CS 5787 Deep Learning, Spring 2020
Homework 2
Ananya Shivaditya (aks298)

Due: See Canvas

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

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: Code: Appendix 1
Chosen pretrained model: VGG-16

Image to be classified:
Top 3 predicted classes and their probabilities:

```
['bell_pepper', 'banana', 'Granny_Smith']
[0.9511, 0.0192, 0.0082]
```

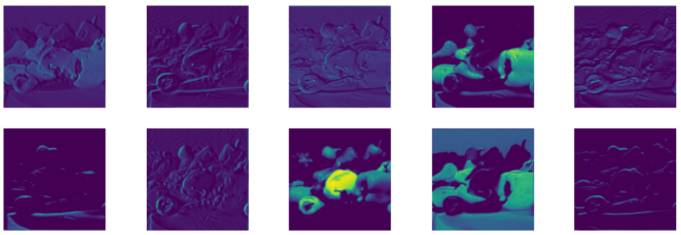
Part 2 - Visualizing Feature Maps (5 points)

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.

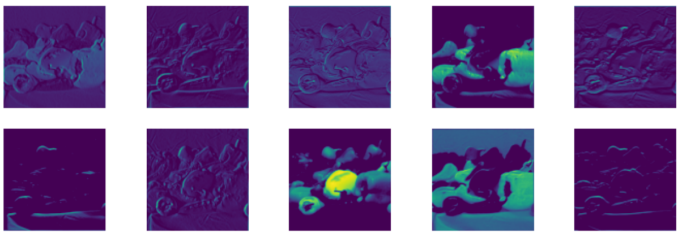
Solution: Code: Appendix 1
CNNs are less inscrutable than other NNs, can be made somewhat interpretable by the structure of feature maps from each layer. These are the activation of filters as an image is passed through the network.
We can understand some filters, like edges (layer 0, filter 2) and foreground/background (layer1, filers 4 and 9), as we see in the first few layers of the architecture. It becomes increasingly difficult to interpret successive layers feature maps in the network, much like nested function outputs. This is apparent from the later layers, as these activations represent higher level features, which may be hard to visualize.

Feature map visualization from 5 first layers

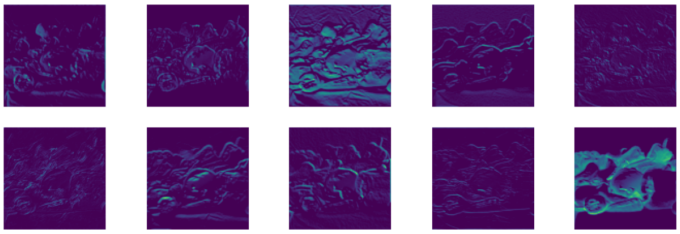
Layer 0:



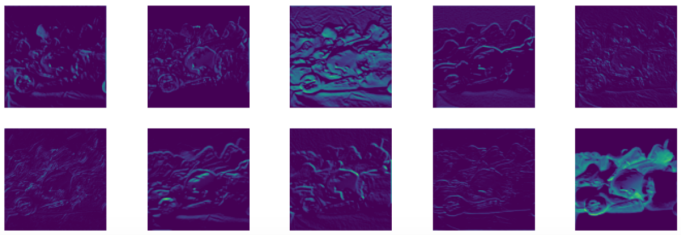
Layer 1:



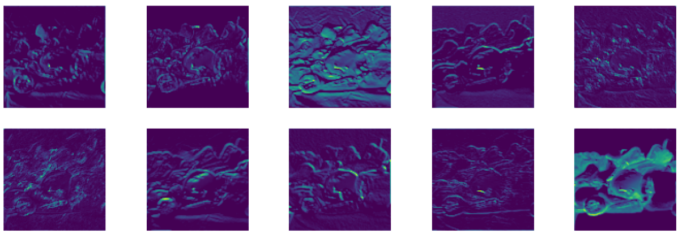
Layer 2:



Layer 3:

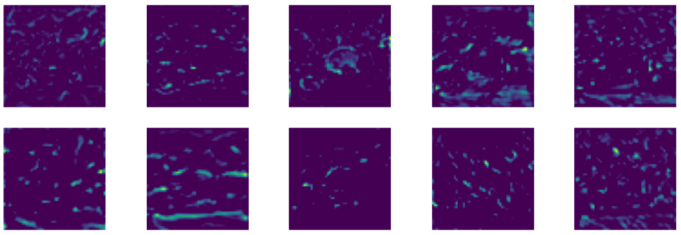


Layer 4:

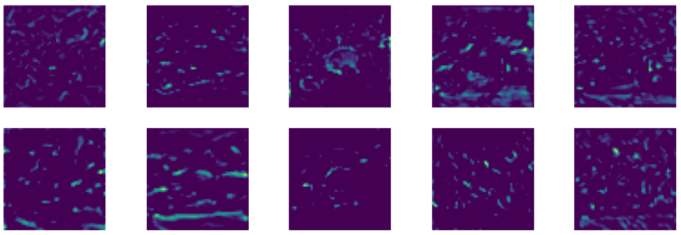


Feature map visualization from 5 mid layers

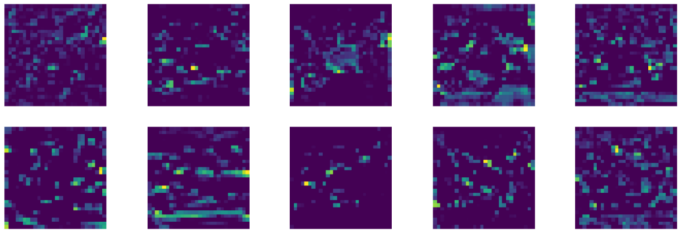
Layer 14:



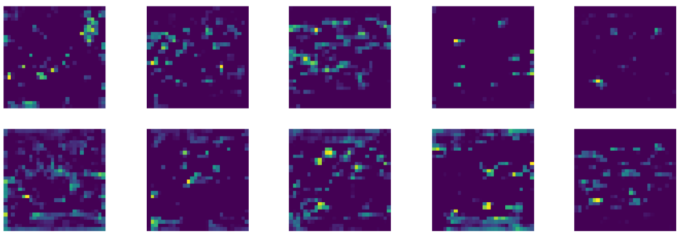
Layer 15:



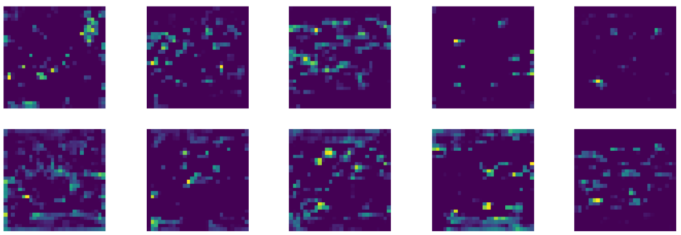
Layer 16:



Layer 17:

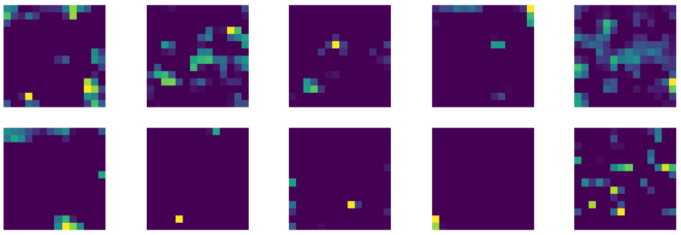


Layer 18:

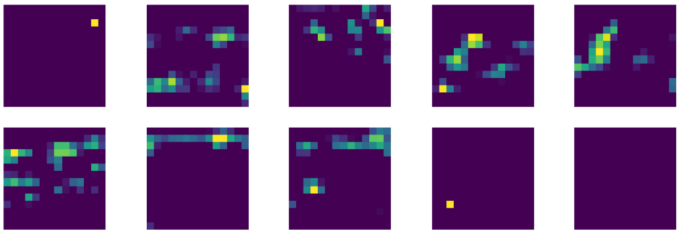


Feature map visualization from 5 end layers

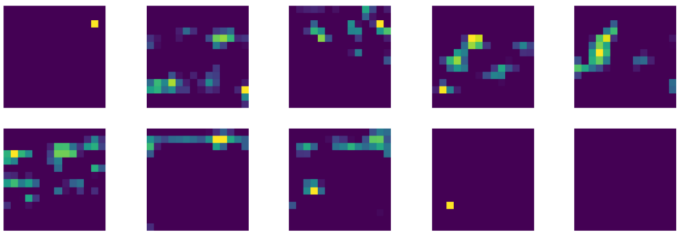
Layer 24:



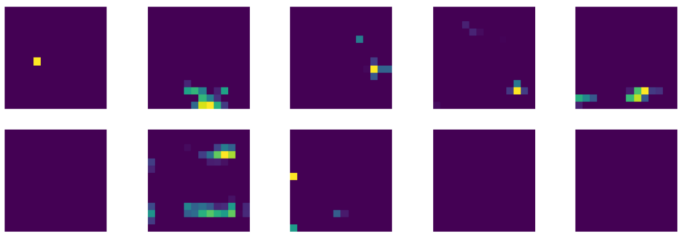
Layer 25:



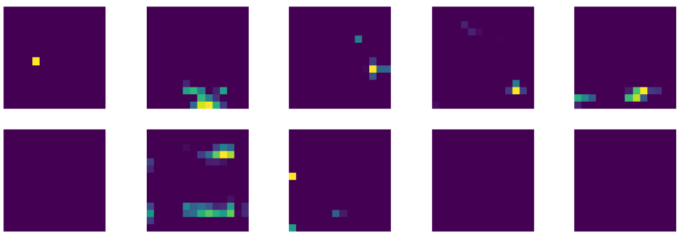
Layer 26:



Layer 27:



Layer 28:



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[1] with the training CNN features, and then classify the test CNN features. Report mean-per-class accuracy and discuss the classifier you used.

Solution:
Code: Appendix 2

Pretrained model of VGG-16 was used. All the parameters were frozen, and the last layer was removed.

The dataset was loaded from TensorFlow datasets. After normalization and resizing to be accepted by the network, each image in the dataset was passed through the network to obtain the 4096 dimensional features from the pretrained VGG-16 model.

These features were then converted to np array of floats to make the Xtrain, and each label was Y train. This was also followed for each image in the test set. For normalization, dividing by L2 norm proved less advantageous than diving by the max pixel value of 255. Perhaps this is because feature values are pushed towards 0 if they are smaller, therefore giving more importance to the large feature values.

A simple linear model was used for classification: Logistic regression from the scikit learn toolbox. This mimiced the last layer of the pretrained model in its softmax layer for classification. Of course, the number of output classes was 37 instead of the original 1000. Accuracy of the trained LR model was 0.42. Mean per class accuracies are shown below:

[1] You could use the softmax classifier you implemented for homework 1 or any toolbox you prefer.

	precision	recall	f1-score	support
Abyssinian	0.32	0.32	0.32	98
american_bulldog	0.34	0.42	0.38	100
american_pit_bull_terrier	0.20	0.13	0.16	100
basset_hound	0.27	0.27	0.27	100
beagle	0.29	0.36	0.32	100
Bengal	0.50	0.52	0.51	100
Birman	0.51	0.41	0.45	100
Bombay	0.53	0.51	0.52	88
boxer	0.25	0.24	0.25	99
British_Shorthair	0.36	0.35	0.36	100
chihuahua	0.34	0.27	0.30	100
Egyptian_Mau	0.65	0.59	0.62	97
english_cocker_spaniel	0.25	0.14	0.18	100
english_setter	0.42	0.48	0.45	100
german_shorthaired	0.32	0.42	0.37	100
great_pyrenees	0.38	0.43	0.40	100
havanese	0.42	0.54	0.47	100
japanese_chin	0.50	0.52	0.51	100
keeshond	0.66	0.60	0.62	99
leonberger	0.50	0.44	0.47	100
Maine_Coon	0.56	0.48	0.52	100
miniature_pinscher	0.53	0.36	0.43	100
newfoundland	0.50	0.58	0.54	100
Persian	0.48	0.43	0.45	100
pomeranian	0.44	0.45	0.44	100
pug	0.54	0.53	0.54	100
Ragdoll	0.36	0.45	0.40	100
Russian_Blue	0.35	0.30	0.32	100
saint_bernard	0.38	0.36	0.37	100
samoyed	0.50	0.60	0.55	100
scottish_terrier	0.49	0.44	0.47	99
shiba_inu	0.42	0.48	0.45	100
Siamese	0.48	0.42	0.45	100
Sphynx	0.42	0.48	0.45	100
staffordshire_bull_terrier	0.17	0.22	0.19	89
wheaten_terrier	0.39	0.41	0.40	100
yorkshire_terrier	0.65	0.57	0.61	100
accuracy			0.42	3669
macro avg	0.42	0.42	0.42	3669
weighted avg	0.42	0.42	0.42	3669

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.

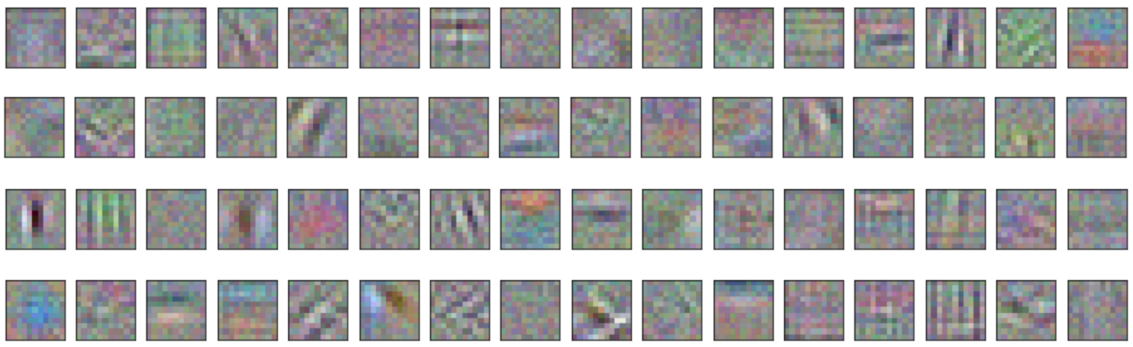
Visualize all of the $11 \times 11 \times 3$ filters learned by the first convolutional layer as an RGB image array (I suggest making a large RGB image that is made up of each of the smaller images, so it will have 4 rows and 16 columns). This visualization of the filters should be similar to the ones 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.

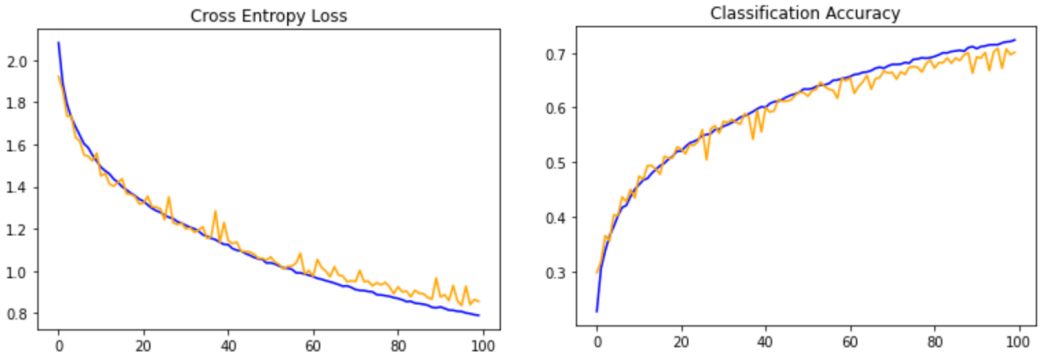
Solution:

The architecture was implemented as asked. It was trained on CIFAR-10 training data for 100 epochs, giving a test accuracy of 70.12 %. Code can be found at: **Appendix 3.1**

Image showing the filters:



Train (blue) and test (orange), loss (<-) and accuracy (->) as a function of epochs:



FINAL TEST DATA ACCURACY : 70.12 %

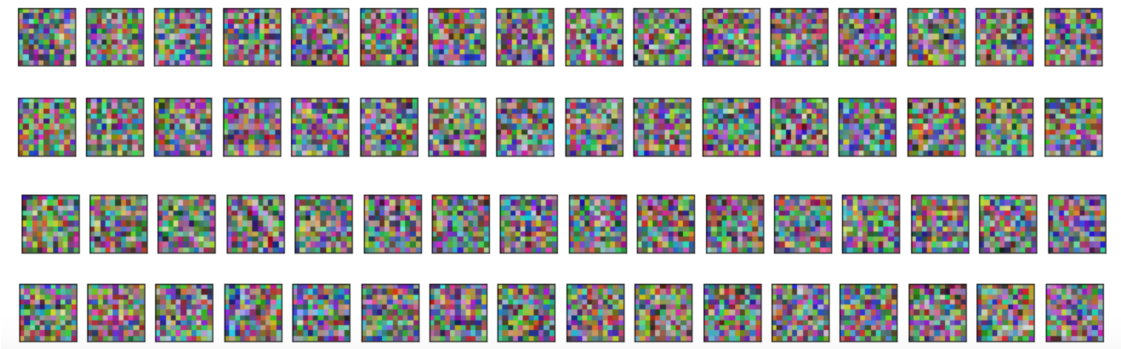
Weight Initialization was Xavier uniform where the weights are drawn from a distribution with zero mean and a variance dependent on the number of incoming neurons from the previous layer. It helps in attaining a global minimum of the cost function faster and more efficiently, and also prevents vanishing gradients. The weights are still random but differ in range depending on the size of the previous layer of neurons.

Hyper-parameters:
Optimizer used was SGD with learning rate 0.001 and momentum factor 0.9 with cross entropy loss. Batch size was 64 and the model was trained for 100 epochs.

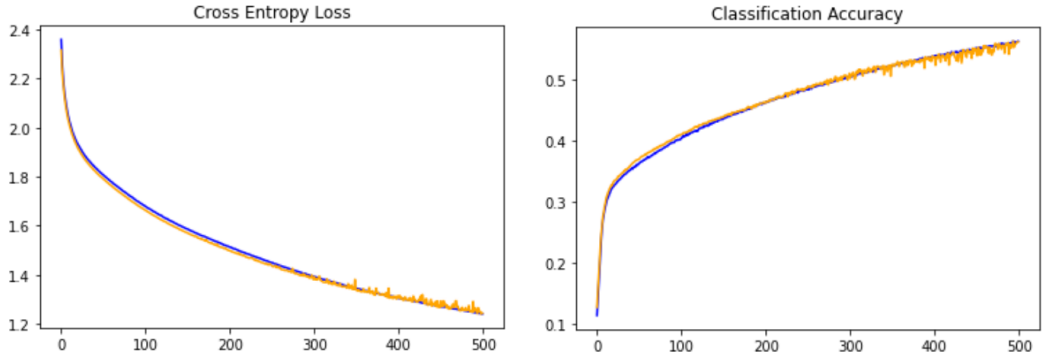
Part 2 (20 points)

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: Code: Appendix 3.2
Image showing the filters:



Train (blue) and test (orange), loss (<-) and accuracy (->) as a function of epochs:



FINAL TEST DATA ACCURACY : 56.10 %

Weight Initialization was the same as earlier, Xavier uniform. Optimizer used was SGD with learning rate 0.0001 and momentum factor 0.5 with cross entropy loss. The model was trained for 500 epochs with batch size 96.

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:

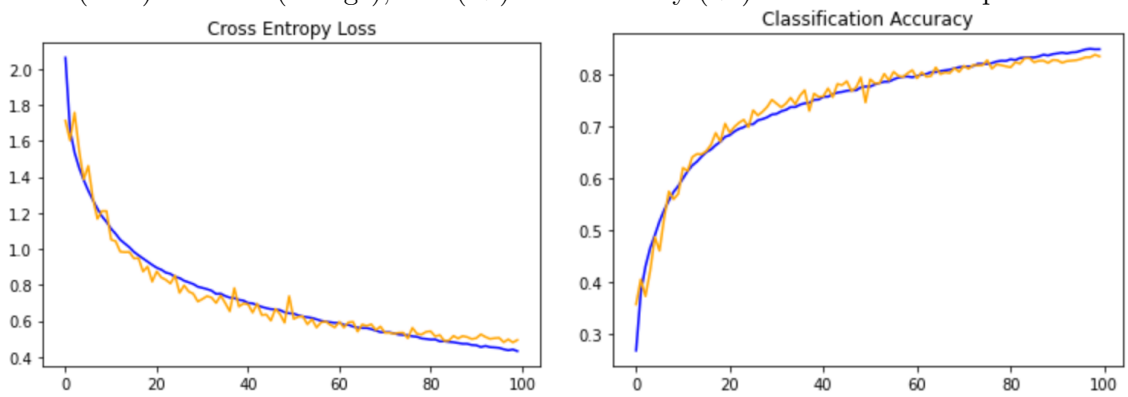
Code: Appendix 3.3

In order to improve accuracy, addition of blocks of conv, conv, pool layers was experimented with. Size of filters in each layer was experimented with, and a fully connected layer was added right before softmax. Dropout was also addded to reduce overfitting. The final architecture is given below:

(0) conv1: 32 filters of size 3x3 with Relu activation
(1) conv2: 32 filters of size 3x3 with Relu activation
(2) pool1: 2 x 2 Max Pooling (stride 2)
(3) drop1: Dropout 0.25
(4) conv3: 64 filters of size 3x3 with Relu activation
(5) conv4: 64 filters of size 3x3 with Relu activation
(6) pool2: 2 x 2 Max Pooling (stride 2)

(7) drop2: Dropout 0.25
(8) conv5: 128 filters of size 3x3 with Relu activation
(9) conv6: 128 filters of size 3x3 with Relu activation
(10) pool3: 2 x 2 Max Pooling (stride 2)
(11) drop3: Dropout 0.25
(12) fc1: dense layer with out dim 10 and Relu activation
(13) drop4: Dropout 0.5
(14) fc2: dense layer with out dim 10
(15) softmax: Softmax layer
Batch normalization was also added after each conv layer.

Train (blue) and test (orange), loss (<-) and accuracy (->) as a function of epochs:



Final test accuracy was 83.46 %. Model was optimzed with SGD with learning rate 0.001 and 0.9 momentum, using Xavier uniform initialization for weights. It was trained for 100 epochs with a batch size of 64.

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:

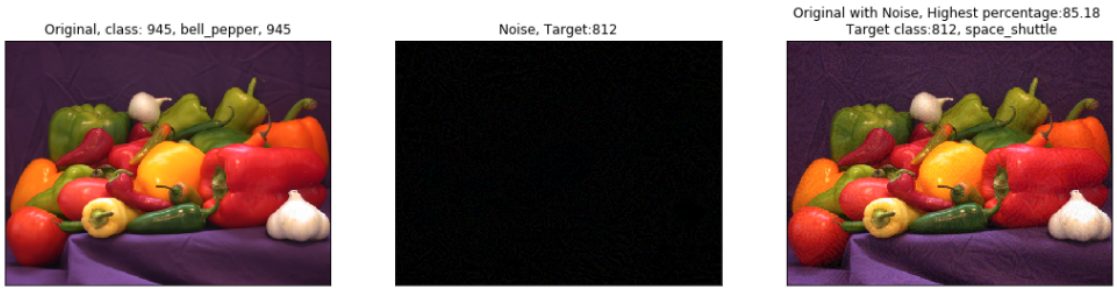


Figure 1: Peppers (Original Image prediction : bell pepper 90.51)

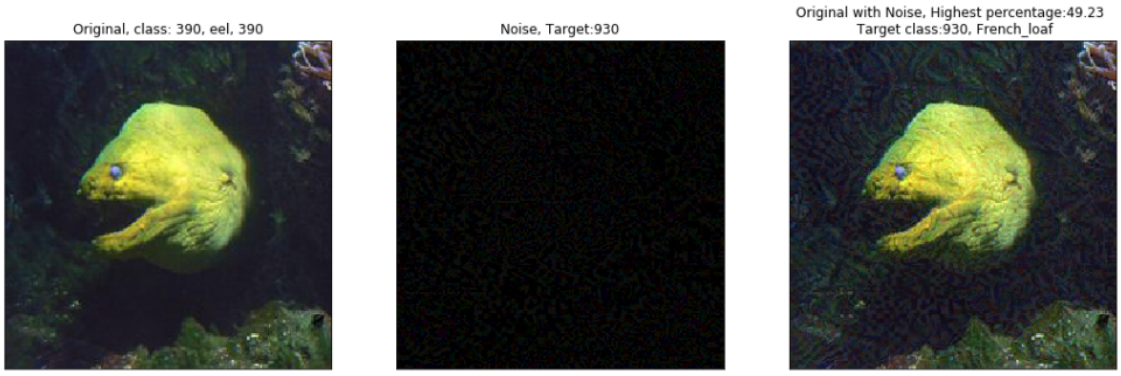


Figure 2: Eel (Original Image prediction : eel 89.09)

Figure 3: Dragonfly (Original Image prediction : dragonfly 93.39)

Figure 4: Junco (Original Image prediction : junco 90.99)

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:

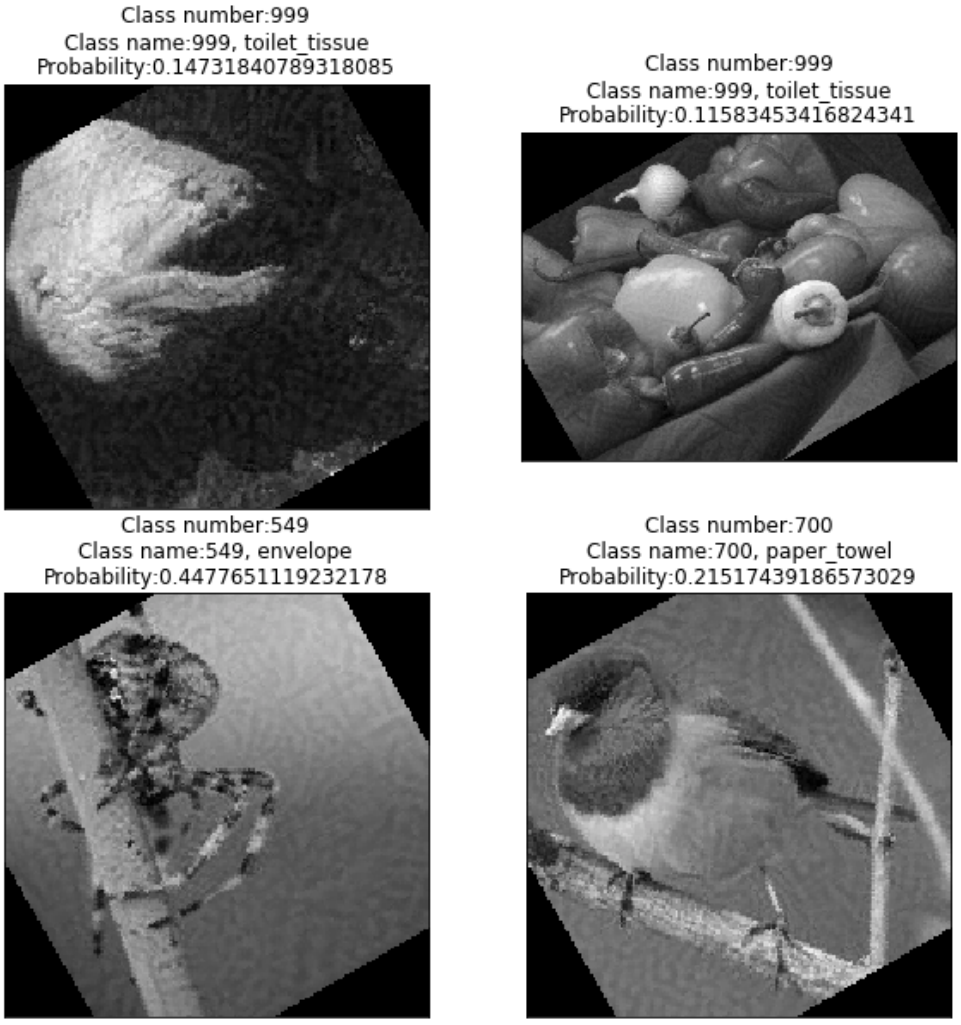


Figure 5: Modified adversarial images with new most likely class and probabilities

The CNN was fooled successfully, with no original image being classified accurately. This may be attributed to no additional transformation variations for images in the train set such as mirroring or rotation. This makes the model unrobust towards such transformations, which have been found by the fooling method. We can make the model robust by including such transformations in the original training data.

Code Appendix

Appendix 1

```
from torchvision import transforms, models
import torch
from torch.autograd import Variable
import matplotlib.pyplot as plt
from PIL import Image
import json
from matplotlib import gridspec
import numpy as np

net = models.vgg16(pretrained=True)

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

class_idx = json.load(open("imagenet_class_index.json"))
idx2label = [class_idx[str(k)][1] for k in range(len(class_idx))]

img = Image.open('peppers.jpg')
plt.imshow(img)

pic = Variable(centre_crop(img).unsqueeze(0))
out = net(pic)

top3 = []
probs3 = []
prob = torch.nn.functional.softmax(out[0], dim=0).sort()
for idx in range(len(prob[0])-3,len(prob[0])):
    top3.insert(0,idx2label[prob[1][idx]])
    probs3.insert(0,round(float(prob[0][idx]),4))
print(top3)
print(probs3)
```

```
# Q1p2
modulelist = list(net.features.modules())

def filter_outputs(image, layer_to_visualize):
    if layer_to_visualize < 0:
        layer_to_visualize += 31
    output = None
    name = None
    for count, layer in enumerate(modulelist[1:]):
        image = layer(image)
        if count == layer_to_visualize:
            output = image
            name = str(layer)
    filters = []
    output = output.cpu().data.squeeze()
    for i in range(output.shape[0]):
        filters.append(output[i,:,:])
    fig = plt.figure()
    plt.rcParams["figure.figsize"] = (12, 4)
    for i in range(10):
        fig.add_subplot(2,5,i+1)
        imgplot = plt.imshow(filters[i])
        plt.axis('off')

for i in [0,1,2,3,4,14,15,16,17,18,24,25,26,27,28]:
    print("Layer "+str(i))
    filter_outputs(pic, i)
```

Appendix 2

```
from torchvision import transforms, models
import torch
from torch.autograd import Variable
import matplotlib.pyplot as plt
import tensorflow_datasets as tfds
tfds.disable_progress_bar()
import tensorflow as tf
from PIL import Image
import json
from matplotlib import gridspec
import numpy as np

# Q2p1

net = models.vgg16(pretrained=True)

for param in net.parameters():
  param.requires_grad = False

net.classifier = net.classifier[:-1]

def load_image(datapoint):
  input_image = tf.image.resize(datapoint['image'], (224, 224))
  input_image = tf.cast(input_image, tf.float32) / 255.0  #### normalized
  return input_image, datapoint['label']

dataset, info = tfds.load('oxford_iiit_pet:3.*.*', with_info=True)

train = dataset['train'].map(load_image, \
        num_parallel_calls=tf.data.experimental.AUTOTUNE)
test = dataset['test'].map(load_image)

TEST_LENGTH = info.splits['test'].num_examples
TRAIN_LENGTH = info.splits['train'].num_examples

train_X = []
train_Y = []
i=0
for image, label in train.take(TRAIN_LENGTH):
```

```
  pic = torch.from_numpy(image.numpy())
  pic.transpose_(0,2)
  train_X.append(net(pic.unsqueeze(0)).numpy())
  train_Y.append(label)
  i += 1
  if i%100 == 0:
      print(i)
test_X = []
test_Y = []
for image, label in test.take(TEST_LENGTH):
  pic = torch.from_numpy(image.numpy())
  pic.transpose_(0,2)
  test_X.append(net(pic.unsqueeze(0)).numpy())
  test_Y.append(label)
  i += 1
  if i%100 == 0:
      print(i)

Xtrain = np.array(train_X).reshape(-1,4096)
Ytrain = np.array(train_Y).reshape(-1,1)
Xtest = np.array(test_X).reshape(-1,4096)
Ytest = np.array(test_Y).reshape(-1,1)

from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0,max_iter=1000).fit(Xtrain, Ytrain.flatten())
preds = clf.predict(Xtest)

_LABEL_CLASSES = [
    "Abyssinian", "american_bulldog", "american_pit_bull_terrier",
    "basset_hound", "beagle", "Bengal", "Birman", "Bombay", "boxer",
    "British_Shorthair", "chihuahua", "Egyptian_Mau", "english_cocker_spaniel",
    "english_setter", "german_shorthaired", "great_pyrenees", "havanese",
    "japanese_chin", "keeshond", "leonberger", "Maine_Coon",
    "miniature_pinscher", "newfoundland", "Persian", "pomeranian", "pug",
    "Ragdoll", "Russian_Blue", "saint_bernard", "samoyed", "scottish_terrier",
    "shiba_inu", "Siamese", "Sphynx", "staffordshire_bull_terrier",
    "wheaten_terrier", "yorkshire_terrier"
]
from sklearn.metrics import classification_report
print(classification_report(Ytest, preds,target_names=_LABEL_CLASSES))
```

Appendix 3.1

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import torchvision
from torchvision import datasets, models, transforms
from tqdm import tqdm
from torch.nn import functional as F

batch_size = 64

tsfms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.4914, 0.4822, 0.4465], [0.247, 0.243, 0.261])
])

train_cifar_data = torchvision.datasets.CIFAR10('.', train=True,\
            download=False, transform=tsfms)

train_loader = torch.utils.data.DataLoader(train_cifar_data,
                                           batch_size=batch_size,
                                           shuffle=True,
                                           num_workers=0)

test_cifar_data = torchvision.datasets.CIFAR10('.', train=False,\
            download=False, transform=tsfms)
test_loader = torch.utils.data.DataLoader(test_cifar_data,
                                           batch_size=batch_size,
                                           shuffle=False,
                                           num_workers=0)

train_iter = iter(train_loader)

class CNN(torch.nn.Module):
    def __init__(self, input_channels):
        super(CNN, self).__init__()
        self.c_1 = torch.nn.Conv2d(input_channels, 64, (11,11))
        self.c_2 = torch.nn.Conv2d(64, 128, (3,3))
        self.c_3 = torch.nn.Conv2d(128, 128, (3,3))
        self.linear = torch.nn.Linear(4608, 10)
```

```
        torch.nn.init.xavier_uniform_(self.c_1.weight)
        torch.nn.init.xavier_uniform_(self.c_2.weight)
        torch.nn.init.xavier_uniform_(self.c_3.weight)


    def forward(self, x):
        c_1_out = F.relu(self.c_1(x))
        mp_1 = torch.nn.MaxPool2d((2,2),2)
        mp_1_out = mp_1(c_1_out)
        c_2_out = F.relu(self.c_2(mp_1_out))
        c_3_out = F.relu(self.c_3(c_2_out))
        avg_p = torch.nn.AvgPool2d((2,2), 1)
        avg_p_out = avg_p(c_3_out)
        flat = torch.nn.Flatten(1,3)
        flat_out = flat(avg_p_out)
        linear_out = self.linear(flat_out)
        sm = torch.nn.Softmax(1)
        sm_out = sm(linear_out)
        return sm_out

model = CNN(3)
# model.eval()

epochs = 100
lr_rate = 0.001
mom = 0.9

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=lr_rate, momentum=mom)

loss_hist = []
acc_hist = []
test_loss_hist = []
test_acc_hist = []

for epoch in range(int(epochs)):
    for i, (imgs, labels) in enumerate(train_loader):

        optimizer.zero_grad()
        outputs = model(imgs)
        acc = (outputs.argmax(1) == labels).sum().numpy() / batch_size
```

```
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # print(loss, acc)

    loss_hist.append(np.array(loss.data.unsqueeze(0))[0])
    acc_hist.append(acc)

    #Test Loss:

    print(f'Calculating Test Loss for Epoch {epoch}')
    _, (test_features, test_labels) = next(enumerate(test_loader))
    test_outputs = model(test_features)
    test_loss = criterion(test_outputs, test_labels)
    test_loss_hist.append(test_loss)
    test_acc = (test_outputs.argmax(1) == labels).sum().numpy() / batch_size
    test_acc_hist.append(test_acc)

plt.plot(loss_hist, label='train', color='blue')
plt.plot(test_loss_hist, label='test', color='orange')
plt.title('Cross Entropy Loss')
plt.show()

plt.plot(acc_hist, color='blue')
plt.plot(test_acc_hist, color='orange')
plt.title('Classification Accuracy')
plt.show()

# torch.save(model, 'custom_cnn')
model = torch.load('custom_cnn')

from torch import nn

class CNN_bottom(nn.Module):
    def __init__(self, original_model, exclude):
        super(CNN_bottom, self).__init__()
        self.features = nn.Sequential(*list(original_model.children())[:-1*exclude])

    def forward(self, x):
        x = self.features(x)
        return x
```

```
CNN_bottom(model, 3)(imgs).shape

def norm_f(t):
    f = t.numpy().transpose(1, 2, 0)
    sub = f - f.min()
    return (sub/sub.max())

c_1_filters = model.c_1.weight.data
c_1_filters.shape

f, ax = plt.subplots(4,16,figsize=(16,16))
for i in range(64):
ax = plt.subplot(4, 16, i+1)
ax.axis('off')
plt.imshow(norm_f(c_1_filters[i]))
```

Appendix 3.2

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import torchvision
from torchvision import datasets, models, transforms
from tqdm import tqdm

batch_size = 96

tsfms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.4914, 0.4822, 0.4465], [0.247, 0.243, 0.261])
])

train_cifar_data = torchvision.datasets.CIFAR10('.', train=True, \
            download=False, transform=tsfms)

train_loader = torch.utils.data.DataLoader(train_cifar_data,
                                           batch_size=batch_size,
                                           shuffle=True,
                                           num_workers=0)

test_cifar_data = torchvision.datasets.CIFAR10('.', train=False, \
            download=False, transform=tsfms)
test_loader = torch.utils.data.DataLoader(test_cifar_data,
                                           batch_size=batch_size,
                                           shuffle=False,
                                           num_workers=0)

train_iter = iter(train_loader)

from torch.nn import functional as F

class CNN_with_batch_norm(torch.nn.Module):
    def __init__(self, input_channels):
        super(CNN_with_batch_norm, self).__init__()
        self.c_1 = torch.nn.Conv2d(input_channels, 64, (11,11))
        self.c_1_bn = torch.nn.BatchNorm2d(64)
```

```
        self.c_2 = torch.nn.Conv2d(64, 128, (3,3))
        self.c_2_bn = torch.nn.BatchNorm2d(128)

        self.c_3 = torch.nn.Conv2d(128, 128, (3,3))
        self.c_3_bn = torch.nn.BatchNorm2d(128)

        self.linear = torch.nn.Linear(4608, 10)


    def forward(self, x):
        ##Block 1:
        c_1_out = F.relu(self.c_1(x))
        c_1_bn_out = self.c_1_bn(c_1_out)
        mp_1 = torch.nn.MaxPool2d((2,2),2)
        mp_1_out = mp_1(c_1_bn_out)

        #Block 2:
        c_2_out = F.relu(self.c_2(mp_1_out))
        c_2_bn_out = self.c_2_bn(c_2_out)

        #Block 3:
        c_3_out = F.relu(self.c_3(c_2_bn_out))
        c_3_bn_out = self.c_3_bn(c_3_out)

        #Average Pool
        avg_p = torch.nn.AvgPool2d((2,2), 1)
        avg_p_out = avg_p(c_3_bn_out)

        flat = torch.nn.Flatten(1,3)
        flat_out = flat(avg_p_out)

        linear_out = self.linear(flat_out) #shape (b, 10)

        sm = torch.nn.Softmax(1)
        sm_out = sm(linear_out)


        return sm_out

model = CNN_with_batch_norm(3)
```

```
# model.eval()

epochs = 500
lr_rate = 0.0001
mom = 0.5

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=lr_rate, momentum=mom)

loss_hist = []
acc_hist = []
test_loss_hist = []
test_acc_hist = []

for epoch in range(int(epochs)):
    for i, (imgs, labels) in enumerate(train_loader):

        optimizer.zero_grad()
        outputs = model(imgs)
        acc = (outputs.argmax(1) == labels).sum().numpy() / batch_size
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # print(loss, acc)

    loss_hist.append(np.array(loss.data.unsqueeze(0))[0])
    acc_hist.append(acc)

    #Test Loss:

    print(f'Calculating Test Loss for Epoch {epoch}')
    _, (test_features, test_labels) = next(enumerate(test_loader))
    test_outputs = model(test_features)
    test_loss = criterion(test_outputs, test_labels)
    test_loss_hist.append(test_loss)
    test_acc = (test_outputs.argmax(1) == labels).sum().numpy() / batch_size
    test_acc_hist.append(test_acc)

plt.plot(loss_hist, label='train', color='blue')
plt.plot(test_loss_hist, label='test', color='orange')
plt.title('Cross Entropy Loss')
```

```
plt.show()

plt.plot(acc_hist, color='blue')
plt.plot(test_acc_hist, color='orange')
plt.title('Classification Accuracy')
plt.show()

# torch.save(model, 'custom_cnn')
model = torch.load('custom_cnn')

from torch import nn

class CNN_bottom(nn.Module):
    def __init__(self, original_model, exclude):
        super(CNN_bottom, self).__init__()
        self.features = nn.Sequential(*list(original_model.children())[:-1*exclude])

    def forward(self, x):
        x = self.features(x)
        return x

CNN_bottom(model, 3)(imgs).shape

def norm_f(t):
    f = t.numpy().transpose(1, 2, 0)
    sub = f - f.min()
    return (sub/sub.max())

c_1_filters = model.c_1.weight.data
c_1_filters.shape

f, ax = plt.subplots(4,16,figsize=(16,16))
for i in range(64):
ax = plt.subplot(4, 16, i+1)
ax.axis('off')
plt.imshow(norm_f(c_1_filters[i]))
```

Appendix 3.3

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import torchvision
from torchvision import datasets, models, transforms
from tqdm import tqdm

batch_size = 64

tsfms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.4914, 0.4822, 0.4465], [0.247, 0.243, 0.261])
])

train_cifar_data = torchvision.datasets.CIFAR10('.', train=True, \
            download=False, transform=tsfms)

train_loader = torch.utils.data.DataLoader(train_cifar_data,
                                           batch_size=batch_size,
                                           shuffle=True,
                                           num_workers=0)

test_cifar_data = torchvision.datasets.CIFAR10('.', train=False, \
            download=False, transform=tsfms)
test_loader = torch.utils.data.DataLoader(test_cifar_data,
                                           batch_size=batch_size,
                                           shuffle=False,
                                           num_workers=0)

train_iter = iter(train_loader)

from torch.nn import functional as F

class CNN_p3(torch.nn.Module):
    def __init__(self, input_channels):
        super(CNN_p3, self).__init__()
        self.c_1 = torch.nn.Conv2d(input_channels, 32, (3,3))
        self.c_1_bn = torch.nn.BatchNorm2d(32)
```

```
        self.c_2 = torch.nn.Conv2d(32, 32, (3,3))

        self.c_3 = torch.nn.Conv2d(32, 64, (3,3))
        self.c_4 = torch.nn.Conv2d(64, 64, (3,3))
        self.c_2_bn = torch.nn.BatchNorm2d(64)

        self.c_5 = torch.nn.Conv2d(64, 128, (3,3))

        self.c_6 = torch.nn.Conv2d(128, 128, (3,3))
        self.c_3_bn = torch.nn.BatchNorm2d(128)

        self.linear = torch.nn.Linear(3200, 10)
        self.drop25 = torch.nn.Dropout(p=0.25)
        self.drop5 = torch.nn.Dropout(p=0.5)
        self.mp_1 = torch.nn.MaxPool2d((2,2),2)
        self.avg_p = torch.nn.AvgPool2d((2,2), 1)

    def forward(self, x):
        ##Block 1:
        x = F.relu(self.c_1(x))
        x = self.c_1_bn(x)
        x = F.relu(self.c_2(x))
        x = self.c_1_bn(x)

        x = self.mp_1(x)
        x = self.drop25(x)

        #Block 2:
        x = F.relu(self.c_3(x))
        x = self.c_2_bn(x)
        x = F.relu(self.c_4(x))
        x = self.c_2_bn(x)

        x = self.drop25(x)

        #Block 3:
        x = F.relu(self.c_5(x))
        x = self.c_3_bn(x)
        x = F.relu(self.c_6(x))
        x = self.c_3_bn(x)
```

```
        x = self.drop25(x)

        x = self.avg_p(x)
        #Flatten
        flat = torch.nn.Flatten(1,3)
        flat_out = flat(x)
        #Linear map to size (b,10)
        linear_out = self.linear(flat_out) #shape (b, 10)

        sm = torch.nn.Softmax(1)
        sm_out = sm(linear_out)
        return sm_out

model = CNN_p3(3)
# model.eval()

imgs, labels = train_iter.next()

imgs.shape

model(imgs)

epochs = 100
lr_rate = 0.001
mom = 0.9

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=lr_rate, momentum=mom)

loss_hist = []
acc_hist = []
test_loss_hist = []
test_acc_hist = []

for epoch in range(int(epochs)):
    for i, (imgs, labels) in enumerate(train_loader):

        optimizer.zero_grad()
        outputs = model(imgs)
        acc = (outputs.argmax(1) == labels).sum().numpy() / batch_size
```

```
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            # print(loss, acc)

        loss_hist.append(np.array(loss.data.unsqueeze(0))[0])
        acc_hist.append(acc)

        #Test Loss:

        print(f'Calculating Test Loss for Epoch {epoch}')
        _, (test_features, test_labels) = next(enumerate(test_loader))
        test_outputs = model(test_features)
        test_loss = criterion(test_outputs, test_labels)
        test_loss_hist.append(test_loss)
        test_acc = (test_outputs.argmax(1) == labels).sum().numpy() / batch_size
        test_acc_hist.append(test_acc)

plt.plot(loss_hist, label='train', color='blue')
plt.plot(test_loss_hist, label='test', color='orange')
plt.title('Cross Entropy Loss')
plt.show()

plt.plot(acc_hist, color='blue')
plt.plot(test_acc_hist, color='orange')
plt.title('Classification Accuracy')
plt.show()
```

```
1
2  import os
3  import cv2
4  import numpy as np
5  from torch.optim import SGD
6  from torchvision import models
7  from torch.nn import functional
8  from PIL import Image
9  from PIL import ImageOps
10 import torch
11 from torch.autograd import Variable
12 import copy
13 import matplotlib.pyplot as plt
14 from torchvision import transforms
15
16 def preprocess_image(cv2im, resize_im=True):
17     mean = [0.485, 0.456, 0.406]
18     std = [0.229, 0.224, 0.225]
19     if resize_im:
20         cv2im = cv2.resize(cv2im, (224, 224))
21     im_as_arr = np.float32(cv2im)
22     im_as_arr = np.ascontiguousarray(im_as_arr[..., ::-1])
23     im_as_arr = im_as_arr.transpose(2, 0, 1)  # Convert array to D,W,H
24     for channel, _ in enumerate(im_as_arr):
25         im_as_arr[channel] /= 255
26         im_as_arr[channel] -= mean[channel]
27         im_as_arr[channel] /= std[channel]
28     im_as_ten = torch.from_numpy(im_as_arr).float()
29     im_as_ten.unsqueeze_(0)
30     im_as_var = Variable(im_as_ten, requires_grad=True)
31     return im_as_var
32
33 def recreate_image(im_as_var):
34     reverse_mean = [-0.485, -0.456, -0.406]
35     reverse_std = [1 / 0.229, 1 / 0.224, 1 / 0.225]
36     recreated_im = copy.copy(im_as_var.data.numpy()[0])
37     for c in range(3):
38         recreated_im[c] /= reverse_std[c]
39         recreated_im[c] -= reverse_mean[c]
40     recreated_im[recreated_im > 1] = 1
41     recreated_im[recreated_im < 0] = 0
42     recreated_im = np.round(recreated_im * 255)
43     recreated_im = np.uint8(recreated_im).transpose(1, 2, 0)
44     recreated_im = recreated_im[..., ::-1]
45     return recreated_im
46
47 def preprocess_image(PIL_image):
48     transform = transforms.ToTensor()
49     img_t = transform(PIL_image)
```

```
    _img = torch.unsqueeze(img_t, 0)
    X = _img
    img = torch.tensor(_img.numpy(), requires_grad=True)
    return img

def get_params(i):
    example_list = [
        ['peppers.jpg', 812, 'shuttle'],
        ['eel.JPG', 930, 'french_loaf'],
        ['bird.JPG', 783, 'screw'],
        ['dragonfly.png', 526, 'desk']
    ]
    img_path = example_list[i][0]
    target_class = example_list[i][1]
    target_class_name = example_list[i][2]
    original_image = cv2.imread(img_path, 1)
    pretrained_model = models.alexnet(pretrained=True)
    return original_image, target_class, target_class_name,
    pretrained_model

class fooling():
    def __init__(self, model, initial_image, target_class,
    minimum_confidence, original_class, target_class_name):
        self.model = model
        self.model.eval()
        self.target_class = target_class
        self.minimum_confidence = minimum_confidence
        self.original_image = initial_image
        self.initial_image = initial_image
        self.original_class = original_class
        self.target_class_name = target_class_name
        if not os.path.exists('../generated'):
            os.makedirs('../generated')

    def generate(self):
        for i in range(1, 500):
            self.processed_image = preprocess_image(self.initial_image)
            optimizer = SGD([self.processed_image], lr=2)
            output = self.model(self.processed_image)
            target_confidence =
    functional.softmax(output)[0][self.target_class].data.numpy()
            if target_confidence > self.minimum_confidence:
                confirmation_image = cv2.imread('../generated/generated_'
    + str(self.original_class) + '_' + str(self.target_class) + '_' +
    target_class_name + '.jpg', 1)
                confirmation_processed_image =
    preprocess_image(confirmation_image)
                op = preprocess_image(self.original_image)
                confirmation_output =
    self.model(confirmation_processed_image)
```

```
93                  softmax_confirmation =
    functional.softmax(confirmation_output)[0][self.target_class].data.numpy()
94                  if softmax_confirmation > self.minimum_confidence:
95                      noise_tensor = op - confirmation_processed_image
96                      noise_image = recreate_image(noise_tensor)
97                      cv2.imwrite('../generated/noise_' +
    str(self.original_class) + '_' + str(self.target_class) + '_' +
    target_class_name + '.jpg', noise_image)
98                      original_image = recreate_image(op)
99                      cv2.imwrite('../generated/orig_' +
    str(self.original_class) + '.jpg', original_image)
100                     break
101
102             class_loss = -output[0, self.target_class]
103             print(str(i), 'target_conf',
    "{0:.4f}".format(target_confidence))
104             self.model.zero_grad()
105             class_loss.backward()
106             optimizer.step()
107
108             self.initial_image = recreate_image(self.processed_image)
109             cv2.imwrite('../generated/generated_' +
    str(self.original_class) + '_' + str(self.target_class) + '_'
    +target_class_name + '.jpg', self.initial_image)
110         return noise_image, confirmation_image
111
112 for target_example in range(1):
113     original_image, fooling_target_class, target_class_name,
    pretrained_model = get_params(target_example)
114     min_confidence = 0.99
115     fool = fooling(pretrained_model, original_image, fooling_target_class,
    min_confidence, target_example, target_class_name)
116     noise, generated_image = fool.generate()
117
118 f, axarr = plt.subplots(1,3, figsize=(18,18))
119 axarr[0].title.set_text('Original,
    class:'+str(original_class)+","+str(original_class_number))
120 axarr[0].imshow(original_image)
121 axarr[1].title.set_text('Noise, Transform
    from:'+str(original_class_number)+" to "+str(target))
122 axarr[1].imshow(noise)
123 axarr[2].title.set_text('Original+Noise, Largest
    percentage:'+str(float(percentage[idx]))+' \n Transformed
    class:'+str(classes[target])+","+str(target))
124 axarr[2].imshow(generated_image)
125 for ax in axarr:
126     ax.set_xticks([])
127     ax.set_yticks([])
128
129 def mod_image(image, index):
```

```
    im = Image.open(image)
    width, height = im.size
    im = ImageOps.mirror(im)
    left = int(0.2*width)
    up = int(0.2*height)
    border = (left, up, 0, 0) # left, up, right, bottom
    im = ImageOps.crop(im, border)
    im = im.rotate( 30, Image.NEAREST )
    im = im.convert('LA')
    im = im.convert("RGB")
    im.save("../generated/modified_"+index)
    return im

def predict(PIL_image):
    transform = transforms.ToTensor()
    img_t = transform(PIL_image)
    img = torch.unsqueeze(img_t, 0)
    pred = model(img)
    __, indices = torch.sort(pred, descending=True)
    percentage = torch.nn.functional.softmax(pred, dim=1)[0]
    idx = indices[0][0]
    return ("Class number:"+str(int(idx))+"\n"+"Class
    name:"+classes[idx]+"\n"+"Probability:"+str(float(percentage[idx])))

image_paths = ["390, eel_390_to_930, French_loaf_930.jpg",
               "945, bell_pepper_945_to_812, space_shuttle_812.jpg",
               "319, dragonfly_319_to_526, desk_526.jpg",
               "13, junco_13_to_783, screw_783.jpg"]
im=[]
s=[]
for i in image_paths:
    img = modify_image(i)
    v = predict(img)
    im.append(img)
    s.append(v)

f, axarr = plt.subplots(2,2, figsize=(10,10))
axarr[0][0].title.set_text(s[0])
axarr[0][0].imshow(im[0])
axarr[0][1].title.set_text(s[1])
axarr[0][1].imshow(im[1])
axarr[1][0].title.set_text(s[2])
axarr[1][0].imshow(im[2])
axarr[1][1].title.set_text(s[3])
axarr[1][1].imshow(im[3])

for ax in axarr[0]:
    ax.set_xticks([])
    ax.set_yticks([])
for ax in axarr[1]:
```

```
179         ax.set_xticks([])
180         ax.set_yticks([])
```