Created by : Utsav Patel

Mounted Google Drive in order to load data from drive

```
from google.colab import drive
drive.mount('/content/drive',force_remount=True)
```

```
Mounted at /content/drive
```

finetune.py and train phase

```
!python finetune.py --train
```

```
Epoch:  0
Accuracy:  1.0
Epoch:  1
Accuracy:  1.0
Epoch:  2
Accuracy:  1.0
Epoch:  3
```

Note: For Simplicity I have taken only one image as test image and only 20 images as train image(18 cats and 2 dogs). Thus some results and vectors plotted might seem little bit apart from usual real case scenario.

```
Accuracy:  1.0
```

finetune.py and Prune Phase

```
!python finetune.py --prune
```

```
Layers that will be prunned {24: 63, 21: 76, 26: 87, 12: 30, 17: 64, 28: 44, 19: 71,
Prunning filters..
Filters prunned 39.39393939393939%
Accuracy:  1.0
Fine tuning to recover from prunning iteration.
Epoch:  0
Accuracy:  1.0
Epoch:  1
Accuracy:  1.0
Epoch:  2
Accuracy:  1.0
Epoch:  3
Accuracy:  1.0
Epoch:  4
Accuracy:  1.0
Epoch:  5
Accuracy:  1.0
Epoch:  6
Accuracy:  1.0
Epoch:  7
Accuracy:  1.0
Epoch:  8
```

model_prunned is the model where we stored our prunned model. SO we are loading that model here.

```
1 pre_trained_model=torch.load("model_prunned")
```

```
/usr/local/lib/python3.6/dist-packages/torch/serialization.py:401: UserWarning: Could
  "type " + container_type.__name__ + ". It won't be checked "
```

Please see the configuration of Pruned Model

```
1 pre_trained_model.eval()
```

```
ModifiedVGG16Model(
  (classifier): Sequential(
    (0): Dropout(p=0.5)
    (1): Linear(in_features=2744, out_features=4096, bias=True)
    (2): ReLU(inplace)
    (3): Dropout(p=0.5)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace)
    (6): Linear(in_features=4096, out_features=2, bias=True)
  )
  (features): Sequential(
    (0): Conv2d(3, 41, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(41, 49, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(49, 90, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(90, 94, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(94, 167, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(167, 159, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(159, 154, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(154, 231, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(231, 204, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

Note: I have flattened the N Dimensional Array to plot the weights at layers 1 to 5.

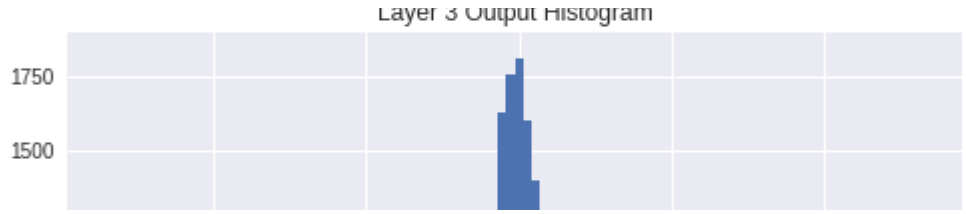
```
    (25): ReLU(inplace)
```

:Weight Histograms are plotted below

```
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
from matplotlib.pyplot import figure
import matplotlib.pyplot as plt
%matplotlib inline
figure(num=None, figsize=(8, 30))

plt.style.use('seaborn-deep')
plt.subplot(5, 1, 1)
plt.hist(list(list(pre_trained_model.parameters())[0].flatten().data.cpu().numpy()),bins
plt.title('Layer 1 Output Histogram')
plt.subplot(5, 1, 2)
plt.hist(list(list(pre_trained_model.parameters())[8].flatten().data.cpu().numpy()),bins
plt.title('Layer 2 Output Histogram')
plt.subplot(5, 1, 3)
plt.hist(list(list(pre_trained_model.parameters())[8].flatten().data.cpu().numpy()),bins
plt.title('Layer 3 Output Histogram')
plt.subplot(5, 1, 4)
plt.hist(list(list(pre_trained_model.parameters())[8].flatten().data.cpu().numpy()),bins
plt.title('Layer 4 Output Histogram')
```

```
plt.subplot(5, 1, 5)
plt.hist(list(list(pre_trained_model.parameters())[8].flatten().data.cpu().numpy()),bins
plt.title('Layer 5 Output Histogram')
plt.show()
```




```
import os
import copy
import numpy as np
from PIL import Image
import matplotlib.cm as mpl_color_map
import torch
from torch.autograd import Variable
from torchvision import models

```

Image loaded into array

```
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg16 import decode_predictions
from keras.applications.vgg16 import VGG16
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
%matplotlib inline
# load the model
model = pre_trained_model
# load an image from file
image1 = load_img('cat.1.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image2 = img_to_array(image1)
# reshape data for the model
image3 = image2.reshape((1, image2.shape[0], image2.shape[1], image2.shape[2]))
# prepare the image for the VGG model
image4 = preprocess_input(image3)
```

Code for Loading model and using it.


```

import os
import numpy as np
import torch
from torch.optim import Adam
from torchvision import models
class CNNLayerVisualization():
    """
        Produces an image that minimizes the loss of a convolution
        operation for a specific layer and filter
    """
    def __init__(self, model, selected_layer, selected_filter):
        self.model = model
        self.model.eval()
        self.selected_layer = selected_layer
        self.selected_filter = selected_filter
        self.conv_output = 0
        # Create the folder to export images if not exists
```

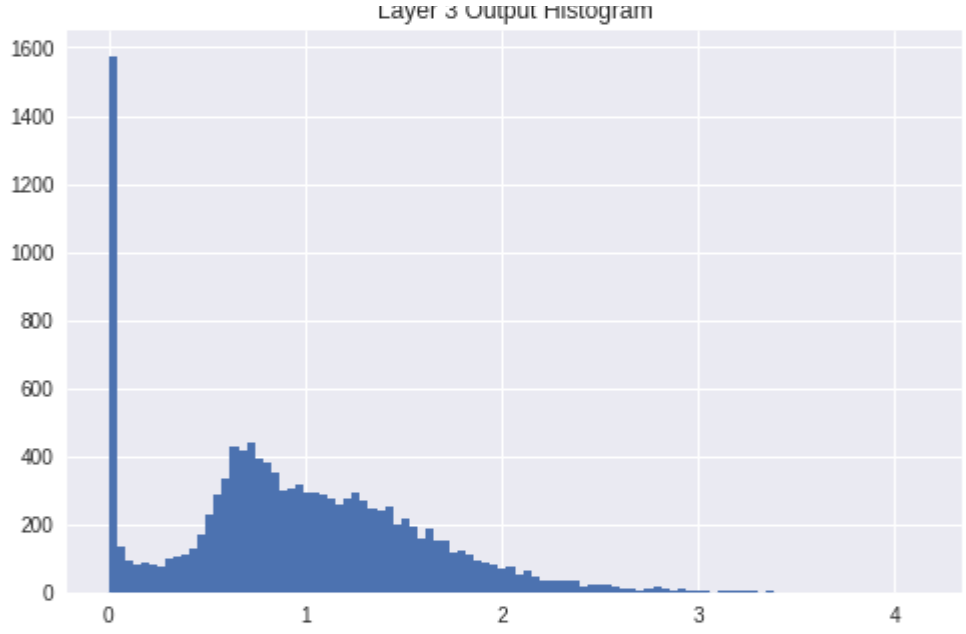
```
        if not os.path.exists('../generated'):
            os.makedirs('../generated')

    def hook_layer(self):
        def hook_function(module, grad_in, grad_out):
            # Gets the conv output of the selected filter (from selected layer)
            self.conv_output = grad_out[0, self.selected_filter]
        # Hook the selected layer
        self.model[self.selected_layer].register_forward_hook(hook_function)

    def visualise_layer_with_hooks(self):
        # Hook the selected layer
        p='inactive'
        self.hook_layer()
        output=[]
        # Generate a random image
        image_path="cat.11.jpg"
        optimizer = Adam([torch.tensor(np.absolute(image4))], lr=0.1, weight_decay=1e-6)
        for i in range(1, 31):
            optimizer.zero_grad()
            # Assign create image to a variable to move forward in the model
            x = preprocess_image(image1)
            for index, layer in enumerate(self.model):
                # Forward pass layer by layer
                # x is not used after this point because it is only needed to trigger
                # the forward hook function
                x = layer(x.type(torch.cuda.FloatTensor))
                # Only need to forward until the selected layer is reached
                self.conv_output = x[0, self.selected_filter]
                output.append(self.conv_output)

                if index == self.selected_layer:
                    # (forward hook function triggered)
                    p='active'
                    break

            # Loss function is the mean of the output of the selected layer/filter
            # We try to minimize the mean of the output of that specific filter
            loss = -torch.mean(self.conv_output)
            # Backward
            loss.backward()
            # Update image
            optimizer.step()
            if(p=='active'):
              return output
        return output
    def preprocess_image(pil_im, resize_im=True):
      mean = [0.485, 0.456, 0.406]
      std = [0.229, 0.224, 0.225]
      # Resize image
      if resize_im:
          pil_im.thumbnail((512, 512))
      im_as_arr = np.float32(pil_im)
      im_as_arr = im_as_arr.transpose(2, 0, 1)  # Convert array to D,W,H
      # Normalize the channels
      for channel, _ in enumerate(im_as_arr):
          im_as_arr[channel] /= 255
          im_as_arr[channel] -= mean[channel]
          im_as_arr[channel] /= std[channel]
      # Convert to float tensor
      im_as_ten = torch.from_numpy(im_as_arr).float()
      # Add one more channel to the beginning. Tensor shape = 1,3,224,224
      im_as_ten.unsqueeze_(0)
      # Convert to Pytorch variable
      im_as_var = Variable(im_as_ten, requires_grad=True)
      return im_as_var


if __name__ == '__main__':
```

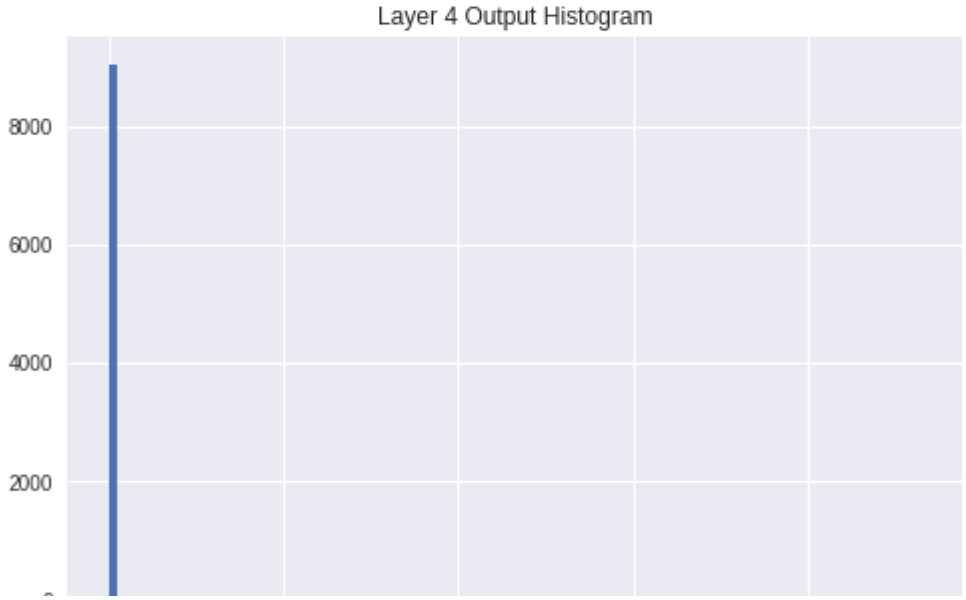
```
    cnn_layer = 16
    filter_pos = 5
    # Fully connected layer is not needed
    pretrained_model = pre_trained_model.features
    layer_vis = CNNLayerVisualization(pretrained_model, cnn_layer, filter_pos)
    output=layer_vis.visualise_layer_with_hooks()

```

Feature Maps are plotted below

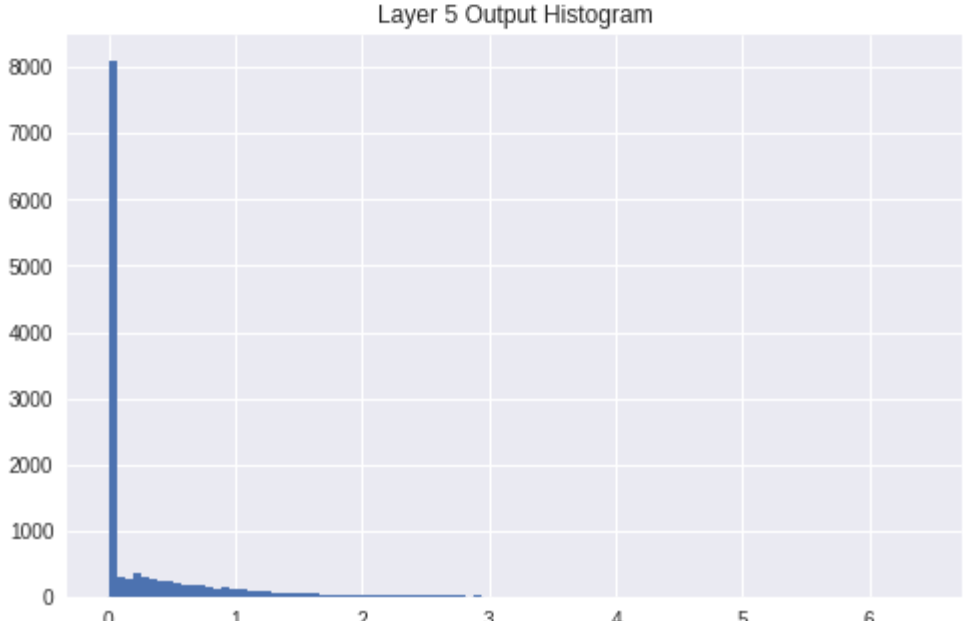
Note that Due to very less size of data (Just 20 images), some graphs might seem distorted from their normal and usual behaviour, but the code and logic part is indeed correct.

```
from matplotlib.pyplot import figure
import matplotlib.pyplot as plt
figure(num=None, figsize=(8, 30))
plt.style.use('seaborn-deep')
plt.subplot(5, 1, 1)
plt.hist(list(list(output[0].flatten().data.cpu().numpy())),bins=200)
plt.title('Layer 1 Output Histogram')
plt.subplot(5, 1, 2)
plt.hist(list(list(output[2].flatten().data.cpu().numpy())),bins=100)
plt.title('Layer 2 Output Histogram')
plt.subplot(5, 1, 3)
plt.hist(list(list(output[4].flatten().data.cpu().numpy())),bins=100)
plt.title('Layer 3 Output Histogram')
plt.subplot(5, 1, 4)
plt.hist(list(list(output[6].flatten().data.cpu().numpy())),bins=100)
plt.title('Layer 4 Output Histogram')
plt.subplot(5, 1, 5)
plt.hist(list(list(output[8].flatten().data.cpu().numpy())),bins=100)
plt.title('Layer 5 Output Histogram')
plt.show()
```


Layer 3 Output Histogram
1600
1400
1200
1000
800
600
400
200
0
0
1
2
3
4


Layer 4 Output Histogram
8000
6000
4000
2000

1


Layer 5 Output Histogram
8000
7000
6000
5000
4000
3000
2000
1000
0

0
1
2
3
4
5
6