghao Yu	cy2475	Sec 002
Zhengyang Xu	zx2229	Sec 001
Yingqiao Zhang	yz3209	Sec 001
Yanxin Li	yl3774	Sec 001

1. Introduction

Neural networks are usually considered as black boxes, due to its complexity. For a neural network with multiple layers, it is difficult to figure out the features learnt by those hidden layers. In this project, we try to visualize the features learned by the filters of each convolutional layers of a CNN.

1.1 Data Used

In this project, MNIST dataset, SVHN dataset and CIFAR10 dataset were used.

The MNIST dataset of handwriting digit images contains a training set with 60000 images, and a test set with 10000 images. Each input is a 28*28 black and white image. Each output is a label with 10 classes. Digit 1 has label 1, and digit 9 has label 9.

For the SVHN, we use the cropped digits. In the dataset, each input is a 32*32 RGB image. The outputs are labels. There are 10 classes, and one for each digit. 0 has label 10. 1 has label 1, and 9 has label 9. The training set contains 73257 digits, and the test set has 26032 digits.

The CIFAR-10 dataset has 60000 32x32 RGB images in 10 classes. Each class contains 6000 images. The training set contains 50000 images, and the test set has 10000 images.

1.2 Framework of Project

In this project, we used keras. We started from training a simple 2-layers CNN on MNIST. Since the project mainly focused on feature visualization, when building the convnet, we used the code provided by Eijaz Allibhai, in the post *Building a Convolutional Neural Network (CNN) in Keras*. After that, we first visualized the activation maps given by each filter in the two convolutional layers. Then, through gradient ascent, we found an image for each filter that maximally activated the filter. To make the resulted images more interpretable, we tried to apply some kinds regularization.

Later, we turned to SVHN dataset. Again, we trained a simple CNN on SVHN dataset, and tested whether the steps mentioned above can be applied.

For our curiosity, we also tried to apply the methods to the CNN trained on CIFAR10 dataset. We wished to know what would learnt by the each layer's filters, in a CNN trained on a dataset containing more complicated images.

1.3 Preparation

Since feature visualization was not covered in class, we did some researches on it.

We watched the video of a Stamford open course, named *Visualizing and Understanding*. Through the video, we got some a rough idea of feature visualization. Then we read the two Google posts and several related papers, like *Visualizing and Understanding Convolutional Networks* and *Understanding Neural Networks Through Deep Visualization*. After finishing the reading, we discussed a few times and figured out the framework of our project.

2. The feature visualization of the CNN trained on MNIST dataset

2.1 Model

To build the CNN, we used the code provided in Eijaz Allibhai's post.

Using Adam as the optimization method, we trained a 2-layer CNN (conv1-pooling1-conv2-pooling2-fc1) on MNIST dataset. In conv1, there are 64 filters. Each of them is 5*5 with stride 1 and pad 0. In conv2 there are 32 filters, and each of them is 5*5 with stride 1 and pad 0. For the pool1 and pool2, pooling size was 2*2.

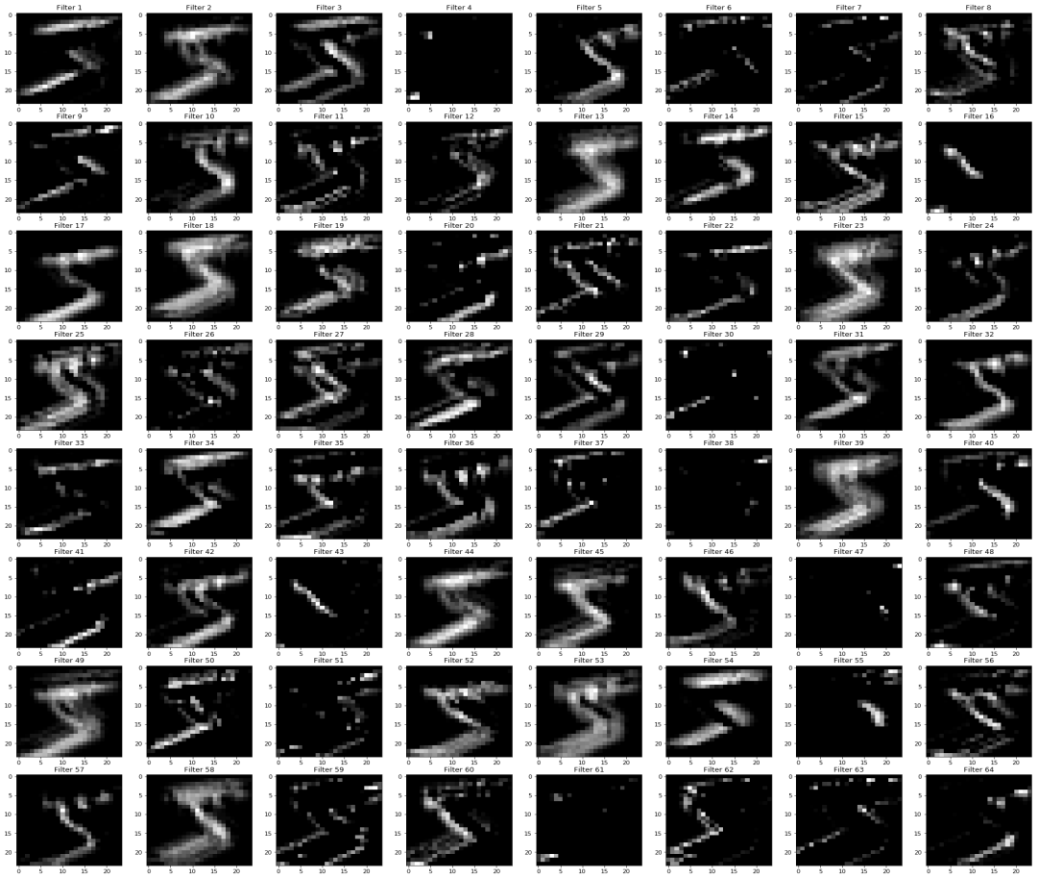
In training, we set the number epochs to 3 and achieved 98% accuracy.

2.2 Visualization of activation maps

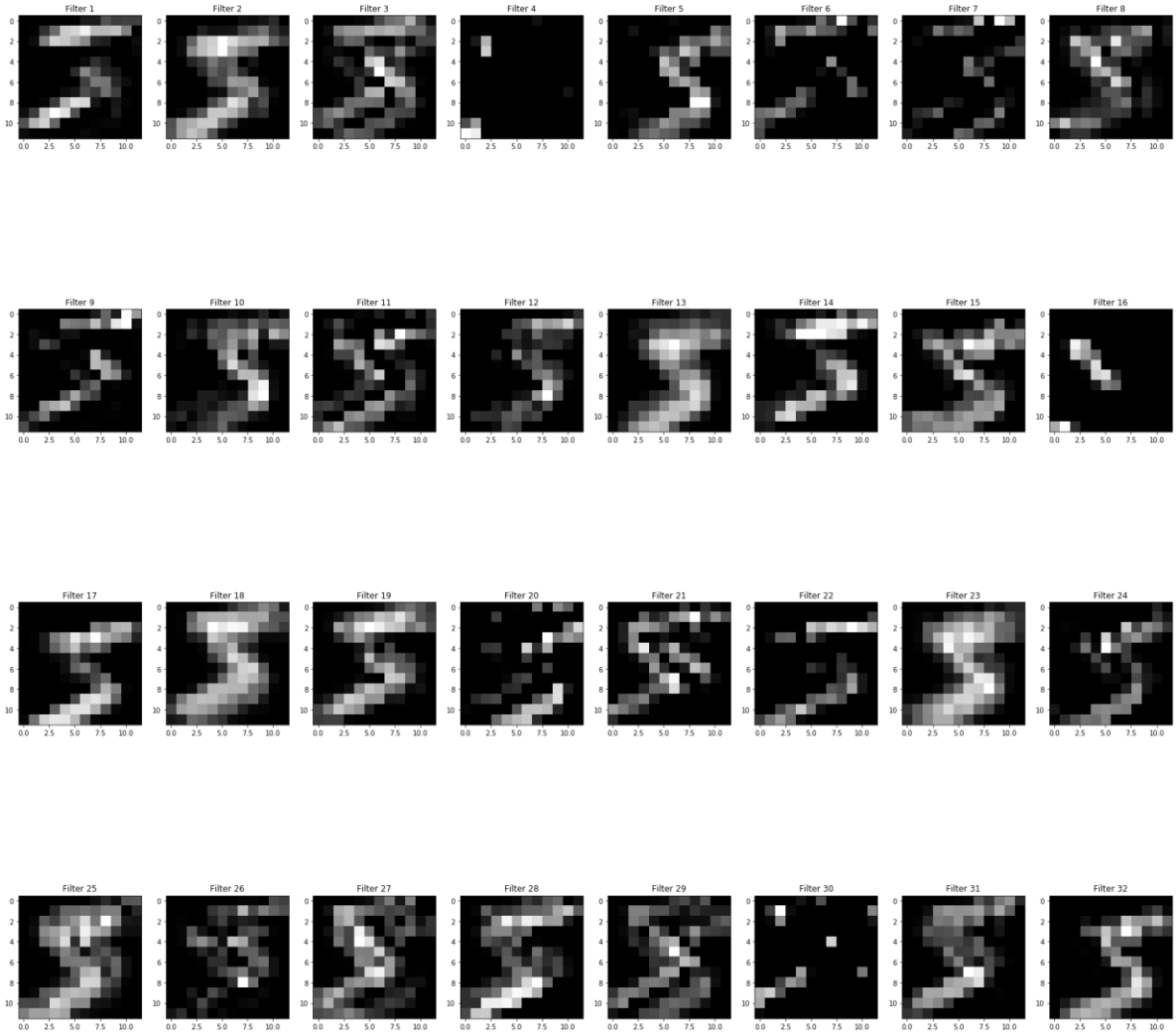
When we got the trained model, first we tried to visualize the activation maps given by each filter in conv1 and conv2. By comparing those activation maps, we wished to get some preliminary understandings of the features learnt by the filters.

To do that, we passed an image of digit 5 to the trained model.

These are the activation maps given by the 64 filters in conv1.



These are the activation maps given by the 32 filters in conv2.



We found that even if in a same layer, different filters gave different activation maps. What is more, the activation maps given by conv2 looked more indistinct then those of conv1.

However, if we only have these activation maps, we could not know more about the features learnt by each filter. It was too difficult to capture the learnt features by human eyes through comparing the activation maps.

2.3 Images that maximize the activation maps

To further understand the filters, we used gradient-based methods to find the input images that maximally activated the filters.

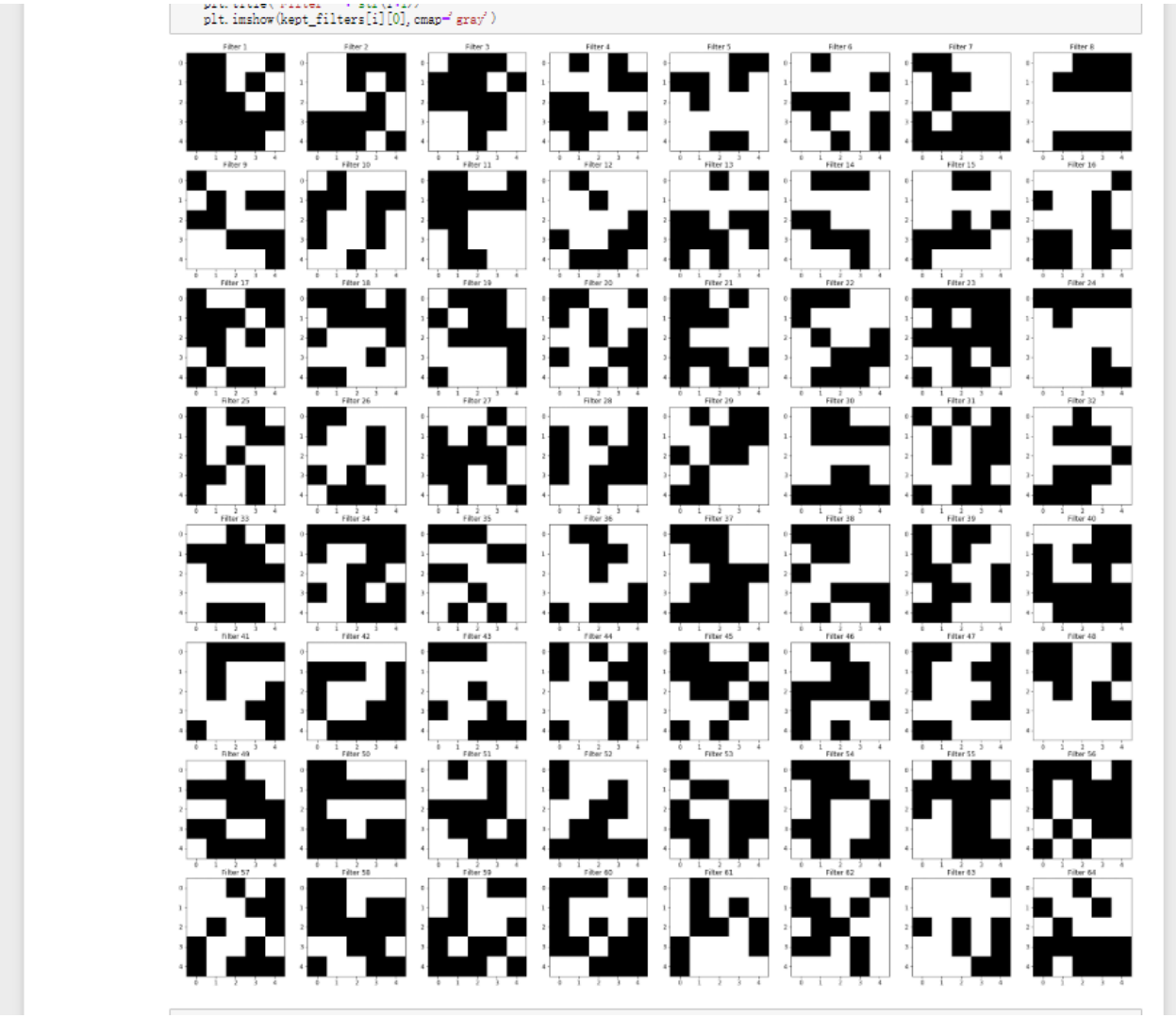
According to the post, *Feature Visualization, how neural networks build up their understanding of images*, we learnt that the objectives of optimization could be a neuron, a channel, a layer, or

other elements we were interested in. We would like to understand individual features, so we could use a neuron or a channel as our objective.

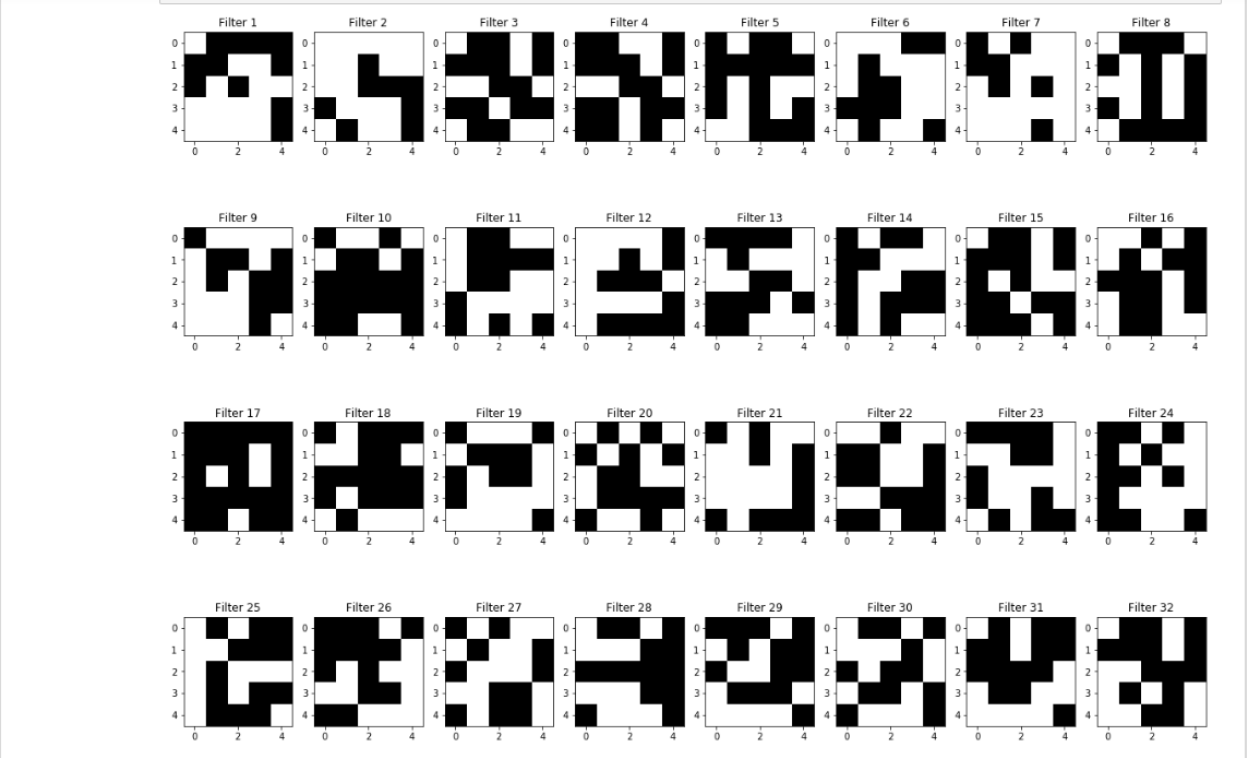
First, we looked for the images that maximized a neuron.

Since all weights in filters were fixed, we could easily get the input which maximize the neuron before relu. The range of the elements in input matrix is [0,1] or [0,255]. Therefore, if the weights in a filter are positive, the corresponding elements in input matrix must be positive and vise versa. To maximize the activation, we let elements equal 1 or 255 if the corresponding elements in filter matrix is positive and let them equal 0 if the corresponding elements is negative.

The resulted images of conv1:



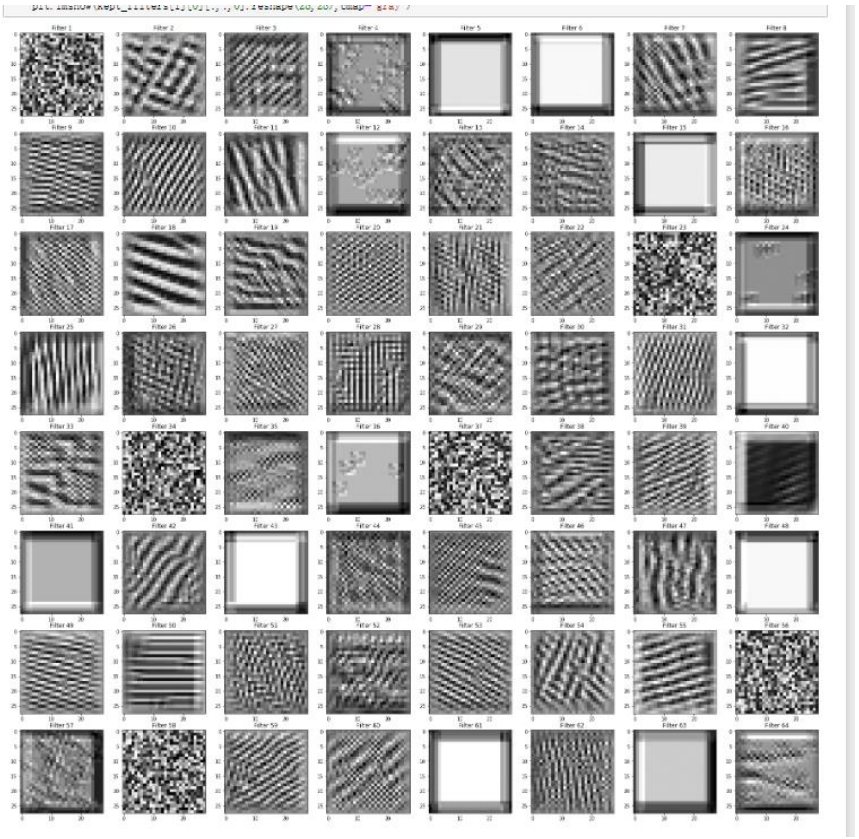
The resulted images of conv2:



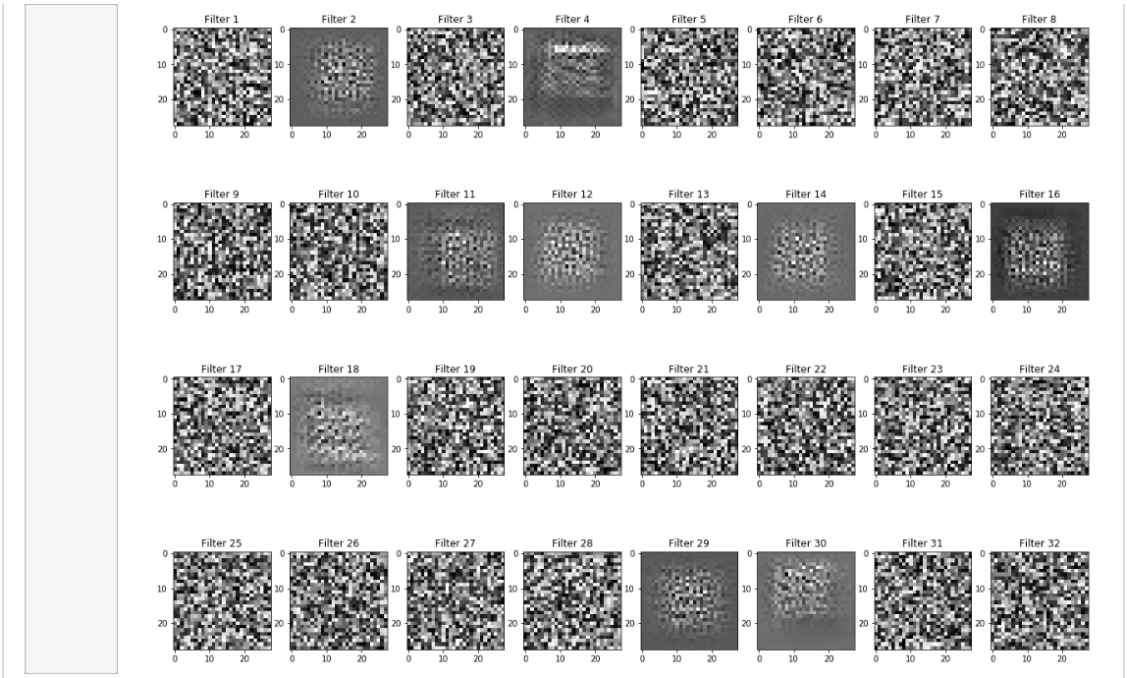
Then, we looked for an image that maximized an entire channel.

We considered an element of an activation map as a linear function, say $a_{ii}(x)$, of an input image. The high value of $a_{ii}(x)$ means that the region scanned could greatly activate the filter. In that case, if we found the gradient of activation map with respect to input image, we would find the input image that maximized the activation map through gradient ascent. However, it was so difficult to realize the idea, since we did not know how to do gradient ascent with a matrix. The post *How convolutional neural networks see the world* inspired us. We turned to find the image that maximize the sum of the elements in an activation map.

The resulted images of filters in conv1:



The resulted images of filters in conv2:



2.4 Summary and problems

The images that maximally activated filters in conv1 showed some edges. We thought that the filters in conv1 learnt edges from input images.

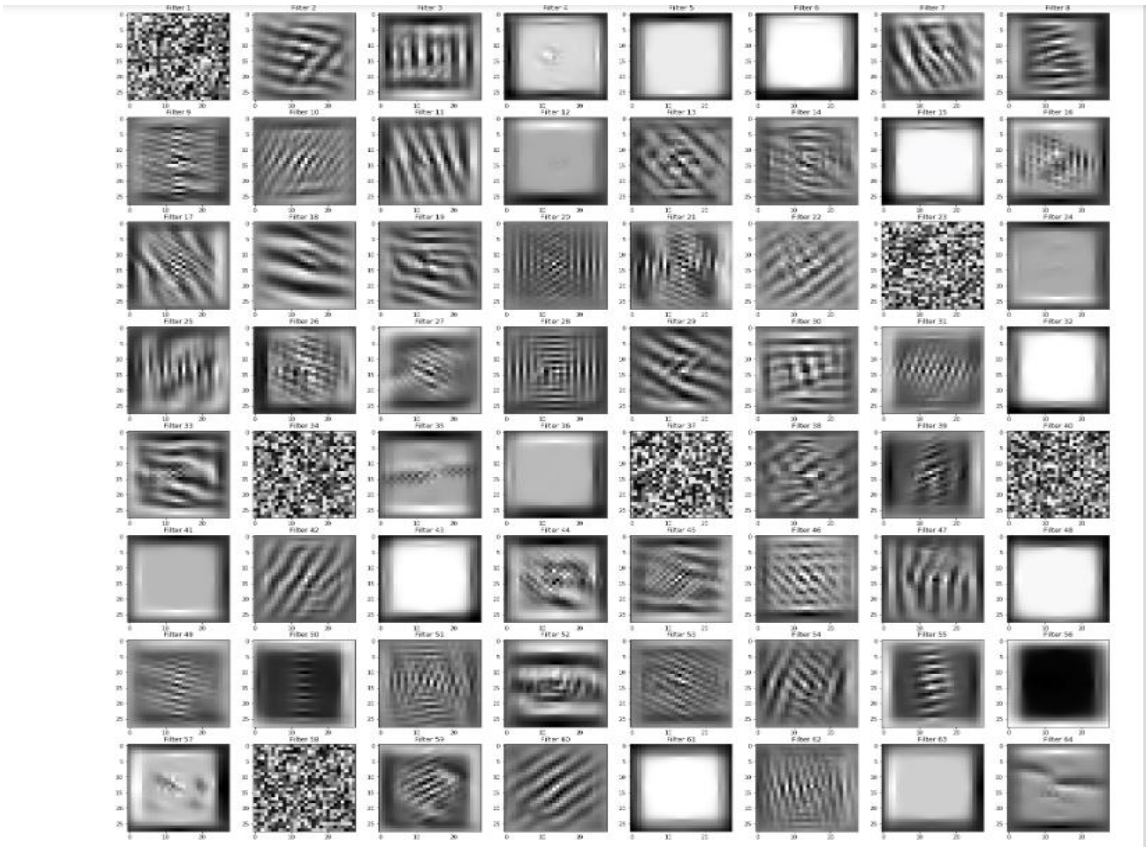
However, we could not tell what were learnt by the filters in conv2. The resulted images were not very interpretable, and many of them were random noise. We thought that there were two reasons.

- The sums of the elements of activation maps given by some filters were zero.
- The starting point and the learning rate of the gradient ascent were randomly selected. The ascent could stop at local max.

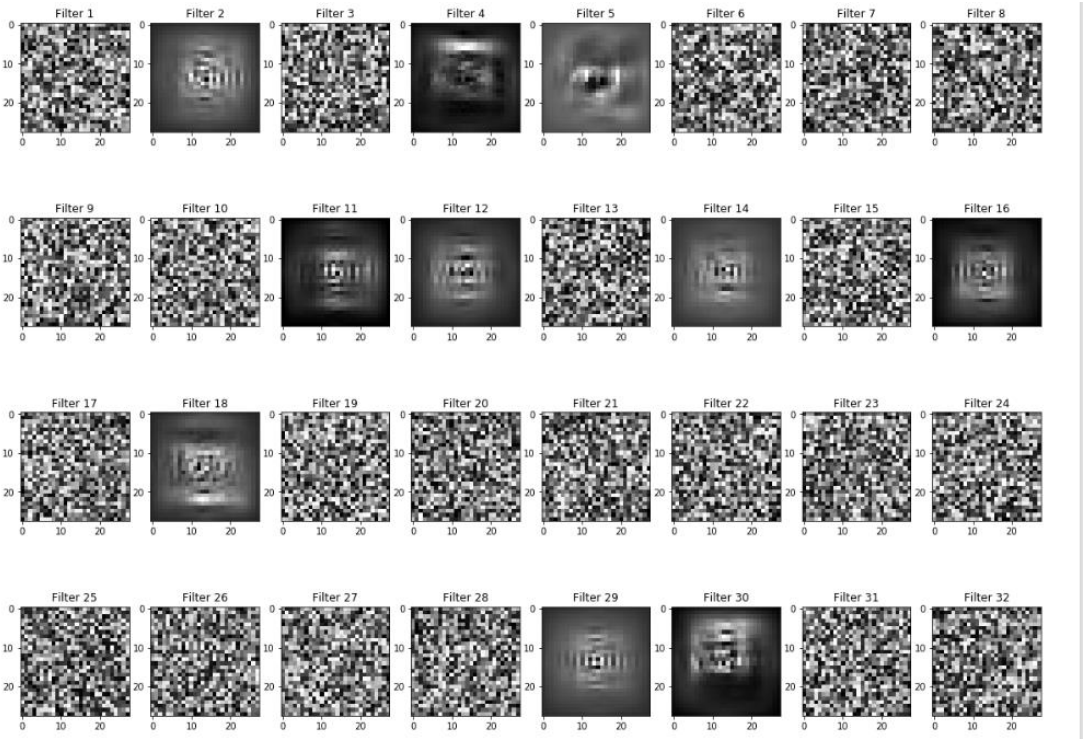
2.5 Improvement

In our project, to make the resulted images more interpretable, we used transformation robustness as the method of regularization. The transformation is applied to the image in each step before the optimization. We use three kinds of transformation: jitter, rotation and scaler. At the beginning, we tried the same parameter used by the google author, which jitter = +-1pix per step, rotate +-5 degree per step, scaler +-1.1 per step. But the result was not good. After we tried several sets, we finally found jitter = 1, rotate +-0.05, and scaler +-0.01 gave obviously noise reduce result.

The resulted images with regularization in conv1:



The resulted images with regularization in conv2:



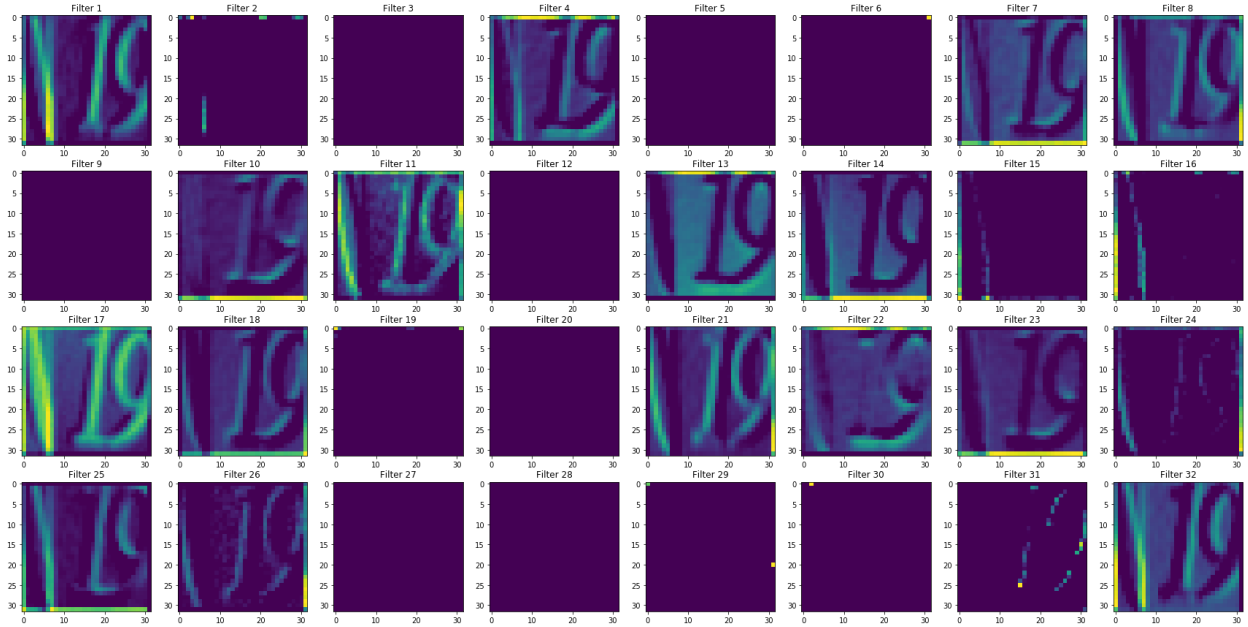
3. The feature visualization of the CNN trained on SVHN dataset

3.1 Model

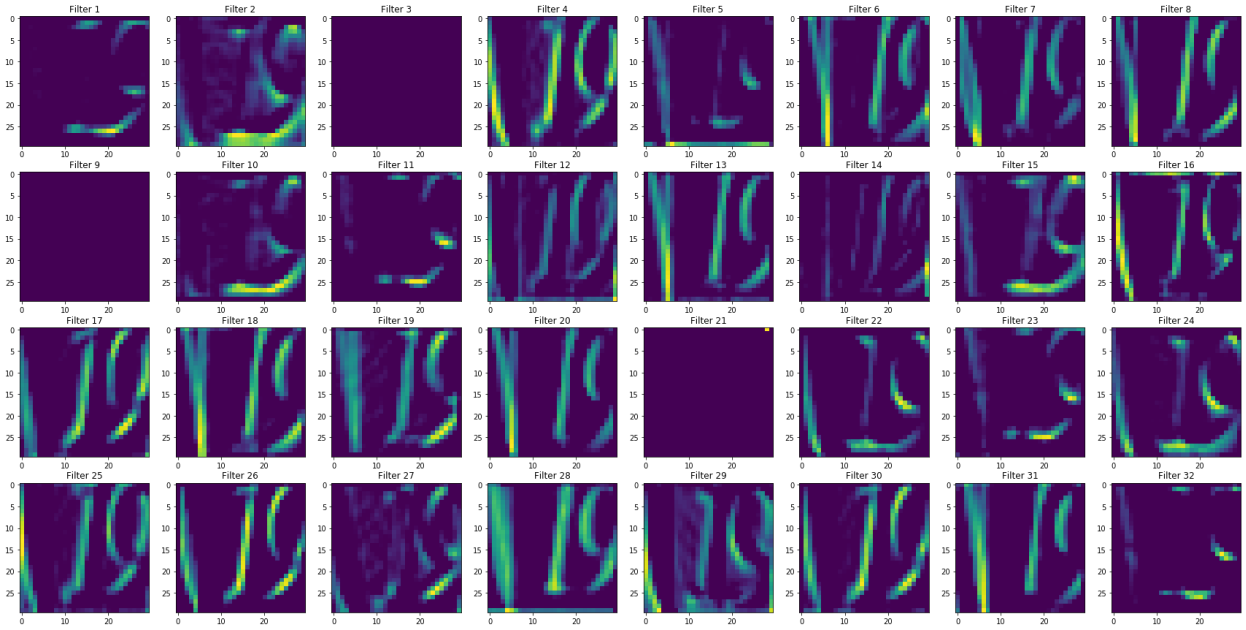
We trained a 4-layer CNN(conv1-conv2-p1-d1-conv3-conv4-p2-d2-fc1-d3-fc2) on SVHN dataset. The structure and the codes of the CNN were from the GitHub post given by Arman Uygur.

3.2 Visualization of activation maps

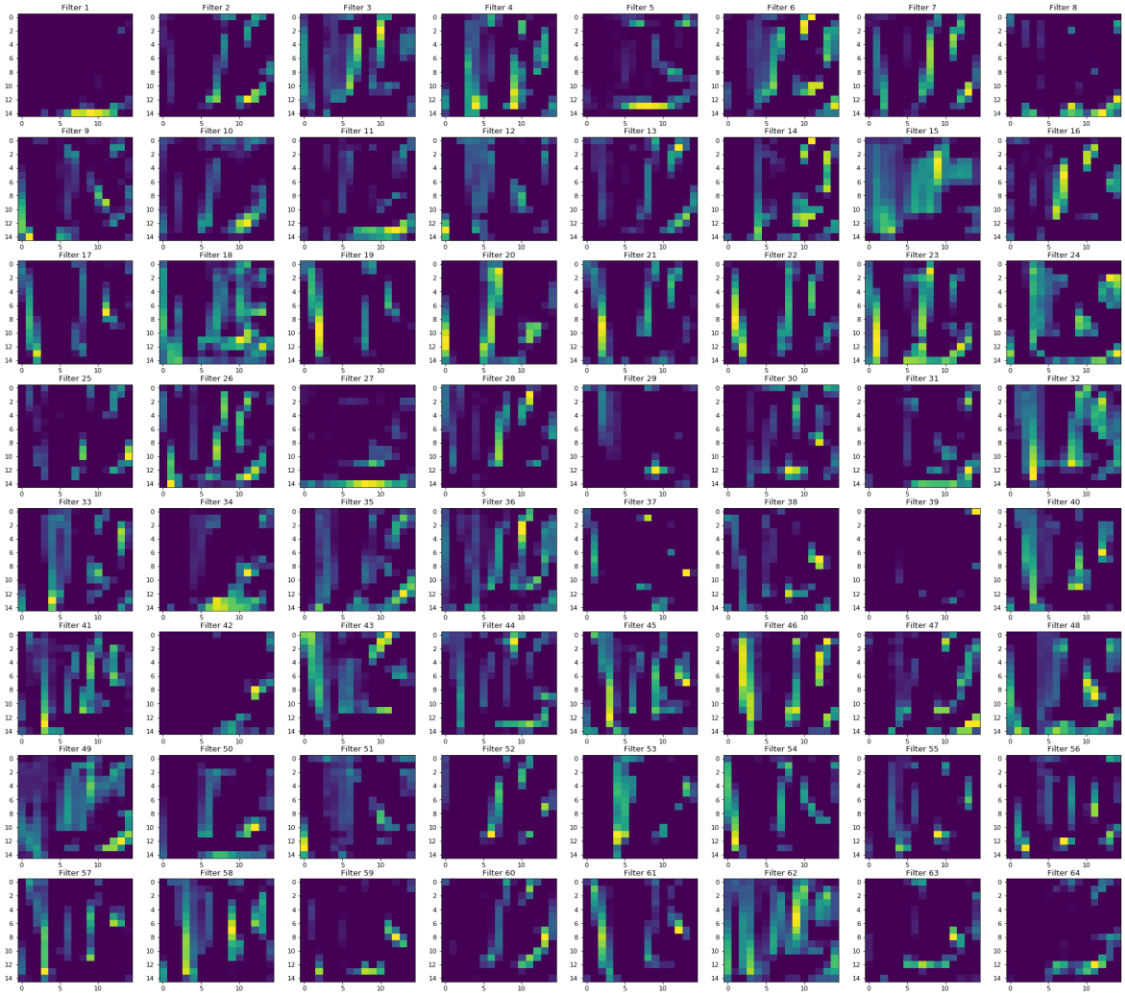
The activation maps given by the filters in conv1:



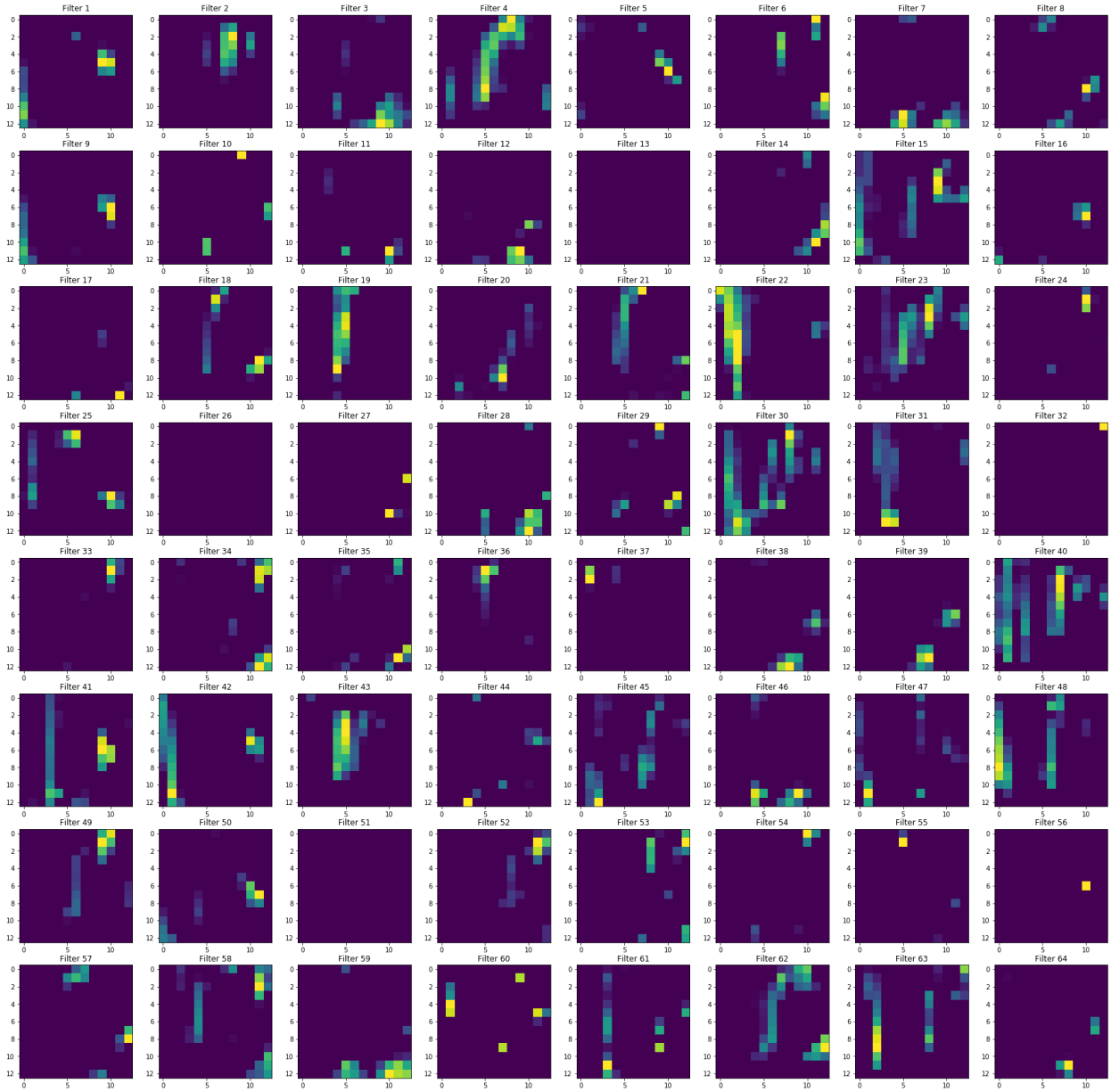
The activation maps given by the filters in conv2:



The activation maps given by the filters in conv3:



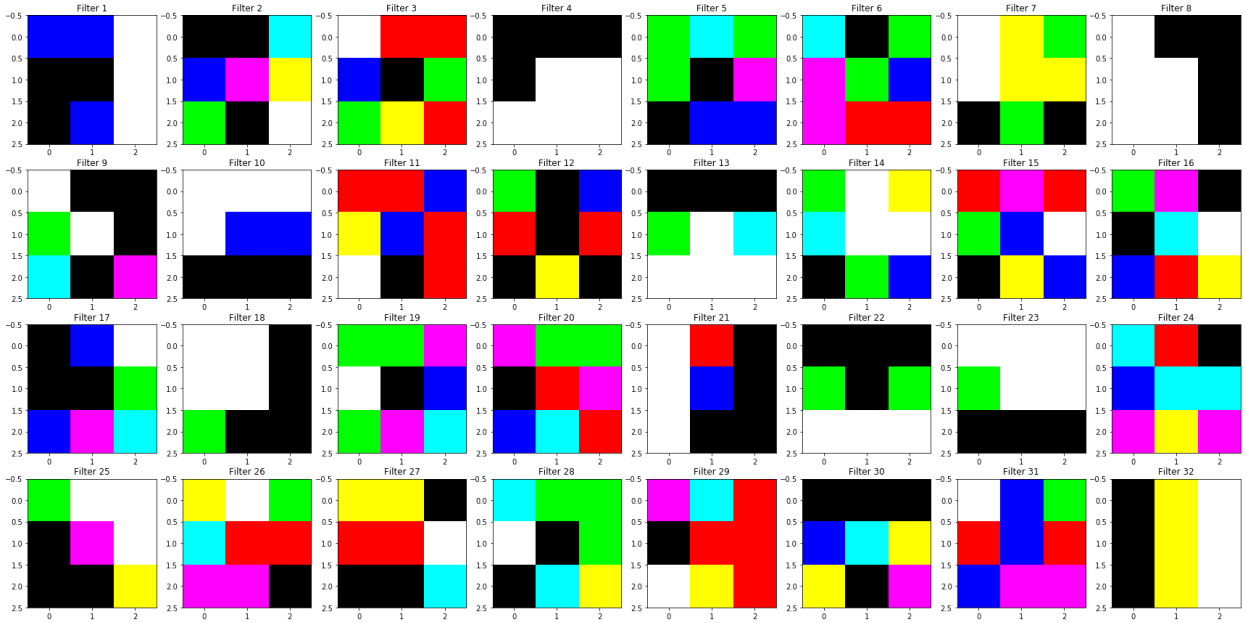
The activation maps given by the filters in conv4:



3.3 Images that maximize the activation maps

When Our objective was a neuron:

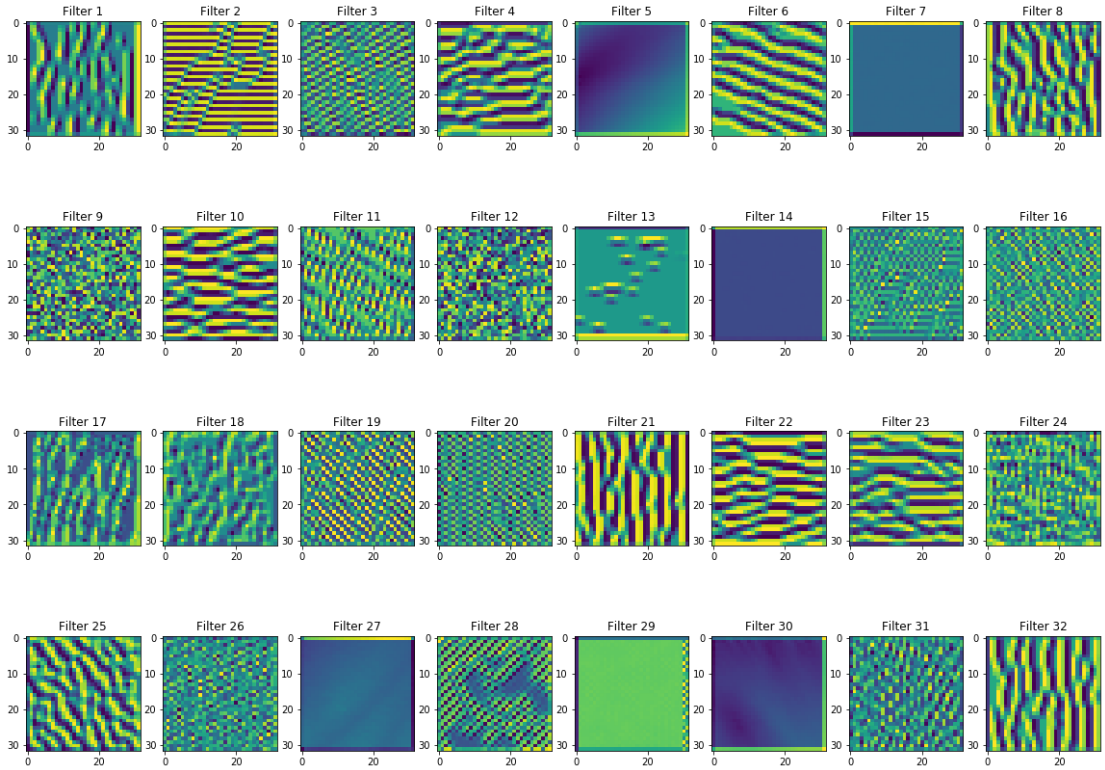
Conv1:



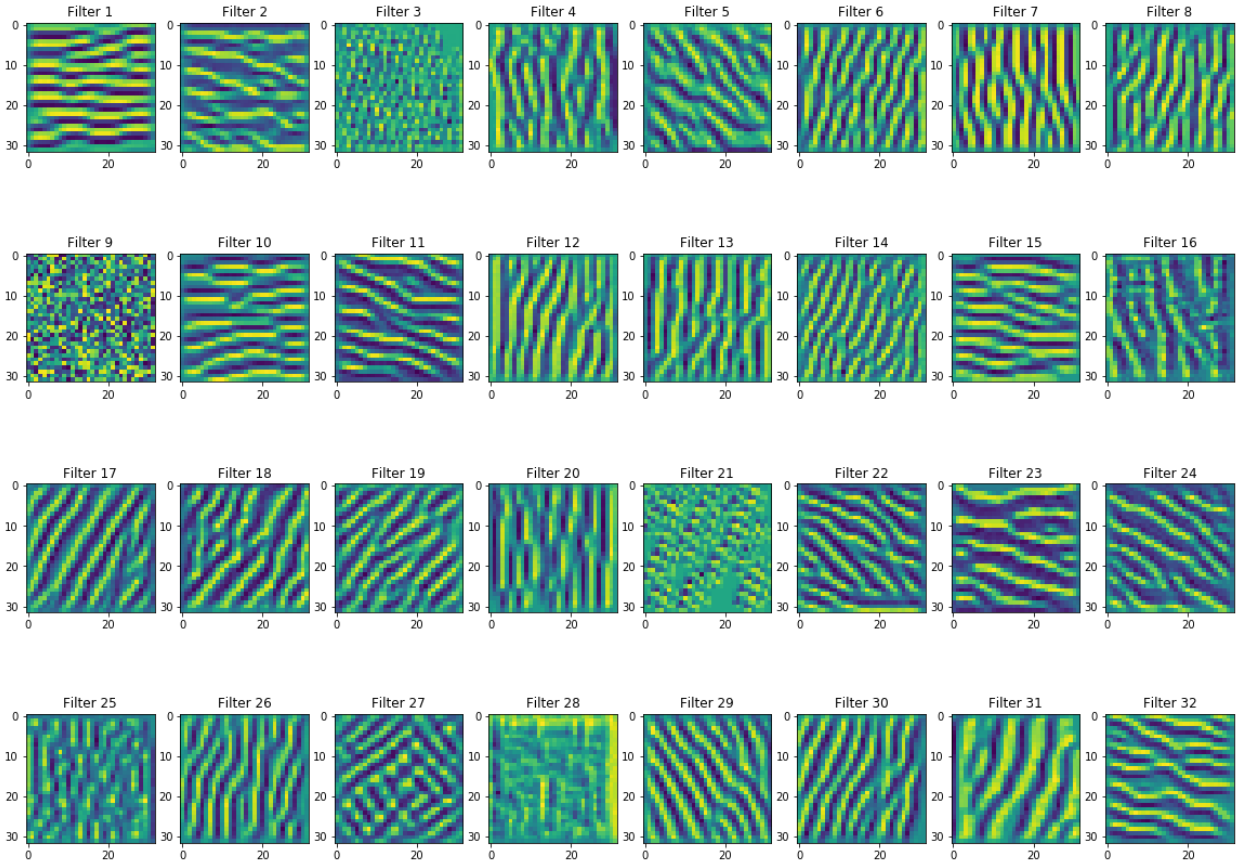
To do this, we needed the weights of the trained model. Since we could not totally understand the structures of the arrays that contained the weights in conv2, conv3, and conv4, we only did this for conv1.

When our objective was an entire layer:

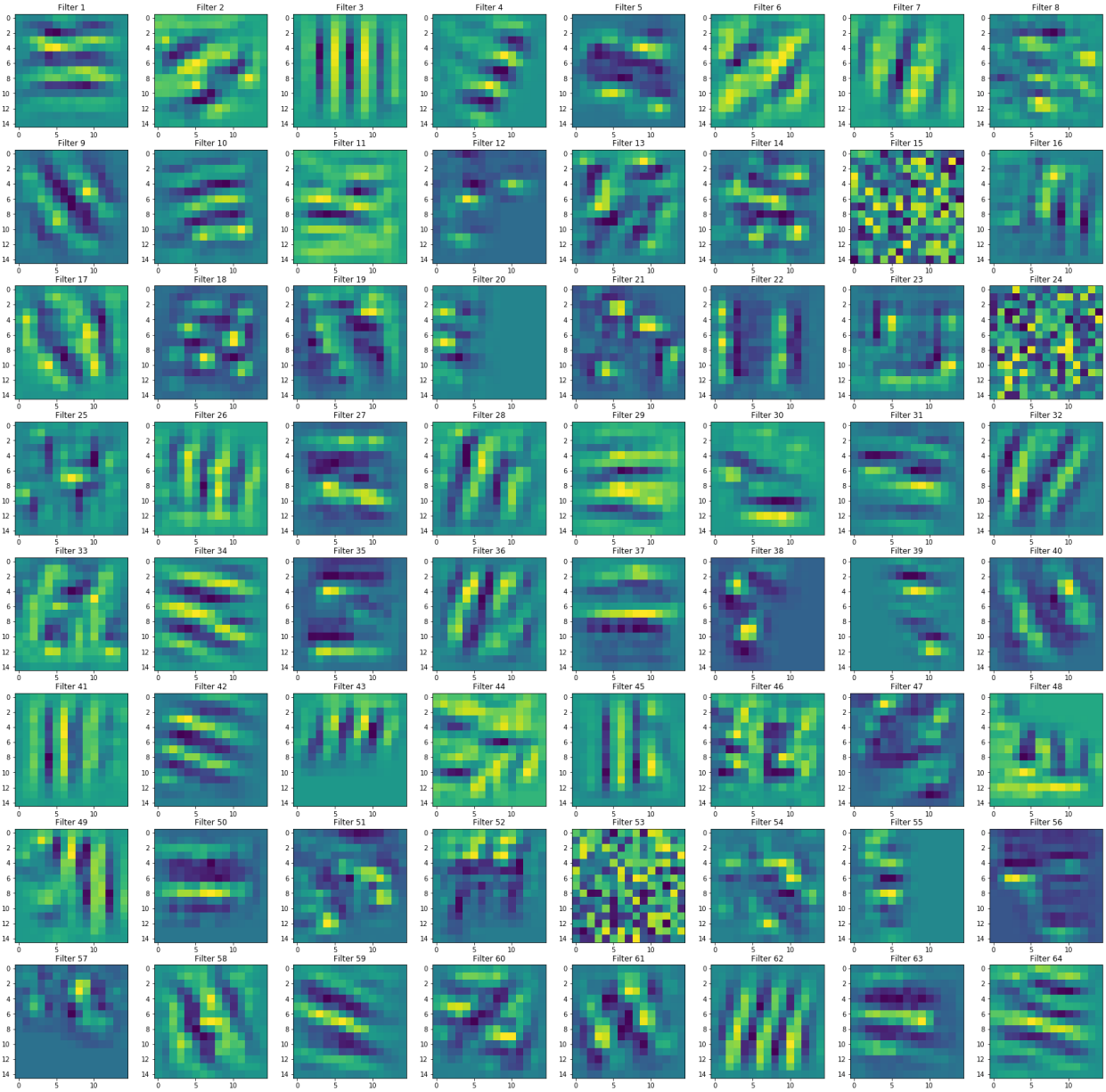
The resulted images for filters in conv1:



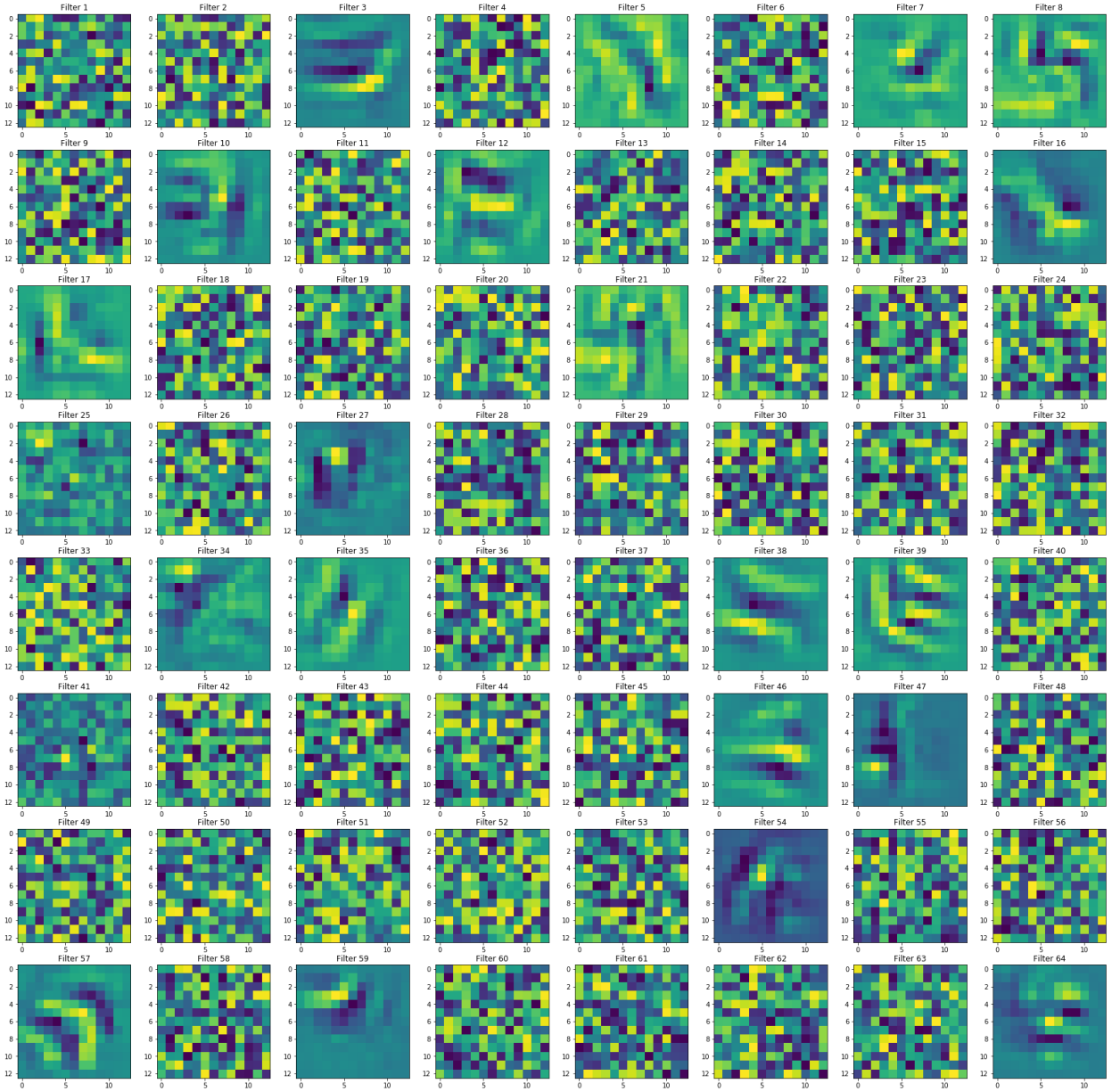
The resulted images for filters in conv2:



The resulted images for filters in conv3:



The resulted images for filters in conv4:



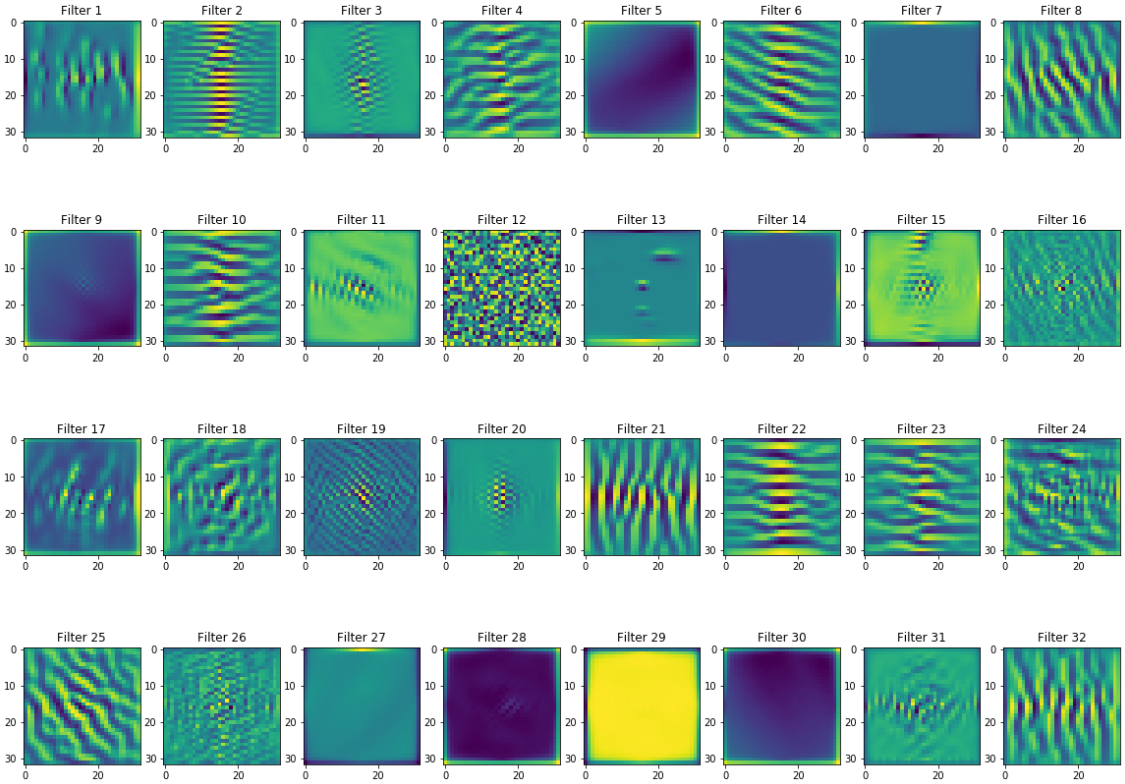
Among the images that maximally activated the filters in conv4, some contained rough outlines of some numbers. For example, the image maximally activated filter 8 looked like 5, and the image maximally activated filter 10 looked like 3.

Trough the four groups of resulted images, we found that the filters in conv1 and conv2 tended to capture small features, such as edges and textures. In deeper layers, the filters would like to learn larger features.

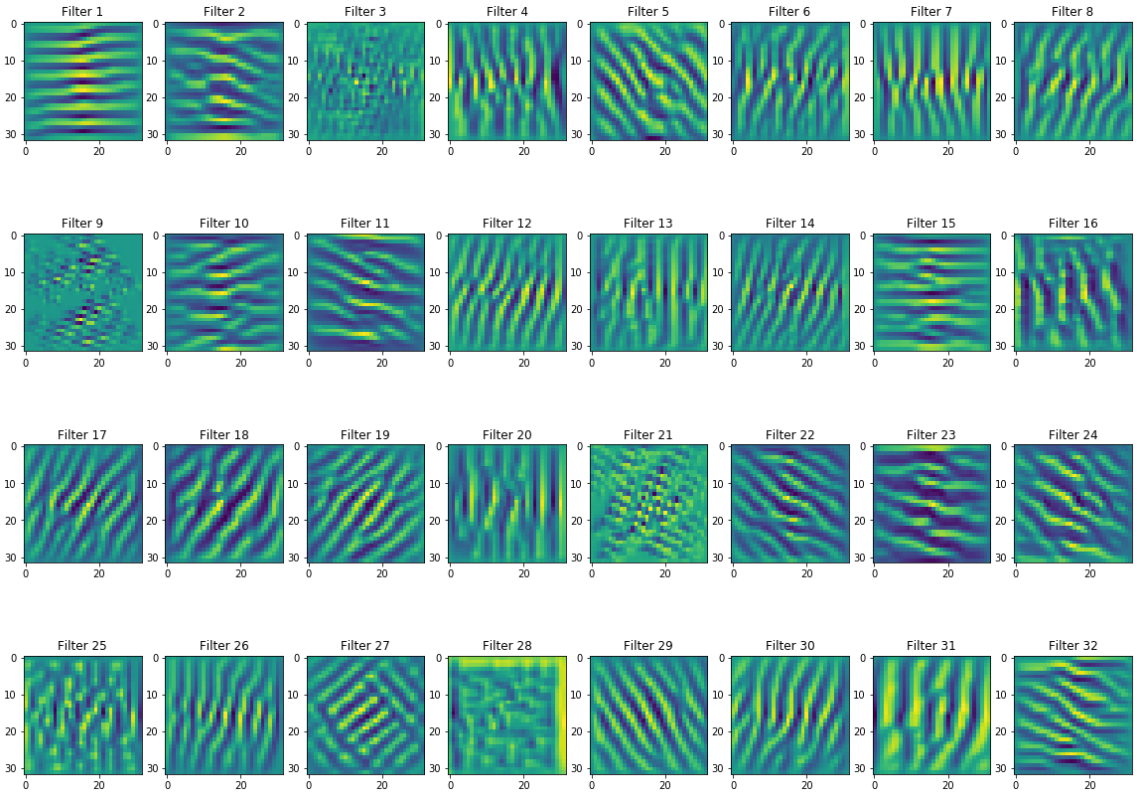
3.5 Improvement

We used three kinds of transformation: jitter, rotation and scaler. jitter = 1, rotate +-0.025, and scaler +-0.01

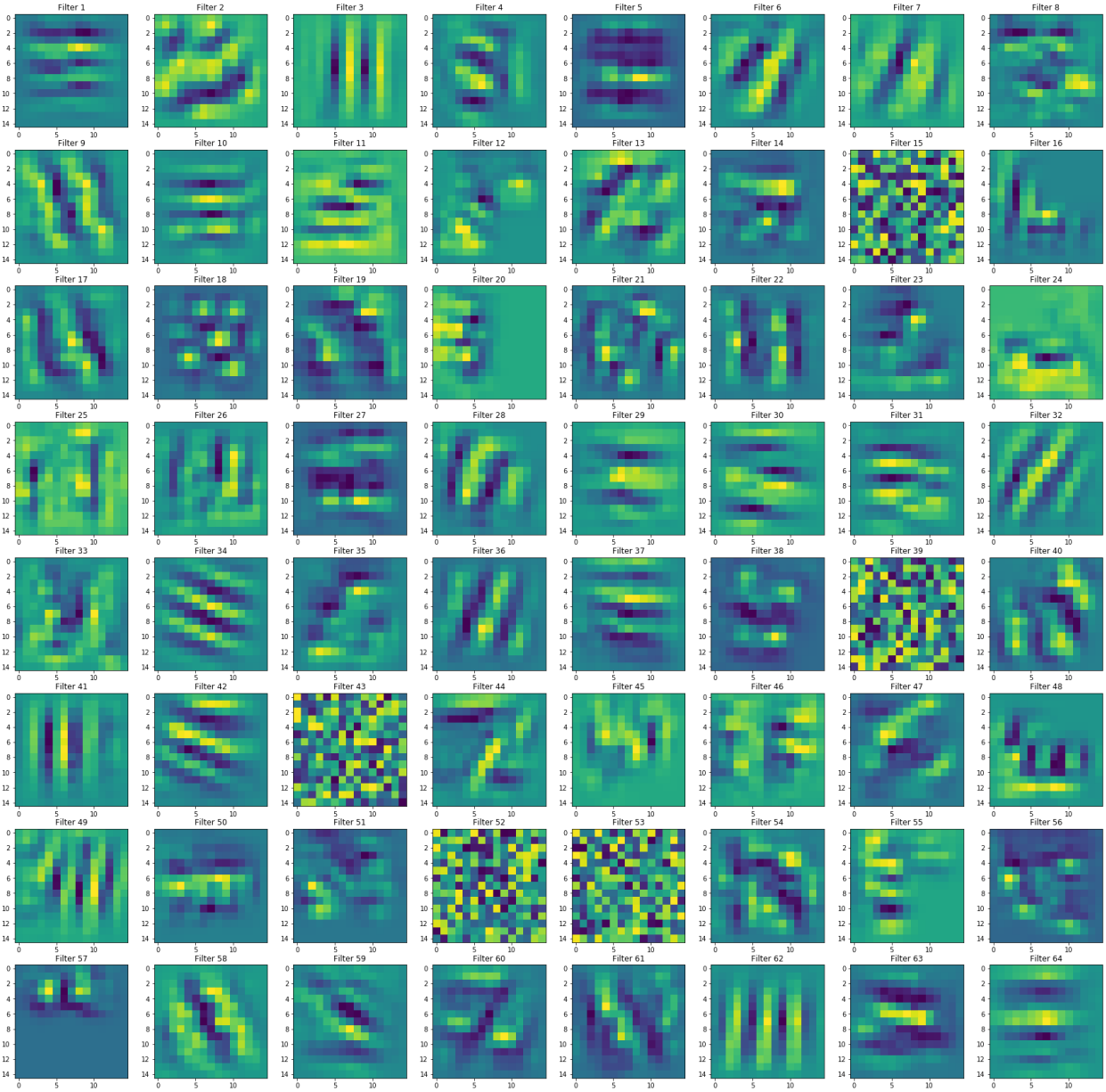
The resulted images with regularization in conv1:



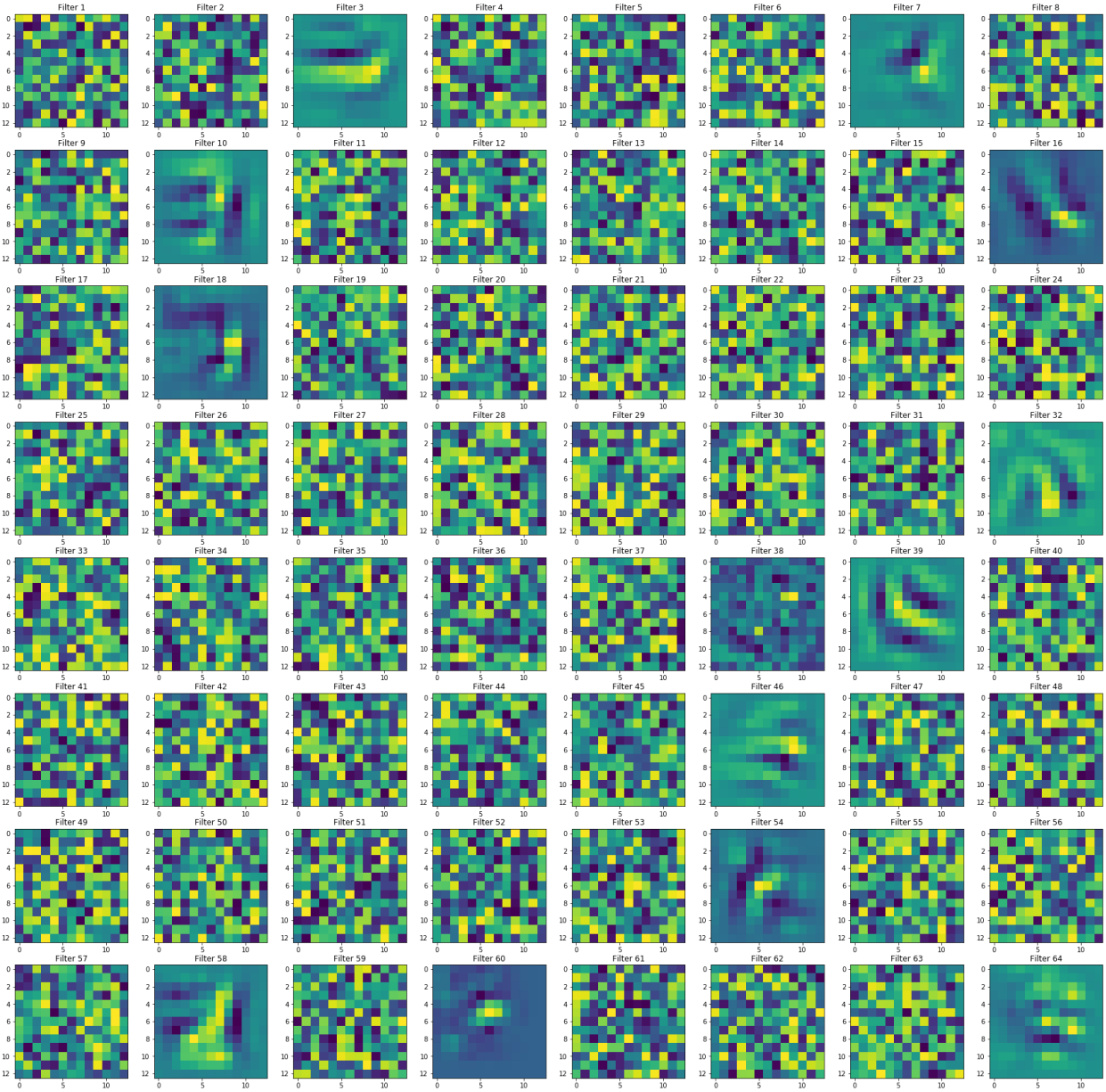
The resulted images with regularization in conv2:



The resulted images with regularization in conv3:



The resulted images with regularization in conv4:



4. The feature visualization of the CNN trained on CIFAR10 dataset

4.1 Model

In this part, we defined and trained a convolutional network model. The structure of the model is straightforward --- two convolutional layers, each with 2*2 maxpooling and a relu gate, followed by three fully connected layers and a softmax classifier. The outputs for the last fully connected layer were 10 scores for each input images.

In this part, we trained our predefined CNN model. In this part The validation accuracy is about 63% in the end, which is acceptable as our goal here is to do visualization on features to find some patterns, we used our final trained model to do so.

4.2 Feature Visualization

The details of the feature visualization are in the file Final_project_cifar10.pdf. It is also in our appendix in the few pages.

Works Cited

- Olah, Chris, et al. “Feature Visualization.” *Distill*, 23 Aug. 2018, distill.pub/2017/feature-visualization/.
- Olah, Chris, et al. “The Building Blocks of Interpretability.” *Distill*, 22 May 2018, distill.pub/2018/building-blocks/.
- Chollet, Francois. “How Convolutional Neural Networks See the World.” *The Keras Blog ATOM*, 30 Jan. 2016, blog.keras.io/how-convolutional-neural-networks-see-the-world.html.
- Allibhai, Eijaz. “Building a Convolutional Neural Network (CNN) in Keras.” *Towards Data Science*, Towards Data Science, 16 Oct. 2018, towardsdatascience.com/building-a-convolutional-neural-network-cnn-in-keras-329fbbadc5f5.
- Auygur. “Auygur/CNN-SVHN_Keras.” *GitHub*, github.com/auygur/CNN-SVHN_Keras/blob/master/main.py.
- Zailer, Matthew D., and Rob Fergus. “Visualizing and Understanding Convolutional Networks.” arxiv.org/abs/1311.2901.
- Yosinski, Jason, et al. “Understanding Neural Networks Through Deep Visualization.” arxiv.org/abs/1506.06579.
- Mahendran, Aravindh, and Andrea Vedaldi. “Understanding Deep Image Representations by Inverting Them.” 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, doi:10.1109/cvpr.2015.7299155.
- https://www.cs.toronto.edu/~kriz/cifar.html
- http://ufldl.stanford.edu/housenumbers/
- https://en.wikipedia.org/wiki/Gaussian_blur

5242 Project 3

Feature Visulization For Cifar10

1. Pre-processing and Model Construction

(1) Import Library and Define functions

Cifar10 Dataset: https://www.cs.toronto.edu/~kriz/cifar.html (https://www.cs.toronto.edu/~kriz/cifar.html)

- The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class.They were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.
- There are 50000 training images and 10000 test images.The dataset is divided into five training batches and one test batch, each with 10000 images.
- The dataset archive contains the files data_batch_1, data_batch_2, ..., data_batch_5, as well as test_batch. Each of these files is a Python "pickled" object produced with cPickle.

In [1]:
```
import tensorflow as tf
import numpy as np
from tensorflow.python.platform import gfile
from random import randint
from IPython.display import clear_output
from IPython.display import Image
from io import BytesIO
from CIFAR10_func import *
import cifar10
import dataset
import os,sys,os.path
import seaborn as sns
sns.reset_orig()
import pandas as pd
from scipy.spatial import distance
#from sklearn.manifold import TSNE
#import tensorflow.contrib.slim as slim
import matplotlib.pyplot as plt
%matplotlib inline
import math
import pickle
```

(1) Load dataset:

We have used the tutorial from Havass-Labs (https://github.com/Hvass-Labs/TensorFlow-Tutorials) to import the dataset.

In [2]:
```
cifar10.data_path = "./data/CIFAR-10/"
class_name = cifar10.load_class_names()
print(class_name)
```

```
Loading data: ./data/CIFAR-10/cifar-10-batches-py/batches.meta
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'sh
ip', 'truck']
```

In [3]:
```
train_images_cifar10, train_class_cifar10, train_labels_cifar10 = cifar10.load
_training_data()
test_images_cifar10, test_class_cifar10, test_labels_cifar10 = cifar10.load_te
st_data()
```

```
Loading data: ./data/CIFAR-10/cifar-10-batches-py/data_batch_1
Loading data: ./data/CIFAR-10/cifar-10-batches-py/data_batch_2
Loading data: ./data/CIFAR-10/cifar-10-batches-py/data_batch_3
Loading data: ./data/CIFAR-10/cifar-10-batches-py/data_batch_4
Loading data: ./data/CIFAR-10/cifar-10-batches-py/data_batch_5
Loading data: ./data/CIFAR-10/cifar-10-batches-py/test_batch
```

In [4]:
```
print("Size for the training images: {0}".format(len(train_images_cifar10)))
print("Shape for the training images: {0}".format(train_images_cifar10.shape))
print("Shape for the training labels: {0}".format(train_labels_cifar10.shape))
print("Size for the test images: {0}".format(len(test_images_cifar10)))
print("Shape for the test images: {0}".format(test_images_cifar10.shape))
print("Shape for the test labels: {0}".format(test_labels_cifar10.shape))
```

```
Size for the training images: 50000
Shape for the training images: (50000, 32, 32, 3)
Shape for the training labels: (50000, 10)
Size for the test images: 10000
Shape for the test images: (10000, 32, 32, 3)
Shape for the test labels: (10000, 10)
```

(2) CNN Model Construction using tensorflow:

In this part, we defined and trained a convolutional network model. The structure of the model is straightforward --- two convolutional layers, each with 2*2 maxpooling and a relu gate, followed by three fully connected layers and a softmax classifier. The outputs for the last fully connected layer were 10 scores for each input images.

In [ ]:
```
################ 7-Layer CNN ##############
################  CIFAR10  ###############
n1 = 32
n2 = 64
n3 = 1024
n4 = 512

x = tf.placeholder(tf.float32, [None, 32, 32, 3], name='x')
y = tf.placeholder(tf.float32, [None, 10], name='y')
keep = tf.placeholder(tf.float32,name='keep')

# CNN_layer1
# use n1 5*5 filters
W_conv1 = tf.get_variable('W_conv1', shape=[5, 5, 3, n1])
b_conv1 = tf.get_variable('b_conv1', shape=[n1])
h_conv1 = tf.nn.relu(tf.add(conv2d(x, W_conv1), b_conv1))
# Pool_layer1, maxpool to 16*16
h_pool1 = maxPool2d(h_conv1)
# CNN_layer2, takes a 16x16 with 32 layers, turns to 8x8 with 64 layers
W_conv2 = tf.get_variable('W_conv2', shape=[5, 5, n1, n2])
b_conv2 = tf.get_variable('b_conv2', shape=[n2])
h_conv2 = tf.nn.relu(tf.add(conv2d(h_pool1, W_conv2), b_conv2))
# Pool_layer2
h_pool2 = maxPool2d(h_conv2)
# FC_layer1
h_pool2_flat = tf.reshape(h_pool2, [-1, 8*8*n2])
W_fc1 = tf.get_variable('W_fc1', shape=[8*8*n2, n3])
b_fc1 = tf.get_variable('b_fc1', shape=[n3])
h_fc1 = tf.nn.relu(tf.add(tf.matmul(h_pool2_flat, W_fc1), b_fc1))
# FC_layer2
W_fc2 = tf.get_variable('W_fc2', shape=[n3, n4])
b_fc2 = tf.get_variable('b_fc2', shape=[n4])
h_fc2 = tf.nn.relu(tf.add(tf.matmul(h_fc1, W_fc2), b_fc2))
# drop_out layer
h_fc2_drop = tf.nn.dropout(h_fc2, keep)
# FC_layer3
W_fc3 = tf.get_variable('W_fc3', shape=[n4, 10])
b_fc3 = tf.get_variable('b_fc3', shape=[10])
logits = tf.add(tf.matmul(h_fc2, W_fc3), b_fc3)
loss = compute_cross_entropy(logits=logits, y=y)
accuracy = compute_accuracy(logits, y)
train_step = tf.train.AdamOptimizer(1e-4).minimize(loss)
```

(3) Train the model

In this part, we trained our predefined CNN model.In this part The validation accuracy is about 63% in the end, which is acceptabl as our goal here is to do visualization on features to find some patterns, we used our final trained model to do so.You can retrain the model from the following code, or you can just load our trained model in the output file.

In [ ]:
```
# loss = compute_cross_entropy(logits=logits, y=y)
# accuracy = compute_accuracy(logits, y)
# train_step = tf.train.AdamOptimizer(1e-4).minimize(loss)

# # Validation set
# batch_all = random_batch(data_images=test_images_cifar10, data_labels=test_l
abels_cifar10, batch_size=100)
# valid_img = batch_all[0]
# valid_lab = batch_all[1]

# batch_size =100
# input_images = train_images_cifar10
# input_labels = train_labels_cifar10
# num_step = 20001

# with tf.Session() as sess:
#         sess.run(tf.global_variables_initializer())

#         for i in range(num_step):
#             # Training set
#             batch_all = random_batch(data_images=input_images, data_labels=i
nput_labels, batch_size=batch_size)
#             train_img = batch_all[0]
#             train_lab = batch_all[1]

#             sess.run(train_step, feed_dict={x: train_img, y: train_lab, kee
p: 0.5})

#             if i%1000 == 0:
#                 valid_acu = sess.run(accuracy, {x: valid_img, y: valid_lab,
 keep:1.0})
#                 print("\rAfter step {0:3d}, validation accuracy {1:0.4f}".fo
rmat(i, valid_acu))
#             if i%10000 == 0:
#                 saver = tf.train.Saver()
#                 saver.save(sess, ".output/model_on_cifar10/", global_step=i)
```

(4) Load model

To avoid the time spent on training, you can load our model directly from following codes

In [ ]:
```
tf.reset_default_graph()
with tf.Session() as sess:
    # Load the saved model
    new_saver = tf.train.import_meta_graph("./output/model_on_cifar10/-20000.m
eta")
    new_saver.restore(sess, tf.train.latest_checkpoint('./output/model_on_cifa
r10/'))
```

Part2: Visualization of CNN model on cifar10 dataset

In [ ]:
```
from CIFAR10_func import *
```

2 Deconvolution Method:

In [ ]:
```
batch_size = 100
input_images = train_images_cifar10
input_labels = train_labels_cifar10
batch_all = random_batch(data_images=input_images,
                         data_labels=input_labels,
                         batch_size=100)
input_img = batch_all[0]
input_lab = batch_all[1]
```

2.1 Visualize the activation map of the first convolutional layer

Activation map of convolutional layer 1

In [ ]:
```
getActivations(layer_name="Relu:0", input_images=input_img, image_idx=1)
```

Activation map of convolutional layer 2

In [ ]:
```
getActivations(layer_name="Relu_1:0", input_images=input_img, image_idx=1)
```

Above plots showed the activation maps of convolutional layer 1 and convolutional layer 2. It showed that the 1st convolutional layer extract the basic features of the orginal picture so basically we can identify the orginal object though it is not that clear. However, in convolutional layer 2, the features extracted are more abstract and even some maps extract no features.We cannot know what the original objec is from activation maps of convolutional 2.

2.2 Visualize the activation maps activated the most in 1st and 2nd convolutional layer

The activation map activated the most in 1st convolutional layer

In [ ]:
```
max_activations_CONV1_cifar10(input_images=input_img, input_labels=input_lab,
batch_size=100, n1=32)
```

From the plots of the activation map activated maximally shown above we can see that, a rough figure for each image.Though it is not very clear, but it still extracts some basic features that we can basically identify what the object is. The first convolutional layer have tried to learn some outstanding patterns from input images, but it still seemed to be unsuccessful. And No.18 activation map has been shown for most the images, which at some points indicated that this filter activated maximally for most of the pictures given.

Visualize maximal activation maps of the 2nd convolutional layer

In [ ]:
```
max_activations_CONV2_cifar10(input_images=input_img, input_labels=input_lab,
batch_size=100, n1=32, n2=64)
```

From the plots shown above we can see that, the second convolutional layer seems to learn the outlines of objects from the backgrounds by the presence of distinct color patterns. But, obviously it's not very successful.And compared with the first layer, we basically cannot identify the object from the activation map that activated maximally.

2.3 Visualize saliency maps

In this part, we are trying to do saliency visualizations for given images.

In [ ]:
```
Grad_cifar10(layer_name="Add_4:0", input_images=input_img, input_labels=input_
lab, batch_size=100)
```

From the plots shown above we can see that, the outlines of the main objects for input images are not very clear, at some points due to the pool visualization for the raw images, but still can figure something out with raw images. These saliency maps at some points tell us which pixels matter for the classifications.

2.4 Visualize back-propagation maps

In this part, we are trying to do back-propagation visualization for given images from a specific layer in CNN.

Taking gradients for the 2nd maxpooling layer

In [ ]:
```
Grad_cifar10(layer_name="MaxPool_1:0", input_images=input_img, input_labels=in
put_lab, batch_size=100)
```

From the plots shown above we can see that, the outlines of the main objects for input images are not very clear, the same for the raw imags. These back-propogation maps at some points show us which pixels matter for the second maxpooling layer in our CNN.

Taking gradients for the 2nd convolutional layer

In [ ]:
```
Grad_cifar10(layer_name="Relu_1:0", input_images=input_img, input_labels=input
_lab, batch_size=100)
```

From the plots shown above we can see that, the outlines of the main objects for input images are not very clear, due to the poor visualization for input images. These back-propogation maps at some points show us which pixels matter for the second convolutional layer in our CNN.

3. Activation Maximization on Cifar10 using keras

In a CNN, each Conv layer has several learned template matching filters that maximize their output when a similar template pattern is found in the input image. The general idea of activation maximization is to generate an input image which maximizes the filter output activations. By minimizing losses during gradient descent iterations, we can understand what sort of input patterns activate a particular filter.

3.1 Preliminary Trial for Activation Maximization

To achieve this goal, first we need to define a loss function to calcualte Activation Maximization Loss. We ramdomly intialize an image as the start point of our gradient descent interation process. In each interation, we calculate the gradient of loss with respect to input image, then use it to update the input in order to reduce Activation Maximization Loss. Here we use keras package to achieve this goal.

In [5]:
```
# Network structure here is the same to what we use in tensorflow
from MaxActivation import *
from keras.layers import Conv2D, MaxPooling2D, Dropout, Dense, Flatten, Activa
tion, Input
from keras.models import Sequential
from keras import backend as K
from keras.models import load_model
IMG_WEIGHT, IMG_HEIGHT = 32, 32
model_path = './keras_saved_models/keras_cifar10_trained_model.h5'
model =  load_model(model_path)
```

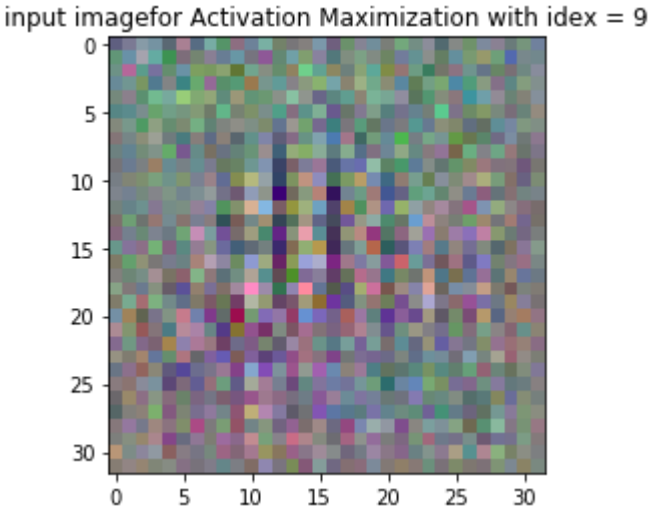
```
Using TensorFlow backend.
```

In [6]:
```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1 (Conv2D)               (None, 32, 32, 32)        2432
_________________________________________________________________
pool1 (MaxPooling2D)         (None, 16, 16, 32)        0
_________________________________________________________________
conv2 (Conv2D)               (None, 16, 16, 64)        51264
_________________________________________________________________
pool2 (MaxPooling2D)         (None, 8, 8, 64)          0
_________________________________________________________________
flatten_3 (Flatten)          (None, 4096)              0
_________________________________________________________________
fc1 (Dense)                  (None, 1024)              4195328
_________________________________________________________________
fc2 (Dense)                  (None, 512)               524800
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
preds (Dense)                (None, 10)                5130
=================================================================
Total params: 4,778,954
Trainable params: 4,778,954
Non-trainable params: 0
_________________________________________________________________
```

In [11]:
```
import matplotlib.pyplot as plt
if K.image_data_format() == 'channels_first':
    input_shape = (3, IMG_WEIGHT, IMG_HEIGHT)
elif K.image_data_format() == 'channels_last':
    input_shape = (IMG_WEIGHT, IMG_HEIGHT, 3)
#init_img = np.random.random(input_shape) * 20 + 128.0
init_img = np.zeros(input_shape)
img = deprocess_image(init_img)
```

In [13]:
```
class_idx = 9
img = init_img[np.newaxis, :]
new_img = visualize_activation_basic(model, class_idx, img, n_iteration=100, v
erbose=False)
new_img = deprocess_image(new_img)
plt.imshow(new_img[0,:])
plt.title("input imagefor Activation Maximization with idex = 9")
plt.show()
```



In [15]:
```
new_img
```

Out[15]:
```
array([[[[ 95,  95, 128],
         [123, 122, 148],
         [135, 155, 167],
         ...,
         [106, 123, 122],
         [118, 149, 124],
         [112, 129, 157]],

        [[ 97, 150, 119],
         [105, 145, 167],
         [165, 163, 174],
         ...,
         [ 83, 105, 175],
         [120, 136, 140],
         [114, 136, 141]],

        [[109, 146, 134],
         [181, 103, 154],
         [ 91, 156, 140],
         ...,
         [121, 111, 117],
         [111, 136, 130],
         [136, 140, 156]],

        ...,

        [[125, 127, 124],
         [134, 143, 128],
         [150, 135, 109],
         ...,
         [117, 113, 141],
         [124, 156, 133],
         [132, 122, 123]],

        [[185, 150, 119],
         [125, 140, 135],
         [140, 134, 156],
         ...,
         [120, 130, 103],
         [139, 142, 132],
         [125, 127, 129]],

        [[134, 125, 115],
         [146, 133, 132],
         [149, 133, 114],
         ...,
         [124, 136, 146],
         [138, 131, 101],
         [122, 137, 128]]]], dtype=uint8)
```

3.2 Increase Rate of Converge and Deal with High-frequency Patterns Using Regularization

There exist two problems in our preliminary trial above. Firstly, the image is updated at a too slow rate that we can't achieve a relevant ideal loss within 500 interation, even if we set the learning rate as 10. Secondly, images we generated above seem to be unrecognizable by humans. This is an issue regarded as the enemy of feature visualization according to Chris Olah from Google team (https://distill.pub/2017/feature-visualization/ (https://distill.pub/2017/feature-visualization/)). It often ends up with a kind of neural network optical illusion — an image full of noise and nonsensical high-frequency patterns that the network responds strongly to. To solve these problems, we explored several regularization methods and combined them to enhance the effect. Generally speaking, there are two major ideas. One is to apply some modification to the image after each optimization step so that the algorithm tends toward nicer images. The other is to modify the loss so that the learning process favors more natural images over unnatural ones.

3.2.1 Decay

This regularization method refers to the paper Understanding Neural Networks Through Deep Visualization (https://arxiv.org/abs/1506.06579 (https://arxiv.org/abs/1506.06579)). A simple regularization is to make the image closer to the mean at each step. It avoids bright pixels with very high values by penalizing large values. Decay tends to prevent a small number of extreme pixel values from dominating the example image. Such extreme single-pixel values neither occur naturally with great frequency nor are useful for visualization.

3.2.2 Blur

This regularization method refers to the paper Understanding Neural Networks Through Deep Visualization (https://arxiv.org/abs/1506.06579 (https://arxiv.org/abs/1506.06579)). Producing images via gradient ascent can arrive at high activations, but they are neither realistic nor interpretable. A useful regularization is thus to penalize high frequency information to make the image smoother.

3.2.3 Clipping pixels with small norm

This regularization method refers to Understanding Neural Networks Through Deep Visualization (https://arxiv.org/abs/1506.06579 (https://arxiv.org/abs/1506.06579)). After applied two regulations above, even if some pixels show the primary object or type of input, the gradient with respect to all other pixels will still generally be nonzero, so these pixels will also shift to show some pattern as well.Our goal in this step is the clip pixels with small norms so that we can remove their effect on maximizing the activation.

In [9]:
```
from MaxActivation import *
fg_images = []
fig_act = plt.figure(figsize=(36,15))
for class_idx in range(10):
    img = init_img[np.newaxis, :]
    new_img = visualize_activation(model, class_idx, img, n_iteration=300,verb
ose=False)
    g_images.append(deprocess_image(new_img))
    plt.subplot(2,5,class_idx+1)
    plt.imshow(new_img[0,:])
    plt.title("class %d" %(int(class_idx)+1),fontsize=30)
    plt.show()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-f73404867d0b> in <module>()
      4 for class_idx in range(10):
      5     img = init_img[np.newaxis, :]
----> 6     new_img = visualize_activation(model, class_idx, img, n_iteration
=300,verbose=False)
      7     g_images.append(deprocess_image(new_img))
      8     plt.subplot(2,5,class_idx+1)

E:\ANACONDA3\jupyternotebook\5242 project\MaxActivation.py in visualize_activ
ation(model, class_idx, init_img, n_iteration, verbose, tv_weight, lp_weight)
    219     for i in range(n_iteration):
    220         if verbose: print("iteration: {0}".format(i))
--> 221         img = gradient_ascent_iter(iterate, img, verbose=verbose,step
=1)
    222     return img
    223

E:\ANACONDA3\jupyternotebook\5242 project\MaxActivation.py in gradient_ascent
_iter(loss_fn, img, step, verbose)
    202     img_row_major = blur_regularization(img_row_major, size=(3,3))
    203     img_row_major = decay_regularization(img_row_major, decay=0.9)
--> 204     img_row_major = clip_weak_pixel_regularization(img_row_major, gra
ds_row_major)
    205     img = np.float32([np.transpose(img_row_major, (2, 0, 1))])
    206     return img

E:\ANACONDA3\jupyternotebook\5242 project\MaxActivation.py in clip_weak_pixel
_regularization(img, percentile)
    122     '''
    123     clipped = img
--> 124     threshold = np.percentile(np.abs(img), percentile)
    125     clipped[np.where(np.abs(img) < threshold)] = 0
    126     return clipped

E:\ANACONDA3\lib\site-packages\numpy\lib\function_base.py in percentile(a, q,
 axis, out, overwrite_input, interpolation, keepdims)
   3536     q = np.true_divide(q, 100.0)  # handles the asarray for us too
   3537     if not _quantile_is_valid(q):
-> 3538         raise ValueError("Percentiles must be in the range [0, 100]")
   3539     return _quantile_unchecked(
   3540         a, q, axis, out, overwrite_input, interpolation, keepdims)

ValueError: Percentiles must be in the range [0, 100]
```

<Figure size 2592x1080 with 0 Axes>

In [32]:
```
fg_images = []
fig_act = plt.figure(figsize=(36,15))
class_idx=1
img = init_img[np.newaxis, :]
new_img = visualize_activation(model, class_idx, img, n_iteration=300,verbose=
False)
# g_images.append(deprocess_image(new_img))
# plt.subplot(2,5,class_idx+1)
# plt.imshow(new_img[0,:])
# plt.title("class %d" %(int(class_idx)+1),fontsize=30)
# plt.show()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-32-ab4fee67b23e> in <module>()
      3 class_idx=1
      4 img = init_img[np.newaxis, :]
----> 5 new_img = visualize_activation(model, class_idx, img, n_iteration=300
,verbose=False)
      6 # g_images.append(deprocess_image(new_img))
      7 # plt.subplot(2,5,class_idx+1)

E:\ANACONDA3\jupyternotebook\5242 project\MaxActivation.py in visualize_activ
ation(model, class_idx, init_img, n_iteration, verbose, tv_weight, lp_weight)
    219     for i in range(n_iteration):
    220         if verbose: print("iteration: {0}".format(i))
--> 221         img = gradient_ascent_iter(iterate, img, verbose=verbose,step
=1)
    222     return img
    223

E:\ANACONDA3\jupyternotebook\5242 project\MaxActivation.py in gradient_ascent
_iter(loss_fn, img, step, verbose)
    202     img_row_major = blur_regularization(img_row_major, size=(3,3))
    203     img_row_major = decay_regularization(img_row_major, decay=0.9)
--> 204     img_row_major = clip_weak_pixel_regularization(img_row_major, gra
ds_row_major)
    205     img = np.float32([np.transpose(img_row_major, (2, 0, 1))])
    206     return img

E:\ANACONDA3\jupyternotebook\5242 project\MaxActivation.py in clip_weak_pixel
_regularization(img, percentile)
    122     '''
    123     clipped = img
--> 124     threshold = np.percentile(np.abs(img), percentile)
    125     clipped[np.where(np.abs(img) < threshold)] = 0
    126     return clipped

E:\ANACONDA3\lib\site-packages\numpy\lib\function_base.py in percentile(a, q,
 axis, out, overwrite_input, interpolation, keepdims)
   3536     q = np.true_divide(q, 100.0)  # handles the asarray for us too
   3537     if not _quantile_is_valid(q):
-> 3538         raise ValueError("Percentiles must be in the range [0, 100]")
   3539     return _quantile_unchecked(
   3540         a, q, axis, out, overwrite_input, interpolation, keepdims)

ValueError: Percentiles must be in the range [0, 100]
```

```
<Figure size 2592x1080 with 0 Axes>
```

8.3 Check Our Implementation Above

Unfortunately, the images generated above look still not natural, while they are much better than those from our preliminary trial. It may leads to a doubt whether we implement Activation Maximization correctly. We do further work to check whether our code can successfully generate an image with Activation Maximization to help us achieve goal of feature visualization. It turns out that our code does work.

3.3.1 High Confidence Predictions for Unrecognizable Images

We firstly check the score output when we put these images into our model.

In [ ]:
```
for class_idx in range(10):
scores = model.predict(g_images[class_idx])
print("score distribution when we set class as %d"% class_idx )
print(scores)
```

As we can see, these images all have a very high confidence rate in their respective classes. In other words, images score very high for a single class even if they are unrecognizable by humans.

3.4 Conclusion for Activation Maximization

We used Activation Maximization to generate images for visualizing features. After preliminary trial, we also explored and combined 5 regularization to improve recognition. Given the fact that images were still hard to recognize for haman beings, we did further exploration. To confirm that we implemented the method correctly, we tested the output score of images from each class, as well as applied it on dataset MNIST and compared the performance. They both proved that our method could successfully generate an image given specific class, which achieve a high score for the assigned class. Besides, its relevant good performance on MNIST reveals that this method might fit some dataset better. We may expore this issue in the future.

CNN visualization of Cifar10 using keras

zx2229

We have used tensorflow to do visualization of Cifar10 datasets. Since there are many visualization tools in keras, so it will be easier to use keras to do visualization on Cifar10 dataset. Here we will skip some parts we have introduded in tensorflow model part.The code here we refer to the book *Deep learning with python* (https://www.manning.com/books/deep-learning-with-python).

Load Cifar10 dataset

You should download the dataset into assigned path to load data

In [1]:
```
import cifar10
import dataset
import cache
```

In [2]:
```
cifar10.data_path = "./data/CIFAR-10/"
class_name = cifar10.load_class_names()
print(class_name)
```

```
Loading data: ./data/CIFAR-10/cifar-10-batches-py/batches.meta
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'sh
ip', 'truck']
```

In [3]:
```
train_images_cifar10, train_class_cifar10, train_labels_cifar10 = cifar10.load
_training_data()
test_images_cifar10, test_class_cifar10, test_labels_cifar10 = cifar10.load_te
st_data()
```

```
Loading data: ./data/CIFAR-10/cifar-10-batches-py/data_batch_1
Loading data: ./data/CIFAR-10/cifar-10-batches-py/data_batch_2
Loading data: ./data/CIFAR-10/cifar-10-batches-py/data_batch_3
Loading data: ./data/CIFAR-10/cifar-10-batches-py/data_batch_4
Loading data: ./data/CIFAR-10/cifar-10-batches-py/data_batch_5
Loading data: ./data/CIFAR-10/cifar-10-batches-py/test_batch
```

In [4]:
```
print("Size for the training images: {0}".format(len(train_images_cifar10)))
print("Shape for the training images: {0}".format(train_images_cifar10.shape))
print("Shape for the training labels: {0}".format(train_labels_cifar10.shape))
print("Size for the test images: {0}".format(len(test_images_cifar10)))
print("Shape for the test images: {0}".format(test_images_cifar10.shape))
print("Shape for the test labels: {0}".format(test_labels_cifar10.shape))
```

```
Size for the training images: 50000
Shape for the training images: (50000, 32, 32, 3)
Shape for the training labels: (50000, 10)
Size for the test images: 10000
Shape for the test images: (10000, 32, 32, 3)
Shape for the test labels: (10000, 10)
```

1.Triain model

We trained two CNN models here for cifar10 data in this part. The first one has exactly the same structure as the one we trained in tensorflow, which is saved as **keras_cifar10_trained_model**, and the other one is from the keras official team,cifar10_cnn (https://github.com/keras-team/keras/blob/master/examples/cifar10_cnn.py), which is saved as **keras_cifar10_trained_model_2**, both of them are in the file **keras_saved_models**. The latter one gets a 79% classfication accuracy result after 50 epochs, each of which has 1000 steps.Since the goal here is to do visualization on features and try to find the pattern, the performance of the CNN model here is not that important.We have comment out the code here, but you can retrain this part if you want, or you can directly load our trained model.

In [5]:
```
# '''Train a simple deep CNN on the CIFAR10 small images dataset.

# It gets to 75% validation accuracy in 25 epochs, and 79% after 50 epochs.
# (it's still underfitting at that point, though).
# '''

# from __future__ import print_function
# import keras
# from keras.datasets import cifar10
# from keras.preprocessing.image import ImageDataGenerator
# from keras.models import Sequential
# from keras.layers import Dense, Dropout, Activation, Flatten
# from keras.layers import Conv2D, MaxPooling2D
# import os

# batch_size = 32
# num_classes = 10
# epochs = 50
# data_augmentation = True
# num_predictions = 20
# save_dir = os.path.join(os.getcwd(), 'keras_saved_models')
# model_name = 'keras_cifar10_trained_model_2.h5'

# # The data, split between train and test sets:
# (x_train, y_train), (x_test, y_test) = cifar10.load_data()
# print('x_train shape:', x_train.shape)
# print(x_train.shape[0], 'train samples')
# print(x_test.shape[0], 'test samples')

# # Convert class vectors to binary class matrices.
# y_train = keras.utils.to_categorical(y_train, num_classes)
# y_test = keras.utils.to_categorical(y_test, num_classes)

# ################### the CNN model from keras official team ###############

# model = Sequential()
# model.add(Conv2D(32, (3, 3), padding='same',
#                  input_shape=x_train.shape[1:]))
# model.add(Activation('relu'))
# model.add(Conv2D(32, (3, 3)))
# model.add(Activation('relu'))
# model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Dropout(0.25))

# model.add(Conv2D(64, (3, 3), padding='same'))
# model.add(Activation('relu'))
# model.add(Conv2D(64, (3, 3)))
# model.add(Activation('relu'))
# model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Dropout(0.25))

# model.add(Flatten())
# model.add(Dense(512))
# model.add(Activation('relu'))
# model.add(Dropout(0.5))
# model.add(Dense(num_classes))
```

```
# model.add(Activation('softmax'))

# ############## Exactly the same as  the cnn model trained in tensorflow ####
####################

# # n1 = 32
# # n2 = 64
# # n3 = 1024
# # n4 = 512

# # model = Sequential()
# # model.add(Conv2D(n1, (5, 5), input_shape=x_train.shape[1:], activation='re
lu', padding='same', name='conv1'))
# # model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same', n
ame='pool1'))
# # model.add(Conv2D(n2, (5, 5), activation='relu', padding='same', name='conv
2'))
# # model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same', n
ame='pool2'))
# # model.add(Flatten())
# # model.add(Dense(n3, activation='relu', name='fc1'))
# # model.add(Dense(n4, activation='relu', name='fc2'))
# # model.add(Dropout(0.5, noise_shape=None, seed=None))
# # model.add(Dense(10, activation='softmax', name='preds'))
# ################################################################################
##########################

# # initiate RMSprop optimizer
# opt = keras.optimizers.rmsprop(lr=0.0001, decay=1e-6)

# # Let's train the model using RMSprop
# model.compile(loss='categorical_crossentropy',
#               optimizer=opt,
#               metrics=['accuracy'])

# x_train = x_train.astype('float32')
# x_test = x_test.astype('float32')
# x_train /= 255
# x_test /= 255

# if not data_augmentation:
#     print('Not using data augmentation.')
#     model.fit(x_train, y_train,
#               batch_size=batch_size,
#               epochs=epochs,
#               validation_data=(x_test, y_test),
#               shuffle=True)
# else:
#     print('Using real-time data augmentation.')
#     # This will do preprocessing and realtime data augmentation:
#     datagen = ImageDataGenerator(
#         featurewise_center=False,  # set input mean to 0 over the dataset
#         samplewise_center=False,  # set each sample mean to 0
#         featurewise_std_normalization=False,  # divide inputs by std of the
 dataset
#         samplewise_std_normalization=False,  # divide each input by its std
#         zca_whitening=False,  # apply ZCA whitening
```

```
#         zca_epsilon=1e-06,  # epsilon for ZCA whitening
#         rotation_range=0,  # randomly rotate images in the range (degrees, 0
 to 180)
#         # randomly shift images horizontally (fraction of total width)
#         width_shift_range=0.1,
#         # randomly shift images vertically (fraction of total height)
#         height_shift_range=0.1,
#         shear_range=0.,  # set range for random shear
#         zoom_range=0.,  # set range for random zoom
#         channel_shift_range=0.,  # set range for random channel shifts
#         # set mode for filling points outside the input boundaries
#         fill_mode='nearest',
#         cval=0.,  # value used for fill_mode = "constant"
#         horizontal_flip=True,  # randomly flip images
#         vertical_flip=False,  # randomly flip images
#         # set rescaling factor (applied before any other transformation)
#         rescale=None,
#         # set function that will be applied on each input
#         preprocessing_function=None,
#         # image data format, either "channels_first" or "channels_last"
#         data_format=None,
#         # fraction of images reserved for validation (strictly between 0 and
 1)
#         validation_split=0.0)

#     # Compute quantities required for feature-wise normalization
#     # (std, mean, and principal components if ZCA whitening is applied).
#     datagen.fit(x_train)

#     # Fit the model on the batches generated by datagen.flow().
#     model.fit_generator(datagen.flow(x_train, y_train,
#                                      batch_size=batch_size),
#                         steps_per_epoch = 1000,
#                         epochs=epochs,
#                         validation_data=(x_test, y_test),
#                         workers=4)

# # Save model and weights
# if not os.path.isdir(save_dir):
#     os.makedirs(save_dir)
# model_path = os.path.join(save_dir, model_name)
# model.save(model_path)
# print('Saved trained model at %s ' % model_path)

# # Score trained model.
# scores = model.evaluate(x_test, y_test, verbose=1)
# print('Test loss:', scores[0])
# print('Test accuracy:', scores[1])
```

In [6]:
```
import keras
keras.__version__
```

```
Using TensorFlow backend.
```

Out[6]: '2.2.4'

2. Import trained model

We trained two keras models in previous steps, and we use the second one, which is from the keras official team here.

In [7]:
```
from keras.models import load_model

model = load_model('./keras_saved_models/keras_cifar10_trained_model_2.h5')
model.summary()  # As a reminder to show the strucure of CNN model.
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
activation_7 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
activation_8 (Activation)    (None, 30, 30, 32)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
dropout_7 (Dropout)          (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 15, 15, 64)        18496
_________________________________________________________________
activation_9 (Activation)    (None, 15, 15, 64)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 13, 13, 64)        36928
_________________________________________________________________
activation_10 (Activation)   (None, 13, 13, 64)        0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout_8 (Dropout)          (None, 6, 6, 64)          0
_________________________________________________________________
flatten_5 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_3 (Dense)              (None, 512)               1180160
_________________________________________________________________
activation_11 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_9 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_12 (Activation)   (None, 10)                0
=================================================================
Total params: 1,250,858
Trainable params: 1,250,858
Non-trainable params: 0
_________________________________________________________________
```

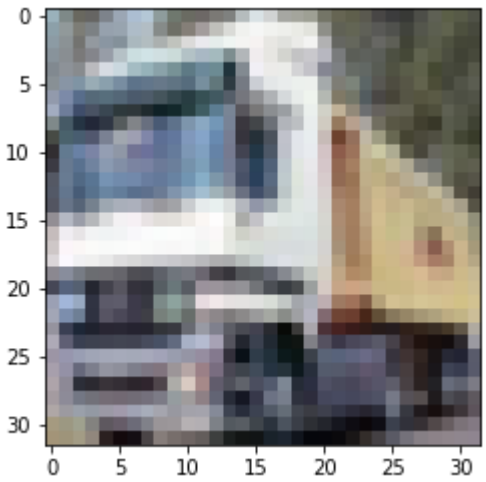
3. Feature Visualization

When fed an image input, above model returns the values of the layer activations in the original model. We will use one picture here to do feature visualization. In this part, we managed to do visualizaiton on trained CNN model,including each layer's visualization and activation maximization.

In [8]:
```
# We preprocess the image into a 4D tensor
from keras.preprocessing import image
import numpy as np
img = train_images_cifar10[1]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
# Remember that the model was trained on inputs
# that were preprocessed in the following way:
print(img_tensor.shape)
```

```
(1, 32, 32, 3)
```

In [10]:
```
# Use one image to do visualizaion
import matplotlib.pyplot as plt
plt.imshow(img_tensor[0])
plt.show() # original image
```



In [11]:
```
from keras import models

# Extracts the outputs of the top 19 layers:
layer_outputs = [layer.output for layer in model.layers[:19]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

In [12]:
```
# This will return a list of 18 Numpy arrays:
# one array per layer activation
activations = activation_model.predict(img_tensor)
len(activations)
```

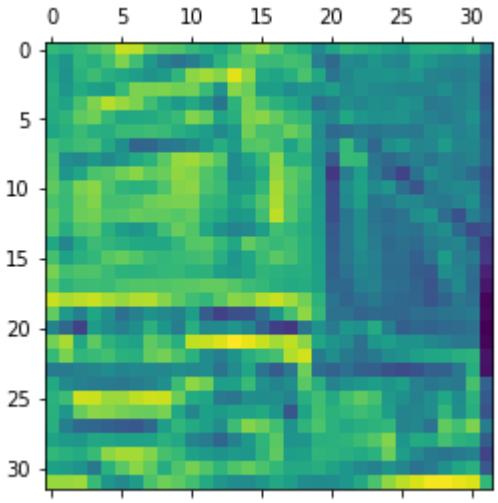
Out[12]: 18

In [13]:
```
# the first layer activation dimesion
first_layer_activation = activations[0]
print(first_layer_activation.shape)
```

```
(1, 32, 32, 32)
```

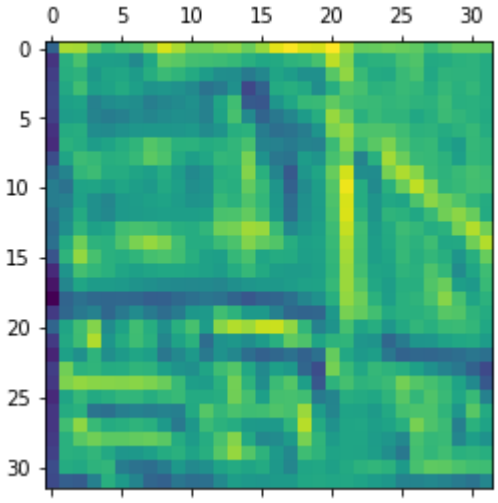
We can see that the first feature map is a 32*32 feature map with 32 channels. Let's try visualizing the 3rd channel:

In [14]:
```
import matplotlib.pyplot as plt

plt.matshow(first_layer_activation[0, :, :, 3], cmap='viridis')
plt.show()
```



The visualizaiton of the 30th channel:

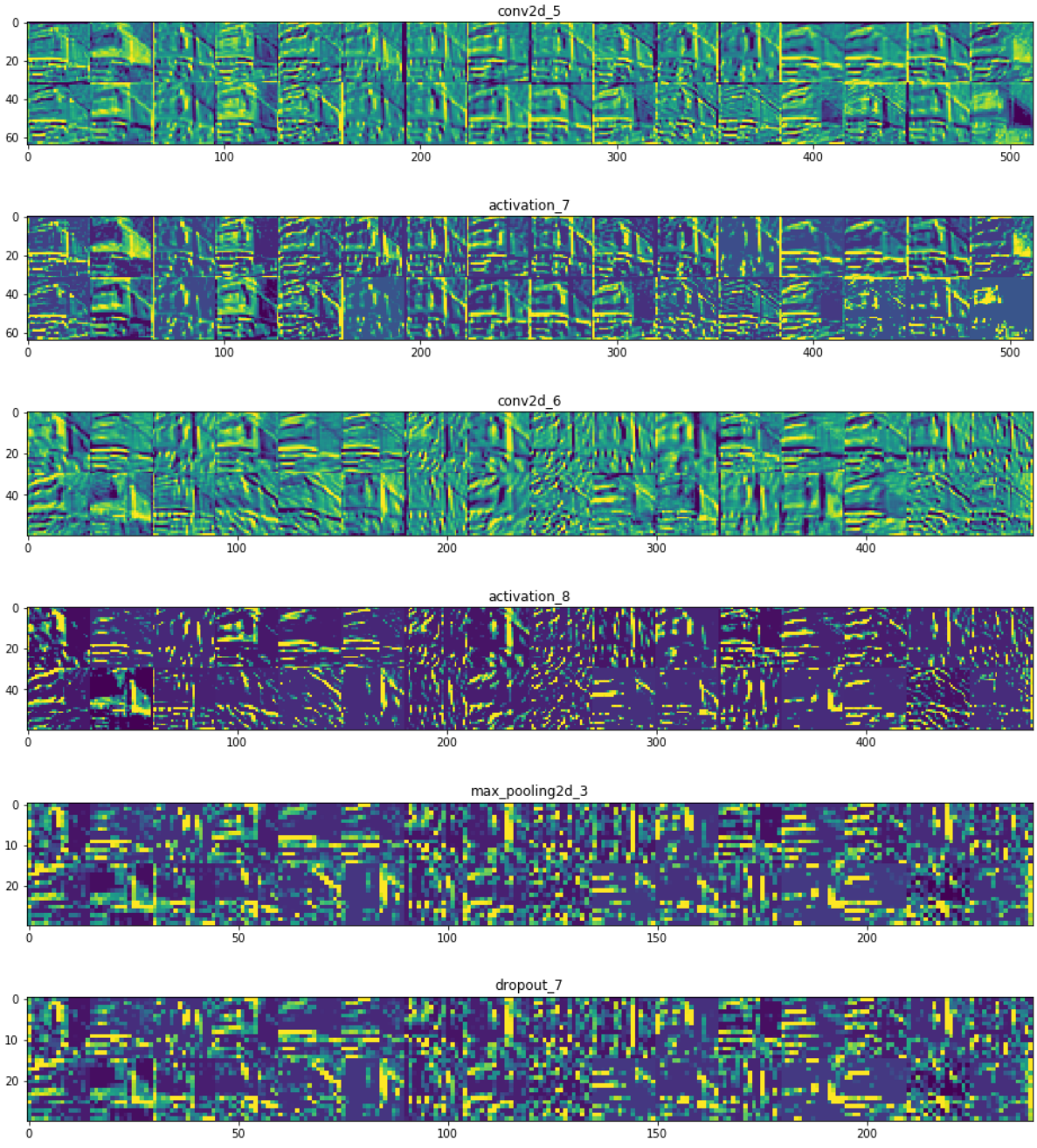
In [15]:
```
plt.matshow(first_layer_activation[0, :, :, 30], cmap='viridis')
plt.show()
```

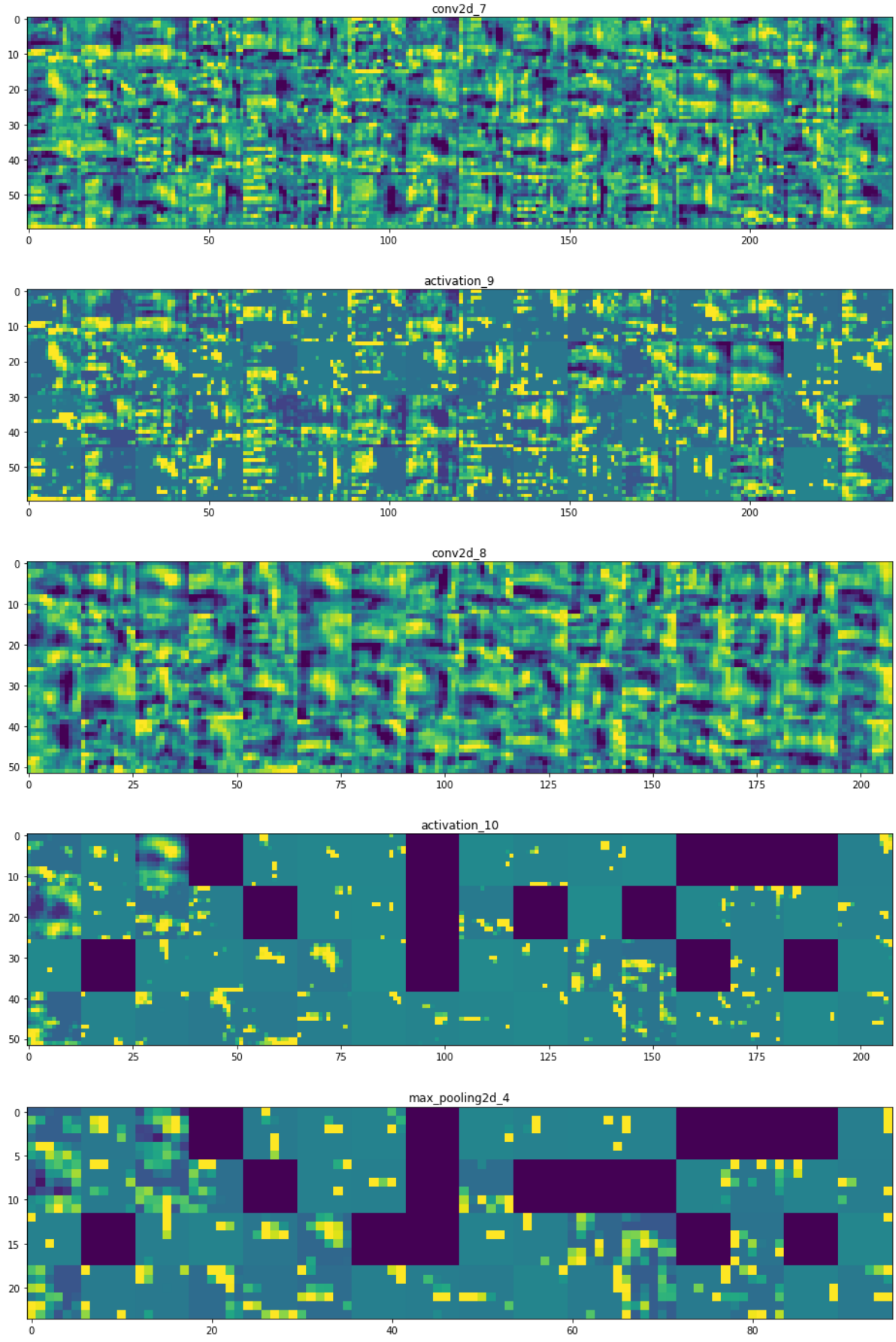


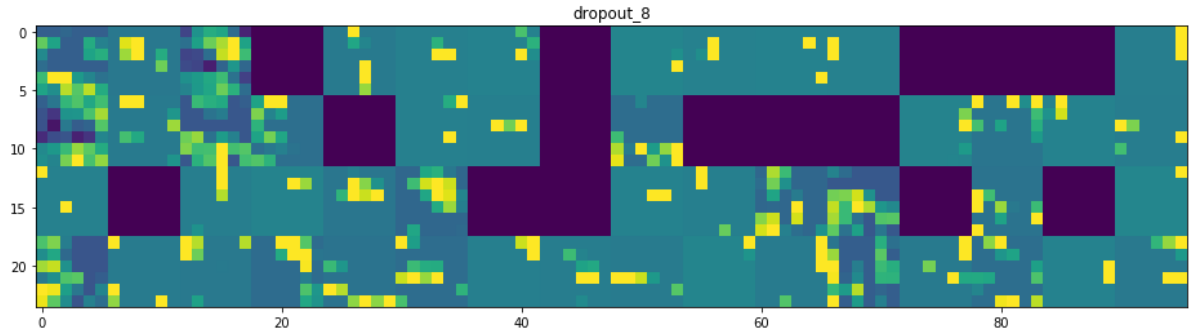
Each layer's visualization

In [16]:
```
import keras
import warnings
warnings.filterwarnings('ignore')
# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:18]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            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

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                        scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show()
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-16-968e7e9a07c9> in <module>()
     19     # We will tile the activation channels in this matrix
     20     n_cols = n_features // images_per_row
---> 21     display_grid = np.zeros((size * n_cols, images_per_row * size))
     22
     23     # We'll tile each filter into this big horizontal grid

MemoryError:
```




conv2d_7
activation_9
conv2d_8
activation_10
max_pooling2d_4



This part we visualized each layer of the whole CNN model.Same as the result of model implemented by Tensorflow, we can find that:

- The first layer acts as a collection of various edge detectors. At that stage, the activations are still retaining almost all of the information present in the initial picture.
- As we go higher-up, the activations become increasingly abstract and less visually interpretable. They start encoding higher-level concepts. Higher-up presentations carry increasingly less information about the visual contents of the image, and increasingly more information related to the class of the image.
- The sparsity of the activations is increasing with the depth of the layer: in the first layer, all filters are activated by the input image, but in the following layers more and more filters are blank. This means that the pattern encoded by the filter isn't found in the input image.

Activation Maximization

In a CNN, each Conv layer has several learned template matching filters that maximize their output when a similar template pattern is found in the input image. The general idea of activation maximization is to generate an input image which maximizes the filter output activations. By minimizing losses during gradient descent iterations, we can understand what sort of input patterns activate a particular filter.

In [17]:
```
from keras import backend as K

layer_name = 'conv2d_5'
filter_index = 0

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

To implement gradient descent, we will need the gradient of this loss with respect to the model's input. To do this, we will use the gradients function packaged with the backend module of Keras:

In [18]:
```
# The call to `gradients` returns a list of tensors (of size 1 in this case)
# hence we only keep the first element -- which is a tensor.
grads = K.gradients(loss, model.input)[0]
```

A non-obvious trick to use for the gradient descent process to go smoothly is to normalize the gradient tensor, by dividing it by its L2 norm (the square root of the average of the square of the values in the tensor). This ensures that the magnitude of the updates done to the input image is always within a same range.

In [19]:
```
# We add 1e-5 before dividing so as to avoid accidentally dividing by 0.
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
```

Now we need a way to compute the value of the loss tensor and the gradient tensor, given an input image. We can define a Keras backend function to do this: iterate is a function that takes a Numpy tensor (as a list of tensors of size 1) and returns a list of two Numpy tensors: the loss value and the gradient value.

In [20]:
```
iterate = K.function([model.input], [loss, grads])

# Let's test it:
import numpy as np
loss_value, grads_value = iterate([np.zeros((1, 150, 150, 3))])
```

At this point we can define a Python loop to do stochastic gradient descent:

In [21]:
```
# We start from a gray image with some noise
input_img_data = np.random.random((1, 32, 32, 3)) * 20 + 128.

# Run gradient ascent for 40 steps
step = 1.  # this is the magnitude of each gradient update
for i in range(40):
    # Compute the loss value and gradient value
    loss_value, grads_value = iterate([input_img_data])
    # Here we adjust the input image in the direction that maximizes the loss
    input_img_data += grads_value * step
```

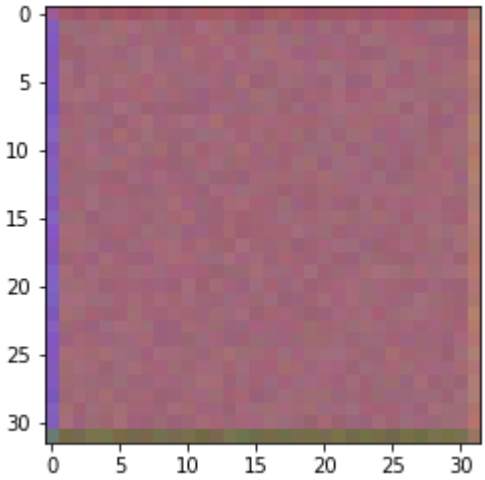
The resulting image tensor will be a floating point tensor of shape (1, 150, 150, 3), with values that may not be integer within [0, 255]. Hence we would need to post-process this tensor to turn it into a displayable image. We do it with the following straightforward utility function:

In [22]:
```
def deprocess_image(x):
    # normalize tensor: center on 0., ensure std is 0.1
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    # clip to [0, 1]
    x += 0.5
    x = np.clip(x, 0, 1)

    # convert to RGB array
    x *= 255
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

Now we have all the pieces, let's put them together into a Python function that takes as input a layer name and a filter index, and that returns a valid image tensor representing the pattern that maximizes the activation the specified filter:

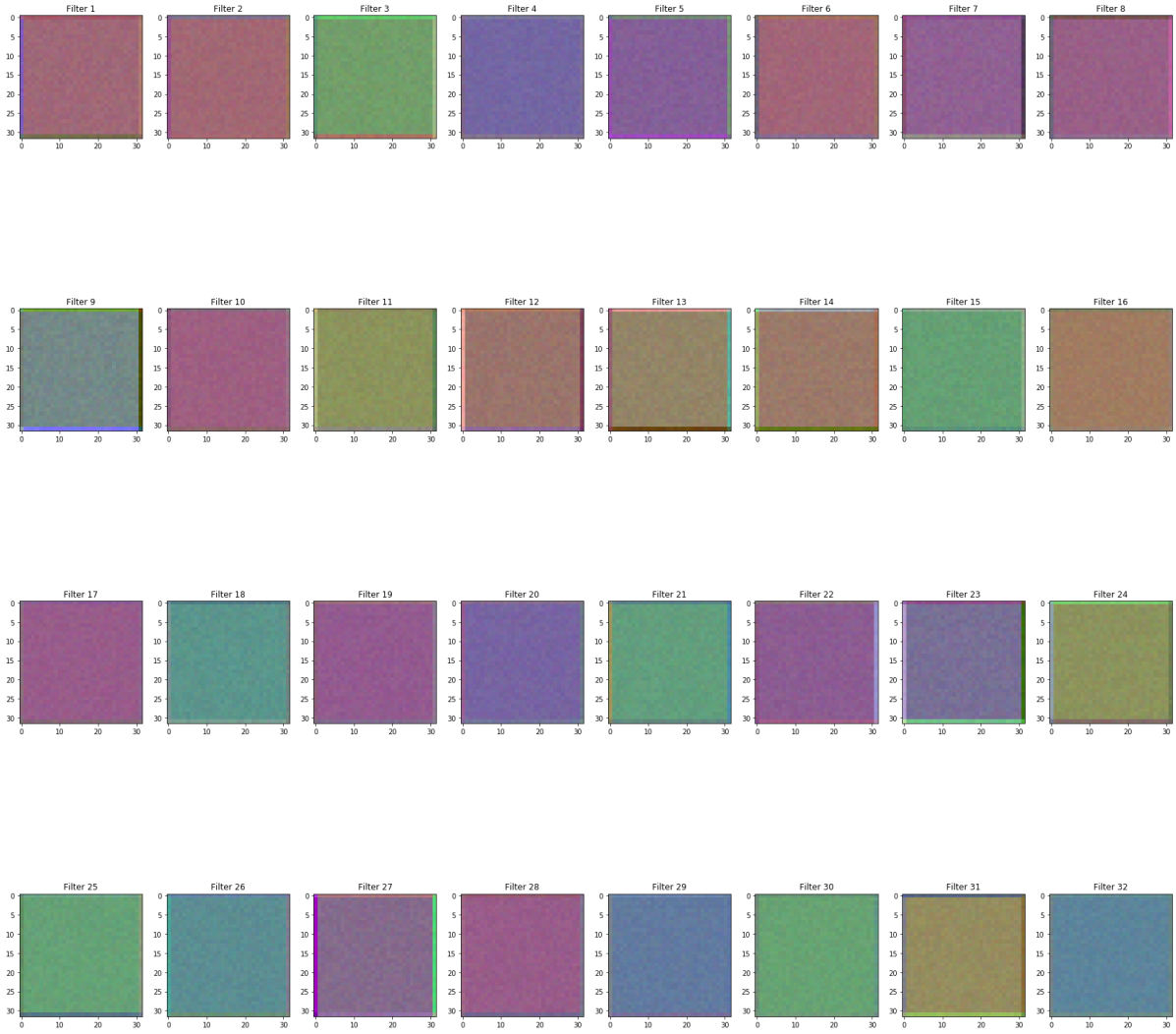
In [23]:
```
def generate_pattern(layer_name, filter_index, size=32):
    # Build a loss function that maximizes the activation
    # of the nth filter of the layer considered.
    layer_output = model.get_layer(layer_name).output
    loss = K.mean(layer_output[:, :, :, filter_index])

    # Compute the gradient of the input picture wrt this loss
    grads = K.gradients(loss, model.input)[0]

    # Normalization trick: we normalize the gradient
    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)

    # This function returns the loss and grads given the input picture
    iterate = K.function([model.input], [loss, grads])

    # We start from a gray image with some noise
    input_img_data = np.random.random((1, size, size, 3)) * 20 + 128.

    # Run gradient ascent for 40 steps
    step = 1.
    for i in range(40):
        loss_value, grads_value = iterate([input_img_data])
        input_img_data += grads_value * step

    img = input_img_data[0]
    return deprocess_image(img)
```

In [24]:
```
plt.imshow(generate_pattern('conv2d_5', 0))
plt.show() #
```

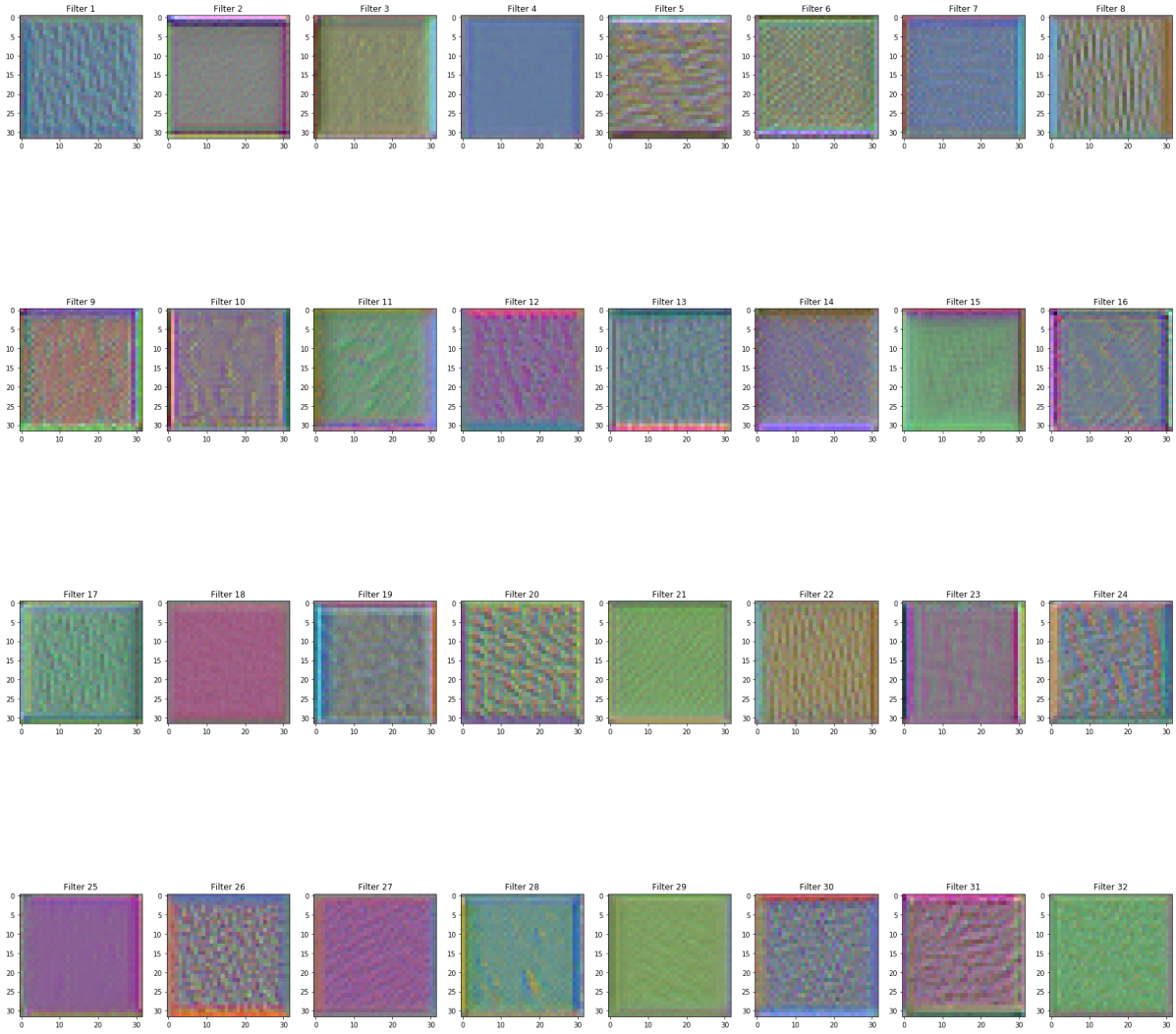


We can start visualising images which maximumly activate each unit in each convolutional layer .

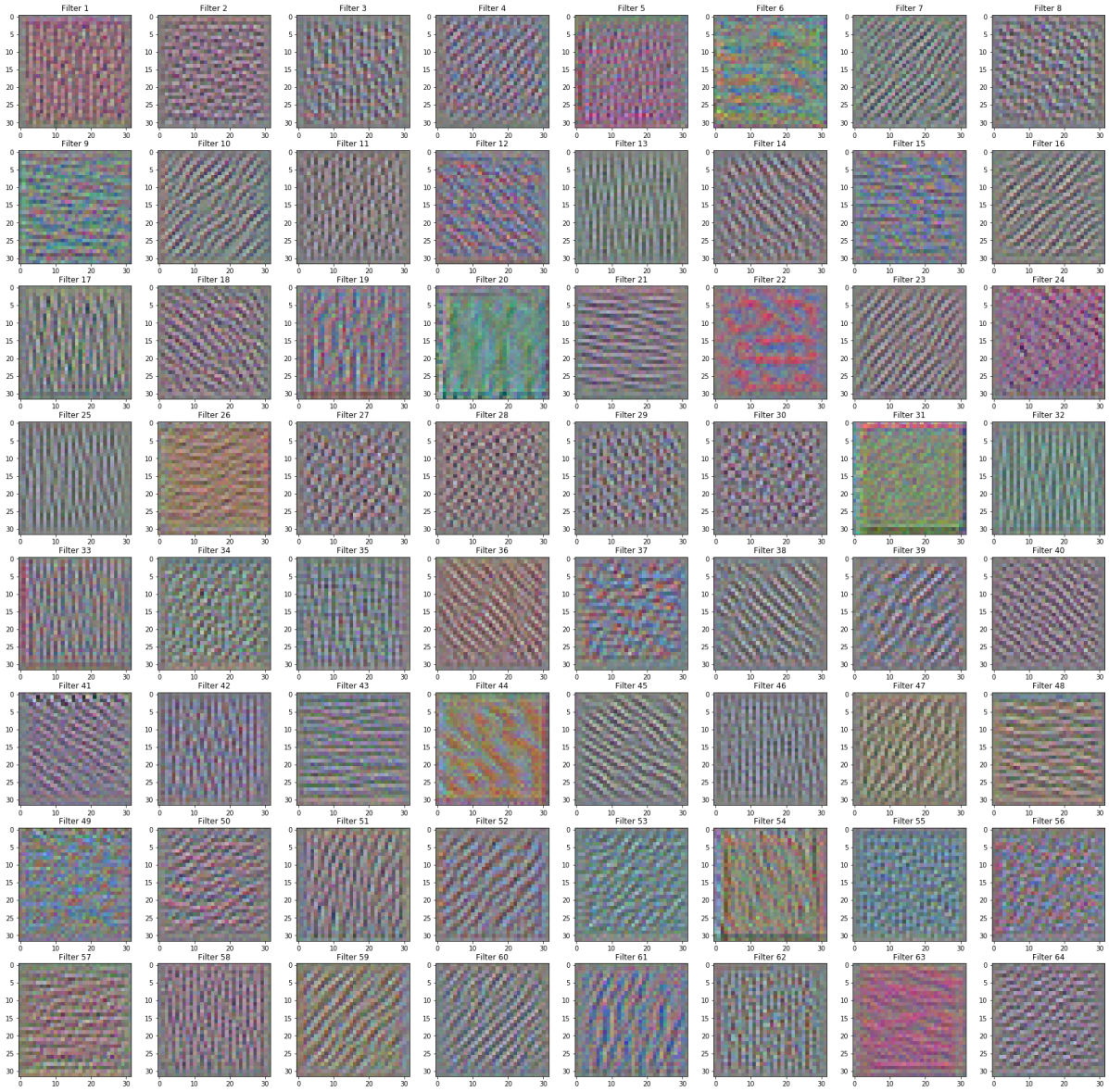
In [25]:
```
# Show the input image that maximaly activate the 32 filters in conv1.
plt.figure(1, figsize=(30,30))
n_columns = 8
n_rows = 4
for i in range(32):
    plt.subplot(n_rows, n_columns, i+1)
    plt.title('Filter ' + str(i+1))
    plt.imshow(generate_pattern('conv2d_5', i))
```



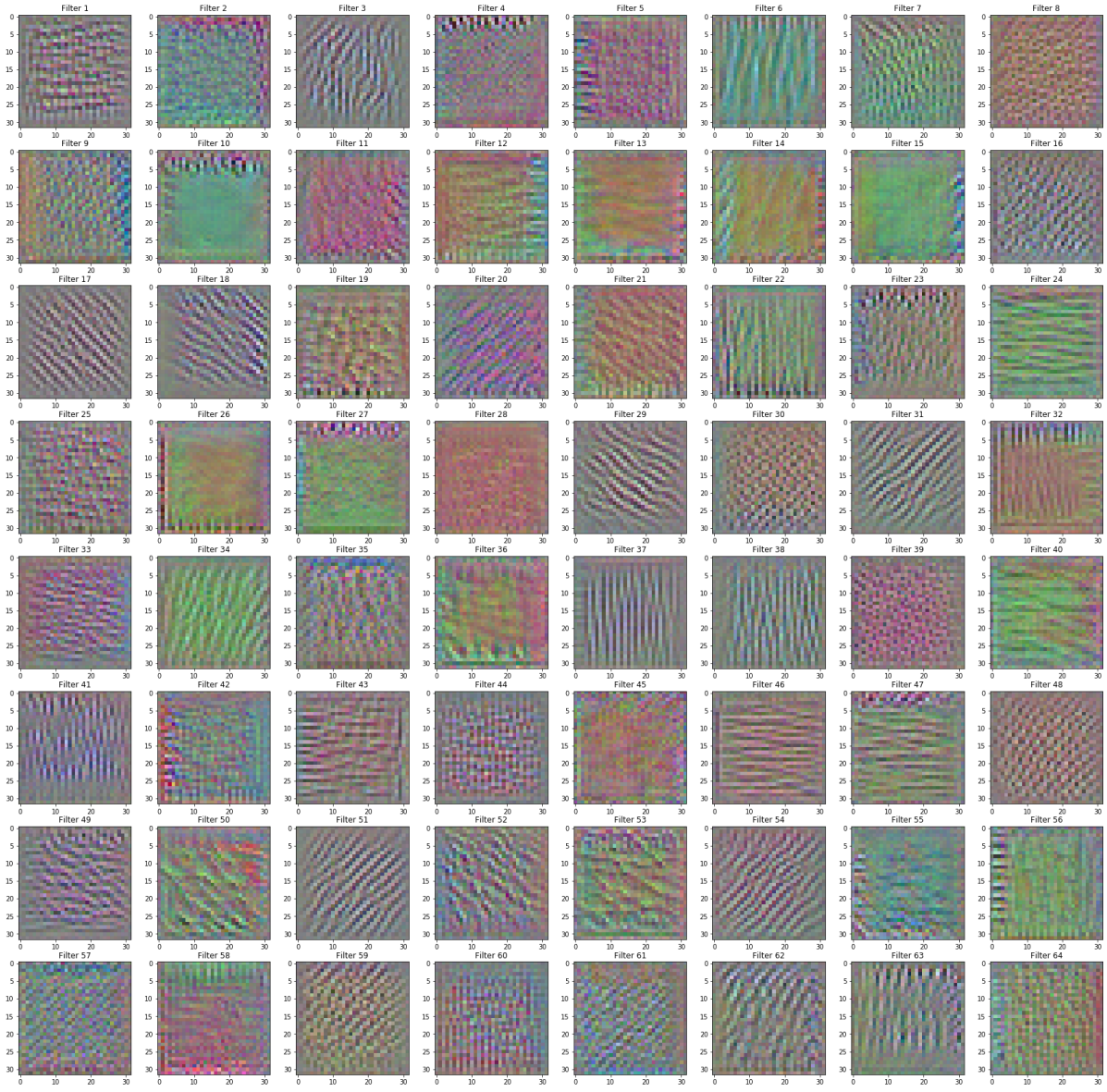
In [26]:
```
# Show the input image that maximaly activate the 32 filters in conv2.
plt.figure(1, figsize=(30,30))
n_columns = 8
n_rows = 4
for i in range(32):
    plt.subplot(n_rows, n_columns, i+1)
    plt.title('Filter ' + str(i+1))
    plt.imshow(generate_pattern('conv2d_6', i))
```



In [27]:
```
# Show the input image that maximaly activate the 64 filters in conv3.
plt.figure(1, figsize=(30,30))
n_columns = 8
n_rows = 8
for i in range(64):
    plt.subplot(n_rows, n_columns, i+1)
    plt.title('Filter ' + str(i+1))
    plt.imshow(generate_pattern('conv2d_7', i))
```



In [28]:
```
# Show the input image that maximaly activate the 64 filters in conv4.
plt.figure(1, figsize=(30,30))
n_columns = 8
n_rows = 8
for i in range(64):
    plt.subplot(n_rows, n_columns, i+1)
    plt.title('Filter ' + str(i+1))
    plt.imshow(generate_pattern('conv2d_8', i))
```



These filter visualizations tell us a lot about how convnet layers see the world. Here we optimized to find images that excited different unit in each layer. This is similar to how the Fourier transform decomposes signals onto a bank of cosine functions. Since the original images of cifar10 pictures have low resolution, we cannot make sure what we have found by doing activation maximization. After reviewing relative chapters in Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python), we find that the images which can activate different convolutional layer's unit most get increasingly complex and refined as we go higher-up in the model:

- The filters from the first layer in the model just encode simple directional edges and colors (or colored edges in some cases).
- The filters from second convolutional layer encode simple textures made from combinations of edges and colors.
- The filters in higher-up layers start resembling textures found in 10 class images.