Homework 2
CS 5787 Deep Learning
Spring 2020

Due: See Canvas
Name: Kushal Singh
NetID: ks2377
Worked with: Chris Shei, Gyan Suri

Instructions

Your homework submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. **If you do work with others, you must list the people you worked with.** Submit your solutions as a PDF to Canvas.

Your homework solution must be typed, and we suggest you do this using LaTeX. Homework must be output to PDF format. I suggest using `http://overleaf.com` to create your document. Overleaf is free and can be accessed online.

Your programs must be written in Python. The relevant code to the problem should be in the PDF you turn in. If a problem involves programming, then the code should be shown as part of the solution to that problem. One easy way to do this in LaTeX is to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}

If told to implement an algorithm, don't use a toolbox, or you will receive no credit. Do not post your code to a public web repository (e.g., GitHub).

Problem 1 - Using a Pre-Trained CNN

For this problem you must use PyTorch.

Part 1 - Using Pre-Trained Deep CNN (5 points)

For this problem you will use a CNN that has been trained on ImageNet-1k. Choose the pre-trained model of your choice (e.g., VGG-16, VGG-19, ResNet-18, ResNet-152, etc.). Run it on `peppers.jpg`. Output the top-3 predicted categories and the probabilities.

Make sure to list the deep CNN model you used. Make sure to pre-process the input image appropriately. Look at the toolbox documentation for the pre-trained model you use to determine how to do this.

Solution:
Using the VGG-16 pre-trained model on ImageNet-1k, we outputted the top-3 predicted categories and probabilities for the peppers image pictured below.

The results are displayed below and the code can be found in the appendix section.

```
Granny Smith 0.12514063715934753
cucumber, cuke 0.23650312423706055
bell pepper 99.32003784179688
```

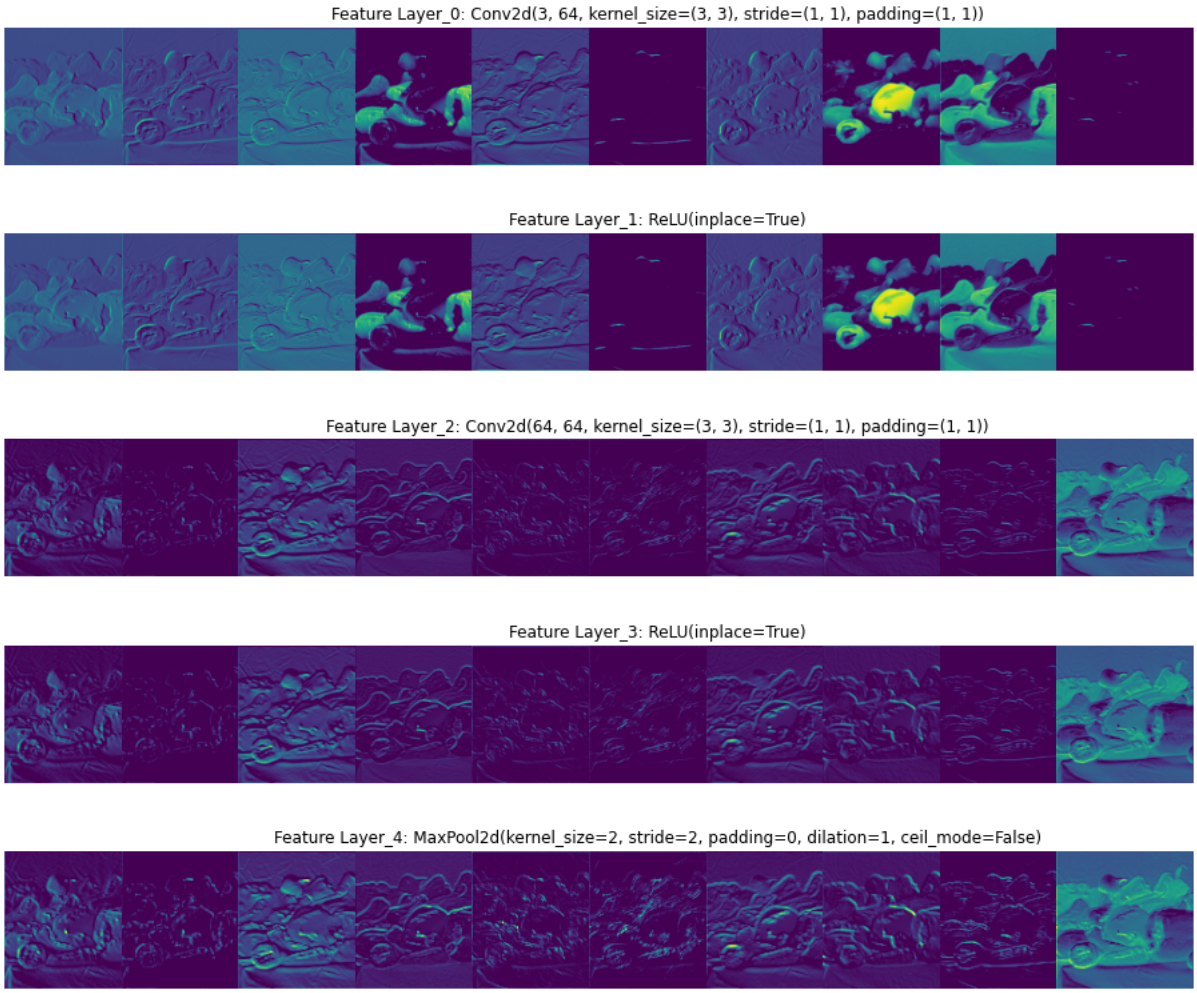
Part 2 - Visualizing Feature Maps (5 points)

Write code to visualize the feature maps in the network as images. You will likely need to normalize the values between 0 and 1 to do this.

Choose five interesting feature maps from early in the network, five from the the middle of the network, and and five close to the end of the network. Display them to us and discuss the structure of the feature maps. Try to find some that are interpretable, and discuss the challenges in doing so.

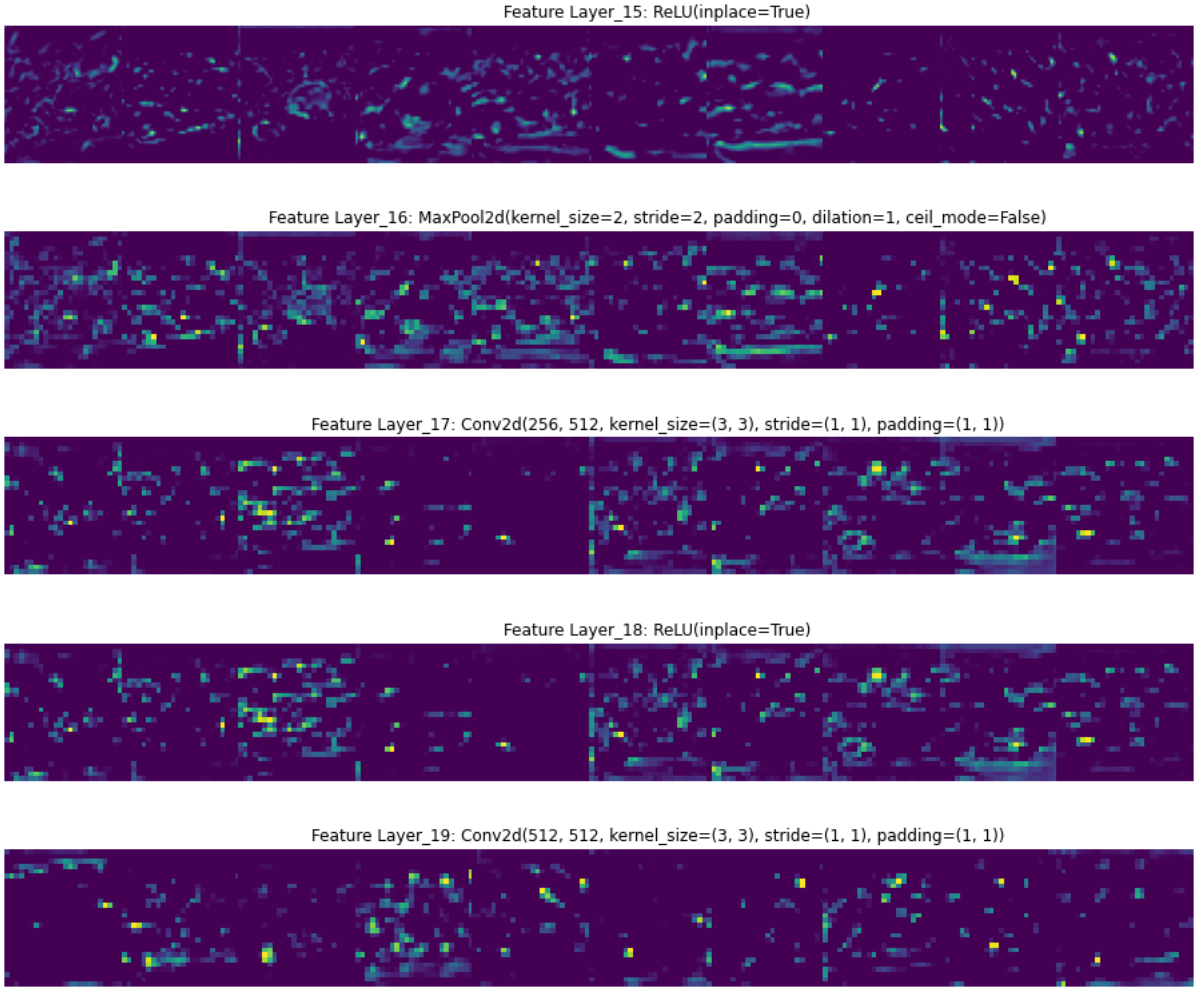
Solution:

5 interesting feature maps from early in the network



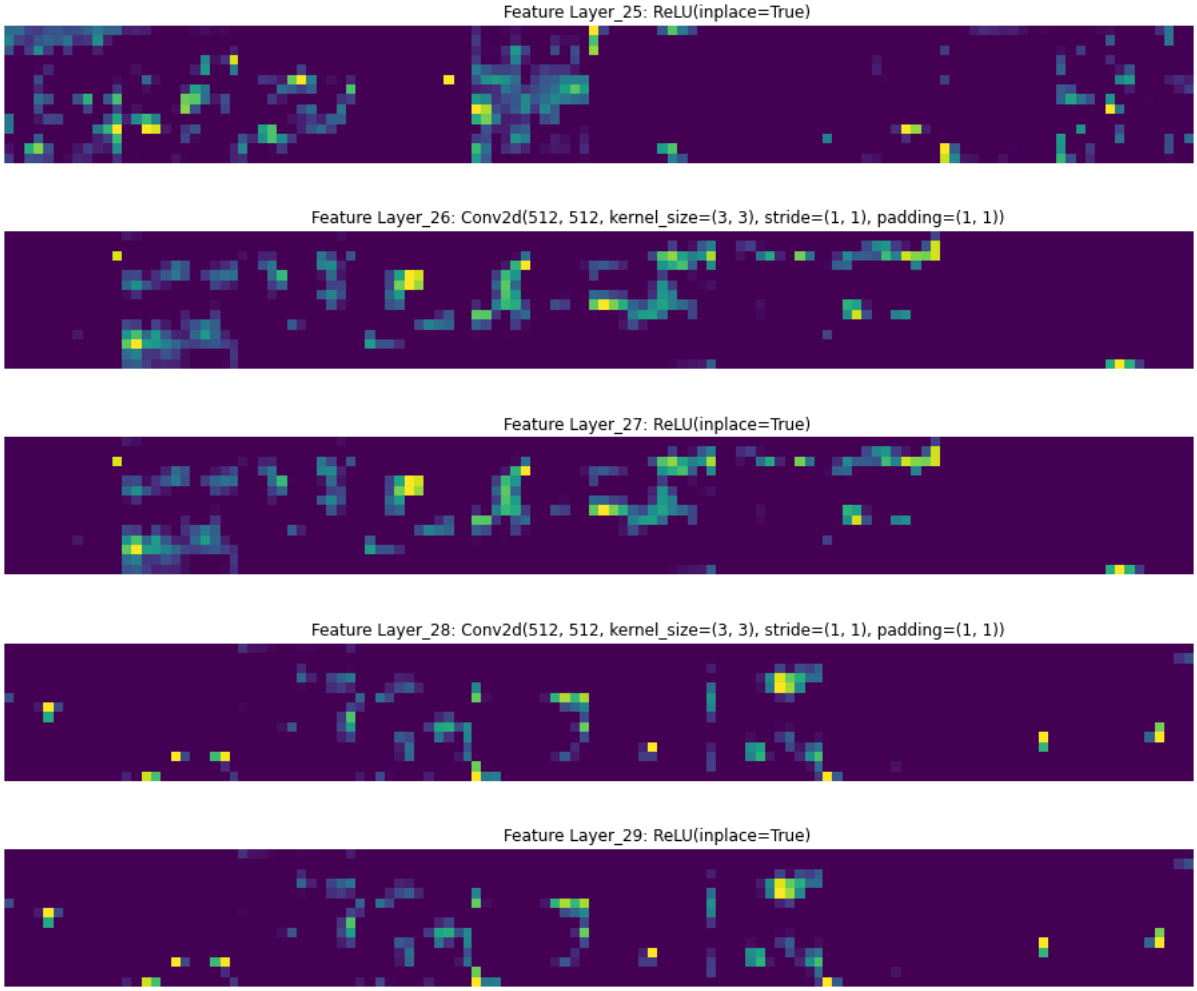
It's fairly straightforward to identify the structure of the feature maps from early on in the network. In the picture below, we highlight features from layers 0 through 4, inclusive. Some of the features that we're learning in this layer include color (images 8,9), shading/shadows (images 4,5), and edges (images 1,2,3).

5 interesting feature maps from middle of the network



It's slightly more nuanced and harder to identify the structure of the feature maps from the middle of the network. In the picture below, we highlight features from layers 15 through 19, inclusive. We can observe different parts of the image being highlighted, but it's a lot more difficult to determine what feature is causing the specific tag. Something that could help us make the determination clearer could be an analogous feature map to something we are familiar with (e.g. human face), which we would be able to use as a reference.

5 interesting feature maps from end of the network



It's more challenging and almost close to uninterpretable to identify the features from the end of the network. In the picture below, we highlight features from layers 25 through 29, inclusive. Because these features build upon abstract concepts from the previous filters, it's a lot harder to identify why certain sections are highlighted. For instance, we can only surmise that the highlighted portions are attempting to differentiate the peppers from the garlic.

Problem 2 - Transfer Learning with a Pre-Trained CNN (20 points)

For this problem you must use PyTorch. We will do image classification using the Oxford Pet Dataset. The dataset consists of 37 categories with about 200 images in each of them. You can find the dataset here: `http://www.robots.ox.ac.uk/~vgg/data/pets/`

Rather than using the final 'softmax' layer of the CNN as output to make predictions as we did in problem 1, instead we will use the CNN as a feature extractor to classify the Pets dataset. For each image, grab features from the last hidden layer of the neural network, which will be the layer **before** the 1000-dimensional output layer (around 2000–6000 dimensions). You will need to resize the images to a size compatible with your network (usually $224 \times 224 \times 3$). If your CNN model is using ReLUs, then you should grab the output just after the ReLU (so all values will be non-negative).

After you extract these features for all of the images in the dataset, normalize them to unit length by dividing by the L_2 norm. Train a linear classifier of your choice with the training CNN features, and then classify the test CNN features. Report mean-per-class accuracy and discuss the classifier you used.

Solution: From the graph attached on the next page, we can observe the architecture of the neural network. We grab the output from line (30), just after the ReLU activation, so that all values will be non-negative.

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
  )
)
```

After extracting these features for all of the images in the dataset (both training and test), we normalize them to unit length by dividing by the L_2 norm.

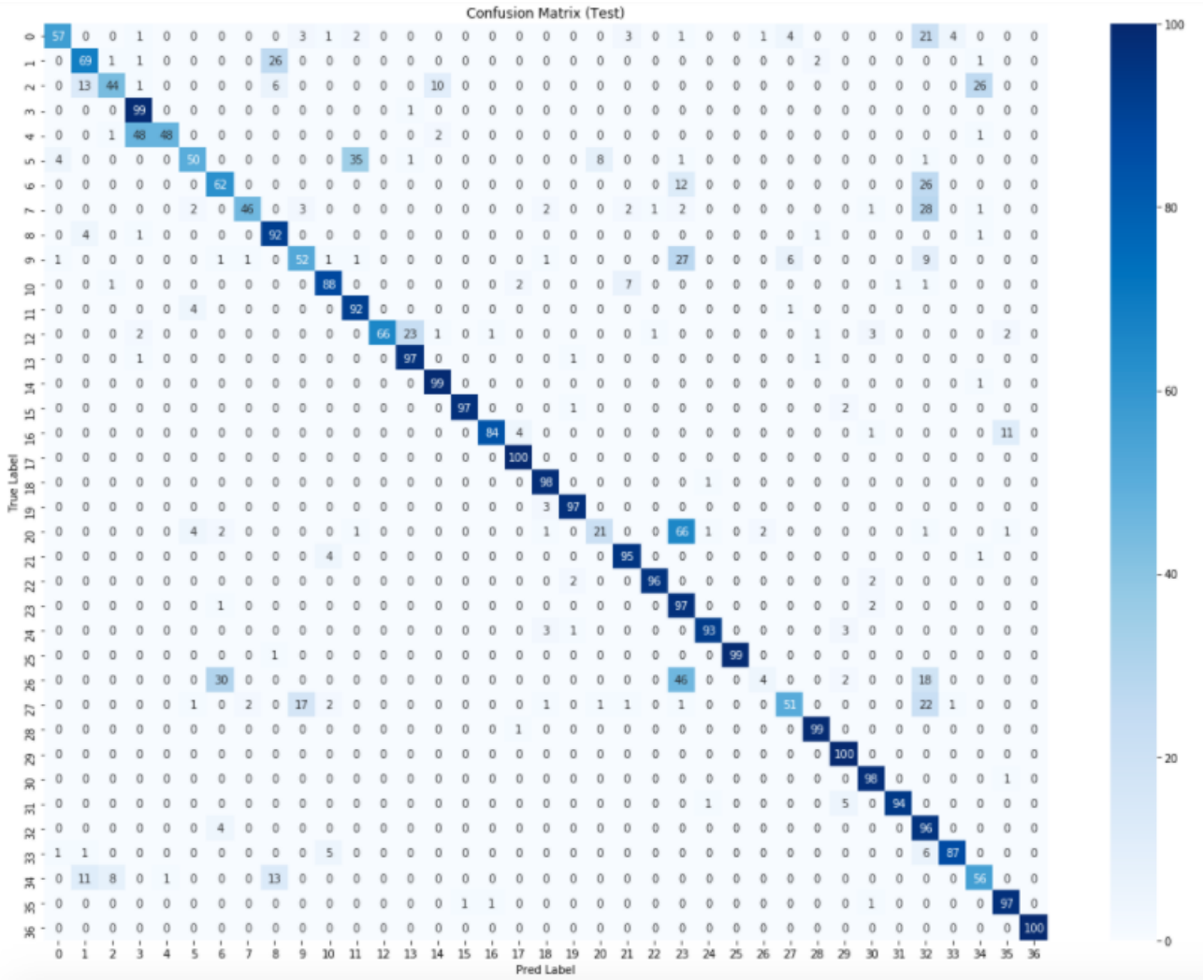
```
train_output = torch.stack(train_output)
train_labels = torch.stack(train_labels)
train_output_norm = np.linalg.norm(train_output)
train_norm = train_output / train_output_norm

test_output = torch.stack(test_output)
```

```
test_labels = torch.stack(test_labels)
test_output_norm = np.linalg.norm(test_output)
test_norm = test_output / test_output_norm
```

****The full code segment can be found in the appendix.****

I then used sklearn's linear support vector machine (SVM) classifier to train on the normalized CNN features, and subsequently tested on the normalized CNN features. This generated the following confusion matrix. We can see that the diagonals are fairly heavy, which means that, for the most part, the test label matched the predicted label.



We can see the precision-per-class for each of the different classes in the summary below. The overall accuracy was around 0.80, as evidenced by the recall column.

	precision	recall	f1-score	support
Abyssinian	0.90	0.58	0.71	98
american_bulldog	0.70	0.69	0.70	100
american_pit_bull_terrier	0.80	0.44	0.57	100
basset_hound	0.64	0.99	0.78	100
beagle	0.98	0.48	0.64	100
Bengal	0.82	0.50	0.62	100
Birman	0.62	0.62	0.62	100
Bombay	0.94	0.52	0.67	88
boxer	0.67	0.93	0.78	99
British_Shorthair	0.69	0.52	0.59	100
chihuahua	0.87	0.88	0.88	100
Egyptian_Mau	0.70	0.95	0.81	97
english_cocker_spaniel	1.00	0.66	0.80	100
english_setter	0.80	0.97	0.87	100
german_shorthaired	0.88	0.99	0.93	100
great_pyrenees	0.99	0.97	0.98	100
havanese	0.98	0.84	0.90	100
japanese_chin	0.93	1.00	0.97	100
keeshond	0.90	0.99	0.94	99
leonberger	0.95	0.97	0.96	100
Maine_Coon	0.70	0.21	0.32	100
miniature_pinscher	0.88	0.95	0.91	100
newfoundland	0.98	0.96	0.97	100
Persian	0.38	0.97	0.55	100
pomeranian	0.97	0.93	0.95	100
pug	1.00	0.99	0.99	100
Ragdoll	0.57	0.04	0.07	100
Russian_Blue	0.82	0.51	0.63	100
saint_bernard	0.95	0.99	0.97	100
samoyed	0.89	1.00	0.94	100
scottish_terrier	0.91	0.99	0.95	99
shiba_inu	0.99	0.94	0.96	100
Siamese	0.42	0.96	0.58	100
Sphynx	0.95	0.87	0.91	100
staffordshire_bull_terrier	0.64	0.63	0.63	89
wheaten_terrier	0.87	0.97	0.92	100
yorkshire_terrier	1.00	1.00	1.00	100
accuracy			0.80	3669
macro avg	0.83	0.79	0.78	3669
weighted avg	0.83	0.80	0.78	3669

SVMs tend to be less effective on noisier datasets with overlapping classes. In this example, although there is a high likelihood for overlap among the classes, since most of the images are of cats and dogs, the features that we train the SVM on are pretty well-defined, since we obtained them from vgg16, which offsets the overlap between classes, as evidenced by the accuracy score of 0.80.

Problem 3 - Training a Small CNN

Part 1 (30 points)

For this problem you must use a toolbox. Train a CNN with three hidden convolutional layers that use the ReLU activation function. Use 64 11×11 filters for the first layer, followed by 2×2 max pooling (stride of 2). The next two convolutional layers will use 128 3×3 filters followed by the ReLU activation function. Prior to the softmax layer, you should have an average pooling layer that pools across the preceding feature map. Do not use a pre-trained CNN.

Train your model using all of the CIFAR-10 training data, and evaluate your trained system on the CIFAR-10 test data.

Visualize all of the $11 \times 11 \times 3$ filters learned by the first convolutional layer as an RGB image array (I suggest making a large RGB image that is made up of each of the smaller images, so it will have 4 rows and 16 columns). This visualization of the filters should be similar to the ones we saw in class. Note that you will need to normalize each filter by contrast stretching to do this visualization, i.e., for each filter subtract the smallest value and then divide by the new largest value.

Display the training loss as a function of epochs. What is the accuracy on the test data? How did you initialize the weights? Discuss your architecture and hyper-parameters.

Solution: Images and plots attached below. Code can be found in the appendix section titled **Problem 3, Part 1**.

WEIGHT INITIALIZATION INFORMATION

We initialized the weights as a uniform distribution scaled by $\frac{1}{sqrtnumberofparameters}$.

*Code implementation courtesy of: *https://discuss.pytorch.org/t/what-is-the-default-initialization-of-a-conv2d-layer-and-linear-layer/16055/5*

```
def reset_parameters(self):
    n = self.in_channels
    for k in self.kernel_size:
        n *= k
    stdv = 1. / math.sqrt(n)
    self.weight.data.uniform_(-stdv, stdv)
    if self.bias is not None:
        self.bias.data.uniform_(-stdv, stdv)
```

DESCRIBE HYPER-PARAMETERS

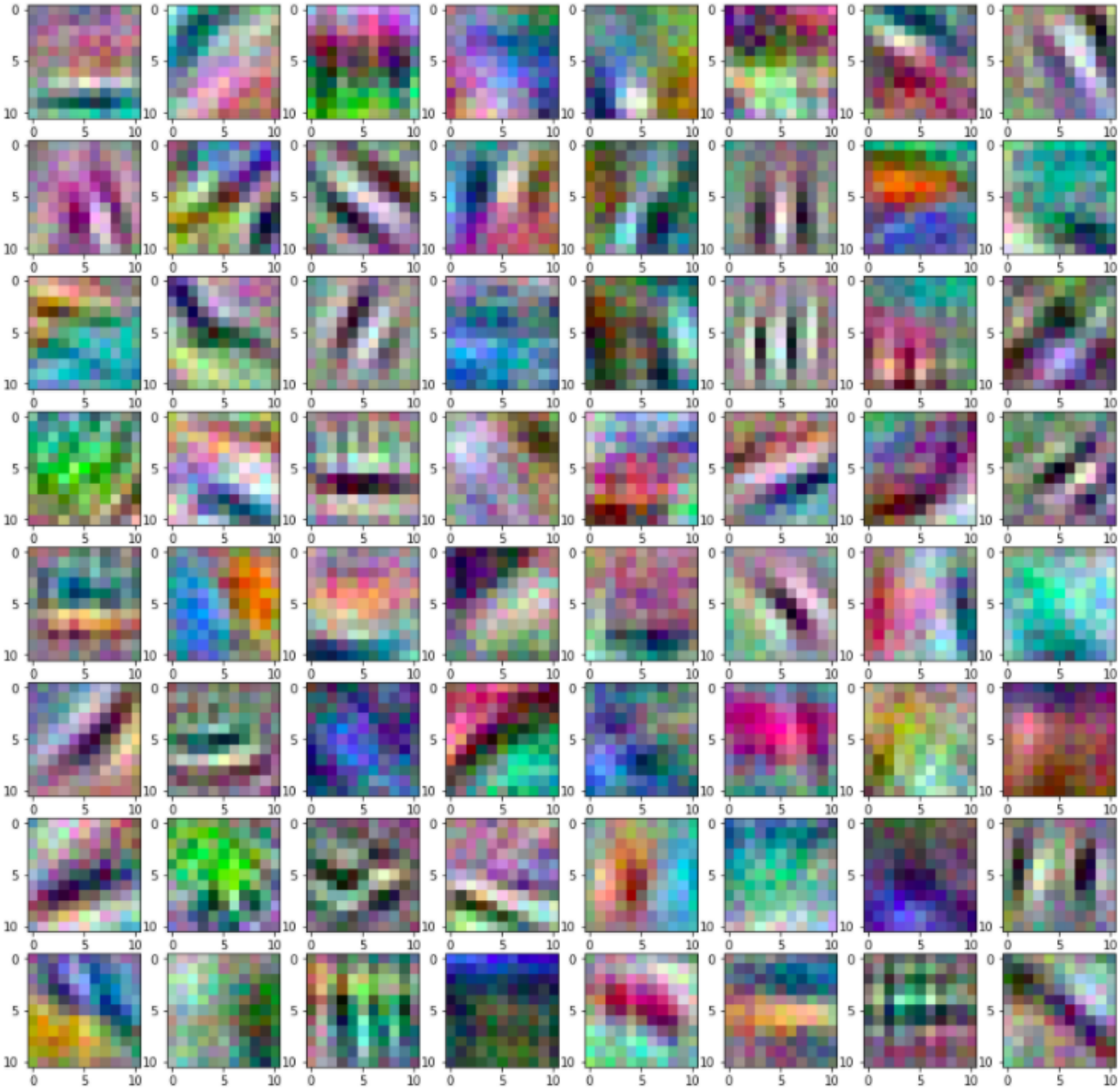
The hyperparameters used were:

* optimizer (using Adam)
* batch size
* learning rate = 0.01
* number of epochs = 10

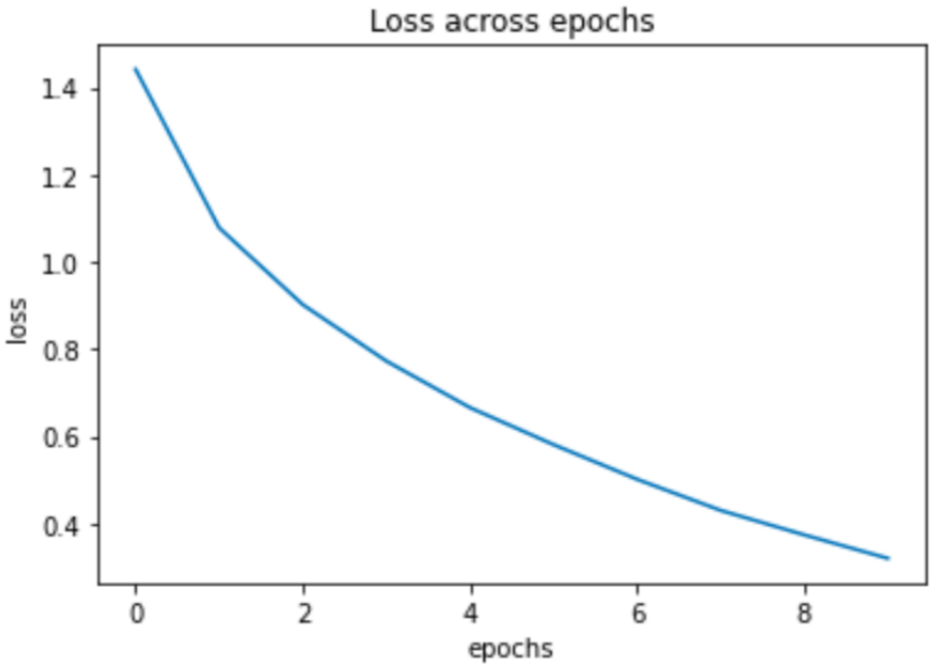
The network architecture comprised the following sequence:

* 3 layer model (first was a 64 11 x 11 filter, followed by a 2 x 2 max pooling with a stride of 2)
* 2 layers using 128 3 x 3 filters
* ReLU activation function
* Average pooling layer
* Softmax layer

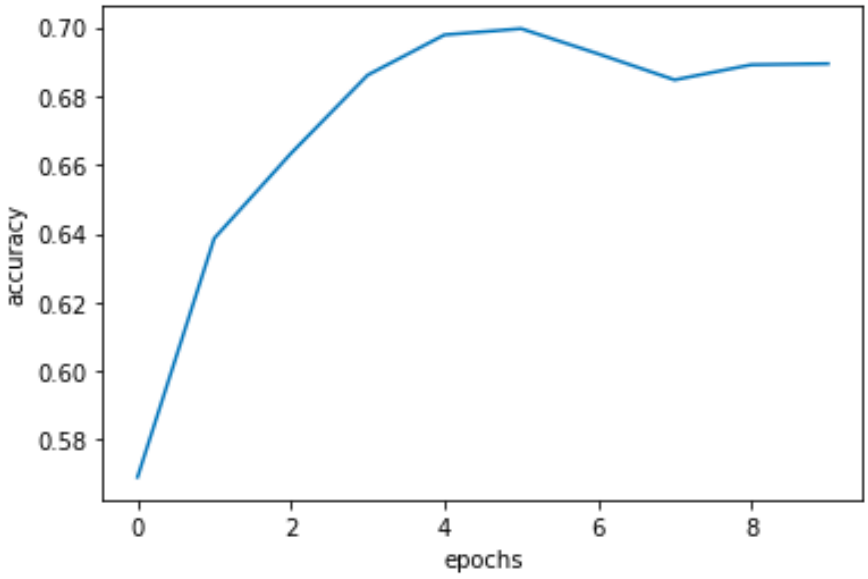
IMAGE SHOWING THE FILTERS



TRAINING LOSS AS FUNCTION OF EPOCHS



TEST DATA ACCURACY = 69%



TRAINING AND FINAL ACCURACY

```
[1,   250] loss: 1.756
[1,   500] loss: 1.509
[1,   750] loss: 1.380
[1,  1000] loss: 1.306
[1,  1250] loss: 1.257
Accuracy of the network on the 10000 test images: 57 %
[2,   250] loss: 1.144
[2,   500] loss: 1.116
[2,   750] loss: 1.071
[2,  1000] loss: 1.048
[2,  1250] loss: 1.011
Accuracy of the network on the 10000 test images: 64 %
[3,   250] loss: 0.894
[3,   500] loss: 0.918
[3,   750] loss: 0.910
[3,  1000] loss: 0.909
[3,  1250] loss: 0.878
Accuracy of the network on the 10000 test images: 67 %
[4,   250] loss: 0.751
[4,   500] loss: 0.771
[4,   750] loss: 0.783
[4,  1000] loss: 0.771
[4,  1250] loss: 0.786
Accuracy of the network on the 10000 test images: 67 %
[5,   250] loss: 0.618
[5,   500] loss: 0.653
[5,   750] loss: 0.675
[5,  1000] loss: 0.675
[5,  1250] loss: 0.707
Accuracy of the network on the 10000 test images: 69 %
[6,   250] loss: 0.511
[6,   500] loss: 0.575
[6,   750] loss: 0.585
[6,  1000] loss: 0.620
[6,  1250] loss: 0.612
Accuracy of the network on the 10000 test images: 70 %
[7,   250] loss: 0.441
[7,   500] loss: 0.485
[7,   750] loss: 0.515
[7,  1000] loss: 0.507
[7,  1250] loss: 0.557
Accuracy of the network on the 10000 test images: 69 %
[8,   250] loss: 0.361
[8,   500] loss: 0.409
[8,   750] loss: 0.434
[8,  1000] loss: 0.460
[8,  1250] loss: 0.484
Accuracy of the network on the 10000 test images: 68 %
[9,   250] loss: 0.319
[9,   500] loss: 0.355
[9,   750] loss: 0.380
[9,  1000] loss: 0.398
[9,  1250] loss: 0.416
Accuracy of the network on the 10000 test images: 70 %
[10,   250] loss: 0.250
[10,   500] loss: 0.296
[10,   750] loss: 0.332
[10,  1000] loss: 0.343
[10,  1250] loss: 0.379
Accuracy of the network on the 10000 test images: 69 %
```

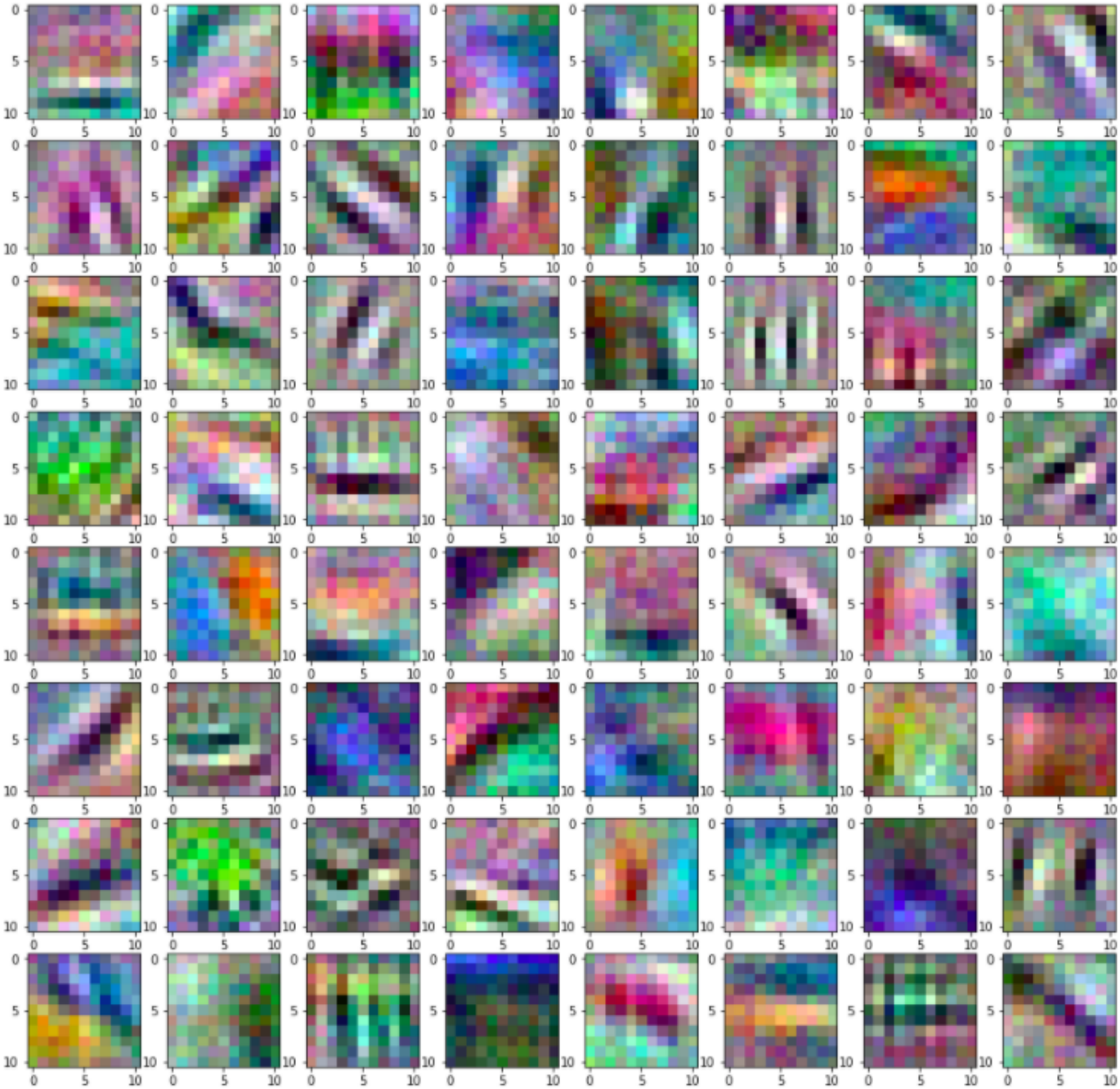
```
Accuracy of plane : 70 %
Accuracy of   car : 79 %
Accuracy of  bird : 54 %
Accuracy of   cat : 49 %
Accuracy of  deer : 66 %
Accuracy of   dog : 57 %
Accuracy of  frog : 79 %
Accuracy of horse : 66 %
Accuracy of  ship : 87 %
Accuracy of truck : 78 %
Accuracy of the network:  0.6894
```

Part 2 (20 points)

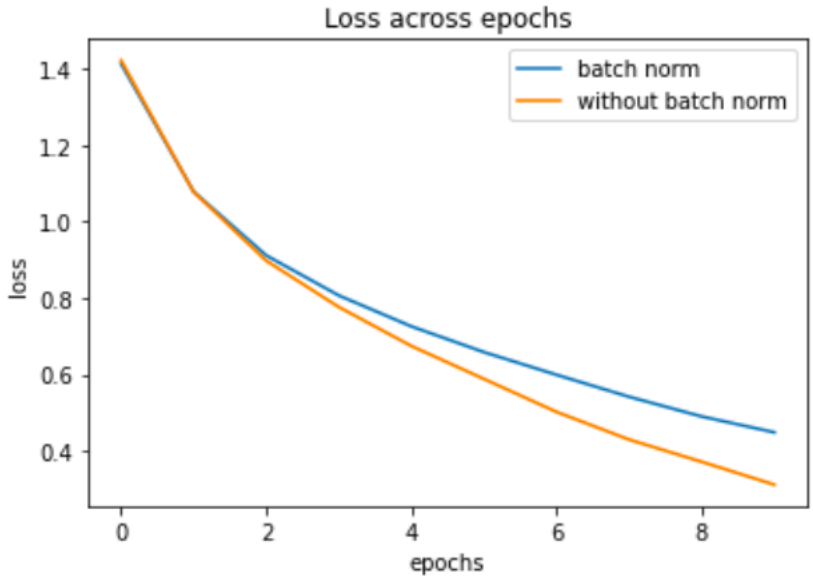
Using the same architecture as in part 1, add in batch normalization between each of the hidden layers. Compare the training loss with and without batch normalization as a function of epochs. What is the final test accuracy? Visualize the filters.

Solution: Images and plots attached below. Code can be found in the appendix section titled **Problem 3, Part 2**.

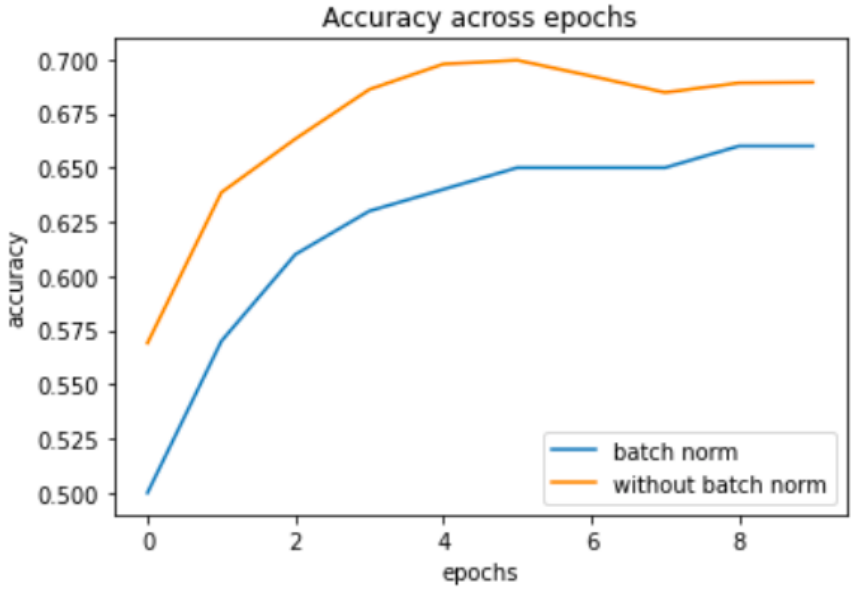
IMAGE SHOWING THE FILTERS



TRAINING LOSS WITH AND WITHOUT BATCH NORM



TEST DATA ACCURACY = 66%



Part 3 (10 points)

Can you do better with a deeper and better network architecture? Optimize your CNN's architecture to improve performance. You may get significantly better results by using smaller filters for the first convolutional layer. Describe your model's architecture and your design choices. What is your final accuracy?

Note: Your model should perform better than the one in Part 1 and Part 2.

Solution: The network architecture comprised the following sequence:

```
*   5 layer model (first was a 32 3 x 3 filter, followed by a 2 x 2 max pooling
    with a stride of 2)
*   2 layers using 32 3 x 3 filters (followed by a 2 x 2 max pooling
    with a stride of 2)
*   1 layer using 64 3 x 3 filters
*   1 layer using 128 3 x 3 filters
*   ReLU activation function
*   Average pooling layer
*   Softmax layer
```

Code can be found in the appendix section titled ***Problem 3, Part 3****.*

To optimize our network architecture, we used a 5 layer CNN with 3×3 filters for each layer, max pooling after the 1st and 3rd layers, and average pooling for the last layer before the softmax. This yielded in a final accuracy of 75%.

In terms of design choices, we decided to use more layers with smaller filters (3x3), akin to vgg16, which seemed to yield good results in the other problems. We removed batch normalization since it did not augment our results.

Problem 4 - Fooling Convolutional Neural Networks

In this problem you will fool the pre-trained convolutional neural network of your choice. One of the simplest ways to do this is to add a small amount of adversarial noise to the input image, which causes the correct predicted label y_{true} to switch to an incorrect adversarial label y_{fool}, despite the image looking the same to our human visual system.

Part 1 (20 points)

More formally, given an input image $\mathbf{X}$, an ImageNet pre-trained network will give us $P(y|\mathbf{X})$, which is a probability distribution over labels and the predicted label can be computed using the argmax function. We assume the network has been trained to correctly

classify $\mathbf{X}$. To create an adversarial input, we want to find $\hat{\mathbf{X}}$ such that $\hat{\mathbf{X}}$ will be misclassified as y_{fool}. To ensure that $\hat{\mathbf{X}}$ does not look radically different from $\mathbf{X}$ we impose a constraint on the distance between the original and modified images, i.e., $\left\|\mathbf{X} - \hat{\mathbf{X}}\right\|_\infty \leqslant \epsilon$, where ϵ is a small positive number. This model can be trained using backpropagation to find the adversarial example, i.e.,

$$\hat{\mathbf{X}} = \arg\min_{\mathbf{X}'} \left(Loss\left(\mathbf{X}', y_{fool}\right) + \frac{\lambda}{2} \left\|\mathbf{X}' - \mathbf{X}\right\|_\infty \right),$$

where $\lambda > 0$ is a hyperparameter and $\|\cdot\|_\infty$ denotes the infinity norm for tensors.

To do this optimization, you can begin by initializing $\mathbf{X}' \leftarrow \mathbf{X}$. Then, repeat the following two steps until you are satisfied with the results (or convergence):

$$\mathbf{X}' \leftarrow \mathbf{X}' + \lambda \frac{\partial}{\partial \mathbf{X}'} P\left(y_{fool} | \mathbf{X}'\right)$$
$$\mathbf{X}' \leftarrow \text{clip}\left(\mathbf{X}', \mathbf{X} - \epsilon, \mathbf{X} + \epsilon\right)$$

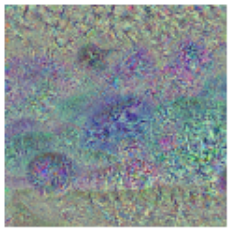
where the `clip` function 'clips' the values so that each pixel is within ϵ of the original image. You may use the neural network toolbox of your choice to do this optimization, but we will only provide help for PyTorch. You can read more about this approach here: `https://arxiv.org/pdf/1707.07397.pdf`. Note that the details are slightly different.

Demonstrate your method on four images. The first image should be 'peppers,' which was used in an earlier assignment. Show that you can make the network classify it as a space shuttle (ImageNet class id 812). You can choose the other three photos, but ensure that they contain an ImageNet object and make the network classify it as a different class. Ensure that the pre-trained CNN that you use outputs the correct class as the most likely assignment and give its probability. Then, show the 'noise image' that will be added to the original image. Then, show the noise+original image along with the new most likely class and the new largest probability. The noise+original image should be perceptually indistinguishable from the original image (to your human visual system). You may use the ImageNet pre-trained CNN of your choice (e.g., VGG-16, ResNet-50, etc.), but mention the pre-trained model that you used. You can show your results as a 4×3 array of figures, with each row containing original image (titled with most likely class and probability), the adversarial noise, and then the new image (titled with most likely class and probability).

Solution: Images and plots attached on the next page. Code can be found in the appendix section titled **Problem 4, Part 1**. ResNet-18 was the pretrained model that was used.

Real image: bell pepper with 99%

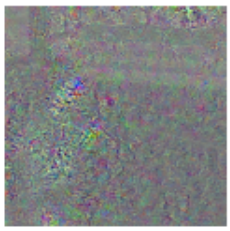
Noise



Fake Image: space shuttle with 99%

Real image: basketball with 93%

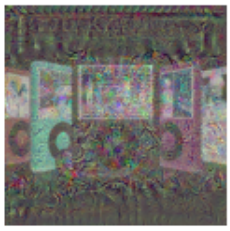
Noise



Fake Image: baseball with 98%

Real image: iPod with 99%

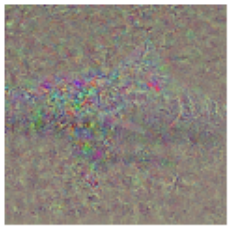
Noise



Fake Image: diaper, nappy, napkin with 97%

Real image: tiger shark, Galeocerdo cuvieri with 99%

Noise



Fake Image: sea anemone, anemone with 96%

Part 2 (10 points)

The method we deployed to make adversarial examples is not robust to all kinds of transformations. To examine the robustness of this, take the four adversarial images you created in part 1 and show how the following image manipulations affect the predicted class probabilities: mirror reflections (flip the image), a crop that contains about 80% of the original object, a 30 degree rotation, and converting the image to grayscale and then replicating the three gray channels. Show the modified adversarial images and title them with the new most likely class and probabilities. Discuss the results and why you think it did or did not work in each case.

Solution: Code can be found in the appendix section titled **Problem 4, Part 1**. The 4 $\times$ 4 array of figures is attached on the following page.

Analysis of Peppers Image
Because the peppers image wasn't too symmetrical, the attack was weak to a mirrored flip, as the model was still able to predict pepper with 99% probability. The areas that were attacked ended up being re-positioned, so the model could still glean that the image was a bell pepper.

Interestingly enough, the grayscale attack made the model predict that the image was that of a grocery store or supermarket. One could assume that because color is an important feature for determining the image as a bell pepper, by removing color from the image, the network has to pick up on other features, such as placement of items and shape/outline/structure, which made it believe that the image was a supermarket.

Analysis of Basketball Image
Because the basketball image is fairly symmetrical, the image flip attack was still able to predict basketball with 90% probability. The attack that was most successful was the 30 degree rotation, which probably goes to show that the orientation of the floor and the players plays an important role in determining the image.

The grayscale attack still made the model predict that the image was a basketball. One could assume that the model didn't really consider color to be as important of a feature for determining the image as a basketball, so by grayscaling the image, the network still has other features to work from, such as the basketball court, the team players, and the bench on the side.

Analysis of iPod Image
Because the iPod image is almost exactly symmetrical, the model was resistant to the image flip attack, and was still able to predict iPod with 99% probability. The grayscale attack still made the model predict that the image was an iPod, which means color wasn't a huge feature for determining iPods, which is fair, since iPods have a distinct shape,

which is probably a more important feature.

The attack that was most successful was the 30 degree rotation, which probably goes to show that most of the iPod images in the training set had an upright orientation, making it more difficult to identify rotated iPod images.

Analysis of Tiger Shark Image

In general, none of the attacks were successful on the tiger shark image. This goes to show that color, orientation, and rotation of image aren't significant features in identifying the image. The tiger shark has more distinctive features, such as fins, teeth, aquatic, and tails that are probably being learned by the model and are not reflected in any of the attacks.

Image Flip: bell pepper with 99%

80% Crop: bell pepper with 98%

30° Rotation: bell pepper with 83%

Grayscaled: grocery store, grocery, food market, market with 12%

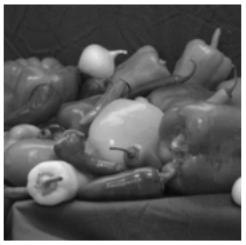


Image Flip: basketball with 90%

80% Crop: basketball with 96%

30° Rotation: bearskin, busby, shako with 9%

Grayscaled: basketball with 54%

Image Flip: iPod with 99%

80% Crop: iPod with 92%

30° Rotation: iPod with 36%

Grayscaled: iPod with 97%

Image Flip: tiger shark, Galeocerdo cuvieri with 99%

80% Crop: tiger shark, Galeocerdo cuvieri with 99%

30° Rotation: tiger shark, Galeocerdo cuvieri with 98%

Grayscaled: tiger shark, Galeocerdo cuvieri with 99%

Code Appendix

Code for Problem 1, Part 1

```
import torch
import pretrainedmodels
import pretrainedmodels.utils as utils

model_name = 'vgg16'
model = pretrainedmodels.__dict__[model_name](num_classes=1000, pretrained='imagenet')
model.eval()

load_img = utils.LoadImage()
tf_img = utils.TransformImage(model)
path_img = 'peppers.jpg'
input_img = load_img(path_img)
input_tensor = tf_img(input_img)
input_tensor = input_tensor.unsqueeze(0)
input = torch.autograd.Variable(input_tensor, requires_grad=False)
output_logits = model(input)

with open('imagenet_synsets.txt', 'r') as f:
  synsets = f.readlines()

synsets = [x.strip() for x in synsets]
splits = [line.split(' ') for line in synsets]
key_to_classname = {spl[0]:' '.join(spl[1:]) for spl in splits}

with open('imagenet_classes.txt', 'r') as f:
  class_id_to_key = f.readlines()
  labels = [line.strip() for line in f.readlines()]

class_id_to_key = [x.strip() for x in class_id_to_key]
output = model(input)
predictions = torch.argsort(output_logits)[0,-3:]
percentage = torch.nn.functional.softmax(output_logits, dim=1)[0] * 100

for i in range(0, 3):
  top_i = output_logits.data.squeeze().sort(0, True)[0][i].item()
  top_i_arg = (output_logits.data.squeeze() == top_i).nonzero().reshape(-1)[0]
  class_id = top_i_arg
```

```
  class_key = class_id_to_key[class_id]
  classname = key_to_classname[class_key]
  print(classname, percentage[predictions[i]].item())
```

Code for Problem 1, Part 2

```
from PIL import Image
from torchvision import transforms
import torchvision.models as models
import matplotlib.pyplot as plt

img = Image.open('peppers.jpg')
transform = transforms.Compose([
transforms.Resize(256),
transforms.CenterCrop(224),
transforms.ToTensor()])

img_transform = transform(img)
batch_transform = torch.unsqueeze(img_transform, 0)

vgg16 = models.vgg16(pretrained=True)
vgg16_bn = models.vgg16_bn(pretrained=True)

first_5_feats, mid_5_feats, last_5_feats, all_feats  = list(), list(),
list(), list()
first_5_layer_names, mid_5_layer_names, last_5_layer_names,
all_layer_names = list(), list(), list()

first_5_layers = [0, 1, 2, 3, 4]
mid_5_layers = [15, 16, 17, 18, 19]
last_5_layers = [25, 26, 27, 28, 29]

# conv_layers = [0, 1, 2, 3, 4, 15, 16, 17, 18, 19, 25, 26, 27, 28, 29]

def hook(module, input, output):
    all_feats.append(output)

for i in first_5_layers:
    feats = vgg16.features[i]
    feats.register_forward_hook(hook)
    first_5_layer_names.append(feats)
    print("First 5 Layer Features: ", feats)
```

```
for j in mid_5_layers:
    feats = vgg16.features[j]
    feats.register_forward_hook(hook)
    mid_5_layer_names.append(feats)
    print("Mid 5 Layer Features: ", feats)

for k in last_5_layers:
    feats = vgg16.features[k]
    feats.register_forward_hook(hook)
    last_5_layer_names.append(feats)
    print("Last 5 Layer Features: ", feats)

all_layer_names.extend(first_5_layer_names)
all_layer_names.extend(mid_5_layer_names)
all_layer_names.extend(last_5_layer_names)

num_layers = 5
num_map_imgs = 10
fig, ax = plt.subplots(num_layers, num_map_imgs)
for layer in range(num_layers):
    f = all_feats[layer][0][:num_map_imgs].detach()
    for i in range(num_map_imgs):
        ax[layer][i].imshow(f[i])
        ax[layer][int(num_map_imgs / 2)].set_title('Feature Layer_ {}' +
str(conv_layers[layer]) + ': ' + str(all_layer_names[layer]))
plt.show()
```

Code for Problem 2

```
import torch
import torchvision
import torchvision.models as models
import numpy as np
import matplotlib.pyplot as plt

from sklearn.svm import LinearSVC
from torchvision import transforms
from torch.utils.data import DataLoader

vgg16 = models.vgg16(pretrained=True)
for param in vgg16.parameters():
    param.requires_grad = False

transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(
    mean=[0.485, 0.456, 0.406],
    std=[0.229, 0.224, 0.225]
    )])

train_transform, test_transform = list(), list()
for img in range(len(ptrain_img)):
    get_train_img = Image.fromarray(ptrain_img[img])
    train_img_norm = transform(get_train_img)
    train_batch_norm = torch.unsqueeze(train_img_norm, 0)
    train_transform.append(train_batch_norm)

for img in range(len(ptest_img)):
    get_test_img = Image.fromarray(ptest_img[img])
    test_img_norm = transform(get_test_img)
    test_batch_norm = torch.unsqueeze(test_img_norm, 0)
    test_transform.append(test_batch_norm)

train_concat = torch.cat(train_transform, 0)
test_concat = torch.cat(test_transform, 0)
```

```
train_transform_set, test_transform_set = list(), list()
for train_item in range(len(train_concat)):
    get_train_transform = [train_concat[train_item], ptrain_y[train_item]]
    train_set.append(get_transform)

for test_item in range(len(test_concat)):
    get_test_transform = [test_concat[test_item], ptest_y[test_item]]
    test_set.append(get_test_transform)

train_data_loader = DataLoader(dataset = train_transform_set, batch_size=4)
test_data_loader = DataLoader(dataset = test_transform_set, batch_size=4)

train_output, train_labels = list(), list()
with torch.no_grad():
    for train_data in train_data_loader:
        images, labels = train_data
        train_labels.extend(labels)
        outputs = vgg16(images)
        train_output.extend(outputs)

train_output = torch.stack(train_output)
train_labels = torch.stack(train_labels)
train_output_norm = np.linalg.norm(train_output)
train_norm = train_output / train_output_norm

test_output, test_labels = list(), list()
with torch.no_grad():
    for test_data in test_loader:
        images, labels = test_data
        test_labels.extend(labels)
        outputs = vgg16(images)
        test_output.extend(outputs)

test_output = torch.stack(test_output)
test_labels = torch.stack(test_labels)
test_output_norm = np.linalg.norm(test_output)
test_norm = test_output / test_output_norm

model = LinearSVC()
model.fit(train_norm, train_labels)
```

Code for Problem 3, Part 1

Implementation courtesy of: https://www.stefanfiott.com/machine-learning/cifar-10-classifier-using-cnn-in-pytorch/

```
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=40,
shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
shuffle=False, num_workers=2)
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck')

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv_layer_1 = nn.Conv2d(3, 64, 11, padding=5)
        self.pool_layer_1 = nn.MaxPool2d(2, 2)
        self.conv_layer_2 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv_layer_3 = nn.Conv2d(128, 128, 3, padding=1)
        self.pool_layer_2 = nn.AvgPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 8 * 8, 10)

    def calc_accuracy():
      num_right, total = 0, 0

      with torch.no_grad():
          for data in testloader:
              img, lab = data
              _, predicted = torch.max(net(img).data, 1)
              total += lab.size(0)
              num = (predicted == lab).sum().item()
              num_right += num
      return num_right / total

    def forward(self, nxt):
        nxt = self.pool_layer_2(F.relu(self.conv_layer_3(F.relu(self.conv_layer_2(self.pool
```

```
        nxt = nxt.view(-1, 128 * 8 * 8)
        return self.fc1(nxt)

net = Net()
LEARNING_RATE = 0.001
EPOCHS = 10
BATCH_SIZE = 50000
NUM_TRAINING = 40
optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)

running_loss, running_acc = [], []
for _ in range(EPOCHS):
    epoch_loss = 0

    for _, data in enumerate(trainloader, 0):
        img, lab = data
        optimizer.zero_grad()
        outputs = net(img)
        loss = nn.CrossEntropyLoss()(outputs, lab)
        loss.backward()
        optimizer.step()
        e_loss += loss.item()

    running_loss.append(e_loss * NUM_TRAINING / BATCH_SIZE)
    running_acc.append(net.calc_accuracy())

plt.plot(running_loss)
plt.xlabel("epochs")
plt.ylabel("loss")
plt.title('Loss across epochs')
plt.show()

plt.plot(running_acc)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.show()

class_correct_classif, class_total_classif = [], []
for i in range(10):
  class_correct_classif[i] = 0
  class_total_classif[i] = 0
```

```
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        corr = (predicted == labels).squeeze()

        for i in range(4):
          label = labels[i]
          class_correct_classif[label] += corr[i].item()
          class_total_classif[label] += 1

layer_one = np.moveaxis(net.conv_layer_1.weight.detach().numpy(), 1, -1)
numer = (layer_one - layer_one.min())
denom = (layer_one.max() - layer_one.min())
layer_one = numer / denom

f, a = plt.subplots(8, 8, figsize=(20, 20))
for i in range(8):
    for j in range(8):
        norm_min = layer_one[i+8*j,:,:,:].min()
        norm_max = layer_one[i+8*j,:,:,:].max()
        numer = (layer_one[i+8*j,:,:,:] - norm_min)
        denom = (norm_max - norm_min)
        norm_layer = numer / denom
        a[i][j].imshow(norm_layer)
```

Code for Problem 3, Part 2

```
running_loss, running_acc = [], []
for _ in range(EPOCHS):
    epoch_loss, running_loss_num  = 0.0, 0.0
    for _, data in enumerate(trainloader, 0):
        img, lab = data
        optimizer.zero_grad()
        outputs = net(img)
        loss = nn.CrossEntropyLoss()(outputs, lab)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        e_loss += loss.item()
```

```
        if i % 250 == 249:
          print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss /
250))
          running_loss_num = = 0.0

    running_loss.append(e_loss * NUM_TRAINING / BATCH_SIZE)
    running_acc.append(net.calc_accuracy())

    class_correct_classif, class_total_classif = [], []
for i in range(10):
  class_correct_classif[i] = 0
  class_total_classif[i] = 0

with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        corr = (predicted == labels).squeeze()

        for i in range(4):
          label = labels[i]
          class_correct_classif[label] += corr[i].item()
          class_total_classif[label] += 1

layer_one = np.moveaxis(net.conv_layer_1.weight.detach().numpy(), 1, -1)
numer = (layer_one - layer_one.min())
denom = (layer_one.max() - layer_one.min())
layer_one = numer / denom

f, a = plt.subplots(8, 8, figsize=(20, 20))
for i in range(8):
    for j in range(8):
        norm_min = layer_one[i+8*j,:,:,:].min()
        norm_max = layer_one[i+8*j,:,:,:].max()
        numer = (layer_one[i+8*j,:,:,:] - norm_min)
        denom = (norm_max - norm_min)
        norm_layer = numer / denom
        a[i][j].imshow(norm_layer)
```

Code for Problem 3, Part 3

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv_layer_1 = nn.Conv2d(3, 32, 3, padding=1)
        self.pool_layer_1 = nn.MaxPool2d(2, 2)
        self.conv_layer_2 = nn.Conv2d(32, 32, 3, padding=1)
        self.conv_layer_3 = nn.Conv2d(32, 32, 3, padding=1)
        self.pool_layer_2 = nn.MaxPool2d(2, 2)
        self.conv_layer_4 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv_layer_5 = nn.Conv2d(64, 128, 3, padding=1)
        self.pool_layer_3 = nn.AvgPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 4 * 4, 10)

    def calc_accuracy():
      num_right, total = 0, 0

      with torch.no_grad():
          for data in testloader:
              img, lab = data
              _, predicted = torch.max(net(img).data, 1)
              total += lab.size(0)
              num = (predicted == lab).sum().item()
              num_right += num
      return num_right / total

    def forward(self, nxt):
        nxt = F.relu(self.conv_layer_2(self.pool_layer_1(F.relu(self.conv_layer_1(nxt)))))
        nxt = F.relu(self.conv_layer_4(self.pool_layer_2(F.relu(self.conv_layer_3(nxt)))))
        nxt = self.pool_layer_3(F.relu(self.conv_layer_5(nxt)))
        nxt = nxt.view(-1, 128 * 4 * 4)
        return self.fc1(nxt)

layer_one = np.moveaxis(net.conv_layer_1.weight.detach().numpy(), 1, -1)
numer = (layer_one - layer_one.min())
denom = (layer_one.max() - layer_one.min())
layer_one = numer / denom

f, a = plt.subplots(8, 8, figsize=(20, 20))
for i in range(8):
```

```
for j in range(8):
    norm_min = layer_one[i+8*j,:,:,:].min()
    norm_max = layer_one[i+8*j,:,:,:].max()
    numer = (layer_one[i+8*j,:,:,:] - norm_min)
    denom = (norm_max - norm_min)
    norm_layer = numer / denom
    a[i][j].imshow(norm_layer)
```

Code for Problem 4, Part 1

```
import torchvision.models as models
import torch
import numpy as np

from PIL import Image
from torchvision import transforms
from torch import nn
from matplotlib import pyplot as plt

resnet18 = models.resnet18(pretrained=True)
resnet18.eval()

with open('imagenet_classes.txt') as f:
  imagenet_classes = [line.strip() for line in f.readlines()]

with open('imagenet_synsets.txt') as f:
  imagenet_synsets = {line.strip().split()[0]: line.strip().split()[1:] for line in f.readl

def conv_image_to_batch_title(img):
  return torch.unsqueeze(transform(Image.open(img)), 0)

def conv_image_to_batch_no_title(img):
  return torch.unsqueeze(transform(img), 0)

def conv_batch_to_image(batch):
  return normalize(np.transpose(torch.squeeze(batch, 0).data.numpy(),(1,2,0)))

def arg_max_img(model, batch):
  out = model(batch)
  perc = torch.nn.functional.softmax(out, dim=1)[0] * 100
  ind = out[0].sort()[1][-3:][2]
  target = ""
  for item in imagenet_synsets.get(str(imagenet_classes[ind])):
    target = target + item + " "
  prob = int(perc[ind].item())
  return f"{target} with {prob}%"

def calc_noise(image, fake):
  noise = fake - image
```

```
  numer = noise - np.min(noise)
  denom = np.max(noise) - np.min(noise)
  return numer / denom

def normalize(fake):
  numer = fake - np.min(fake)
  denom = np.max(fake) - np.min(fake)
  return numer / denom

def neural_net(batch, index_t, model):
  target = ""
  for item in imagenet_synsets.get(str(imagenet_classes[index_t])):
    target += item + " "
  print("Target image class: {}, Target image name: {} ".format(imagenet_classes[index_t],

  first_batch = batch
  batch = torch.autograd.Variable(batch, requires_grad=True)

  for _ in range(EPOCHS):
      res = model(batch)
      loss = -res[0, index_t]
      model.zero_grad()
      loss.backward(retain_graph=True)
      batch.data = batch.data - (LEARNING_RATE * batch.grad.data)
      grad = batch.grad
      batch = torch.max(batch, first_batch - CONST_E)
      batch = torch.min(batch, first_batch + CONST_E)
      batch.grad = grad
      res = model(batch)
      percentage = nn.functional.softmax(res,dim=1).data.numpy()[0,index_t] * 100
      if percentage >= 95:
        break
  return batch

LEARNING_RATE = 0.1
CONST_E = 0.1
EPOCHS = 100

im1_batch = conv_image_to_batch_title("peppers.jpg")
im2_batch = conv_image_to_batch_title("basketball.jpg")
im3_batch = conv_image_to_batch_title("ipod.jpg")
```

```
im4_batch = conv_image_to_batch_title("tigershark.jpg")

im1 = conv_batch_to_image(im1_batch)
im2 = conv_batch_to_image(im2_batch)
im3 = conv_batch_to_image(im3_batch)
im4 = conv_batch_to_image(im4_batch)

fake1 = conv_batch_to_image(neural_net(im1_batch, 812, resnet18))
fake2 = conv_batch_to_image(neural_net(im2_batch, 429, resnet18))
fake3 = conv_batch_to_image(neural_net(im3_batch, 529, resnet18))
fake4 = conv_batch_to_image(neural_net(im4_batch, 108, resnet18))

noise1 = calc_noise(conv_batch_to_image(im1_batch), fake1)
noise2 = calc_noise(conv_batch_to_image(im2_batch), fake2)
noise3 = calc_noise(conv_batch_to_image(im3_batch), fake3)
noise4 = calc_noise(conv_batch_to_image(im4_batch), fake4)

real_title1 = arg_max_img(resnet18, im1_batch)
real_title2 = arg_max_img(resnet18, im2_batch)
real_title3 = arg_max_img(resnet18, im3_batch)
real_title4 = arg_max_img(resnet18, im4_batch)

fake_title1 = arg_max_img(resnet18, neural_net(im1_batch, 812, resnet18))
fake_title2 = arg_max_img(resnet18, neural_net(im2_batch, 429, resnet18))
fake_title3 = arg_max_img(resnet18, neural_net(im3_batch, 529, resnet18))
fake_title4 = arg_max_img(resnet18, neural_net(im4_batch, 108, resnet18))
```

Code for Problem 4, Part 2

```
flip = transforms.RandomHorizontalFlip(p=1)
crop = transforms.Compose([transforms.CenterCrop(int(224*0.8)),transforms.Resize(224)])
rotate = transforms.RandomRotation((30,30))
grayscale = transforms.Grayscale(num_output_channels=3)

final_fake_image = list()
final_fake_image.append(Image.fromarray(np.uint8(fake1*255), mode='RGB'))
final_fake_image.append(Image.fromarray(np.uint8(fake2*255), mode='RGB'))
final_fake_image.append(Image.fromarray(np.uint8(fake3*255), mode='RGB'))
final_fake_image.append(Image.fromarray(np.uint8(fake4*255), mode='RGB'))

fake_flip, fake_crop, fake_rotate, fake_grayscale = list(), list(), list(), list()
```

```
num_iter = len(range(final_fake_image))

for num_iter in final_fake_image:
  fake_flip.append(flip(num_iter))
  fake_crop.append(crop(num_iter))
  fake_rotate.append(rotate(num_iter))
  fake_grayscale.append(grayscale(num_iter))

flip_title, crop_title, rotate_title, graysc_title = list(), list(), list(), list()
for flip in fake_flip:
  flip_title.append(arg_max_img(resnet18, conv_image_to_batch_no_title(flip)))

for crop in fake_crop:
  crop_title.append(arg_max_img(resnet18, conv_image_to_batch_no_title(crop)))

for rotate in fake_rotate:
  rotate_title.append(arg_max_img(resnet18, conv_image_to_batch_no_title(rotate)))

for graysc in fake_grayscale:
  graysc_title.append(arg_max_img(resnet18, conv_image_to_batch_no_title(graysc)))
```