# Homework 2
# CS 5787 - Deep Learning
# Spring 2021

Yuxin Zhang - yz2726@cornell.edu

**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 0 - BatchNorm (5 points)

BatchNorm is a common technique that can accelerate and improve training, especially with deeper networks. In this problem, you will study BatchNorm's properties for a single 'dot product' neuron, but the results are the same if you have multiple neurons in a layer or use convolutional units.

Let

$$x_i = \mathbf{w}^T \mathbf{h}_i + b$$

be the output of the neuron, where $\mathbf{h}_i \in \mathcal{R}^d$ is a vector of inputs to the neuron, $\mathbf{w} \in \mathcal{R}^d$ are the weights of the neuron, and $b$ is the bias. BatchNorm is applied in the subsequent 'layer' before the non-linearity. After computing the activation of the neuron, the BatchNorm transformation of the output activations is given by

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}},$$

where the mean of the activation in the mini-batch is given by

$$\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i,$$

the variance is given by

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2,$$

and where $m$ is the mini-batch size.

Substitute in the neuron's activation function into the BatchNorm equations and simplify. What does this tell you about how BatchNorm impacts the weights and the bias of the neuron? How should you adjust your neural network's architecture if using BatchNorm?

**Solution:**

$$
\begin{aligned}
\hat{x}_i &= \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \\
&= \frac{\mathbf{w}^T\mathbf{h}_i + b - \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{h}_i + b)}{\sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{w}^T\mathbf{h}_i + b - \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{h}_i + b)\right)^2 + \epsilon}} \\
&= \frac{\mathbf{w}^T\mathbf{h}_i - \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{h}_i)}{\sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{w}^T\mathbf{h}_i - \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{h}_i)\right)^2 + \epsilon}} \\
&= \frac{\mathbf{w}^T(\mathbf{h}_i - \frac{1}{m}\sum_{i=1}^{m}\mathbf{h}_i)}{\sqrt{\frac{\mathbf{w}^T}{m}\sum_{i=1}^{m}\left(\mathbf{h}_i - \frac{1}{m}\sum_{i=1}^{m}\mathbf{h}_i\right)^2 + \epsilon}} \\
&= \frac{\sqrt{\mathbf{w}^T}(\mathbf{h}_i - \frac{1}{m}\sum_{i=1}^{m}\mathbf{h}_i)}{\sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{h}_i - \frac{1}{m}\sum_{i=1}^{m}\mathbf{h}_i\right)^2}}
\end{aligned} \tag{1}
$$

Weights would be shirked to the square root of the non-normalized one (calculated from the normalized $h_i$), and there would be no bias in the neuron due to the Batch Normalization.Do BatchNorm in each neuron after the matrix computation and before the activation function. Also, we can build deeper neural networks since the parameters during training would be smaller and more regularized. Also, we can have larger learning rate for more effective training.

# 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:**
Choosing the pre-trained ResNet-18 in pytorch, we use the transforms package from torchvision to perform the following transformation to the loaded image.

- Resize the image to 256 X 256 pixel
- Crop the image to 224 X 224 pixel around the center
- Convert the image to Tensor data type
- Normalize the image by using the mean([0.485, 0.456, 0.406]) and standard deviation([0.229, 0.224, 0.225]) from the pre-trained model

Attaining the 1000 labels from imagenet [1], we use the run the pre-trained model on the transformed image tensor and attain the top three(having the highest score) labels from the output using torch.topk. Also, applying torch.nn.functional.softmax on the model output, we are able to attain the probability of the top three labels output by the model. And the result are as the follows:
The image is predicted as bell pepper with 99.7342758178711 % probability
The image is predicted as cucumber, cuke with 0.17935766279697418% probability
The image is predicted as Granny Smith with 0.03717629984021187% probability

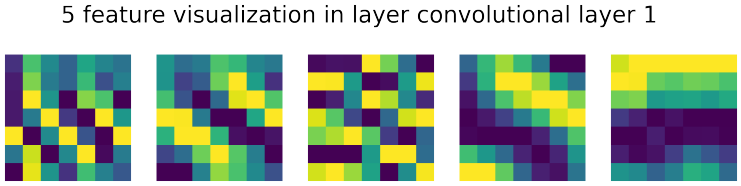
## Part 2 - Visualizing Feature Maps (5 points)

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.

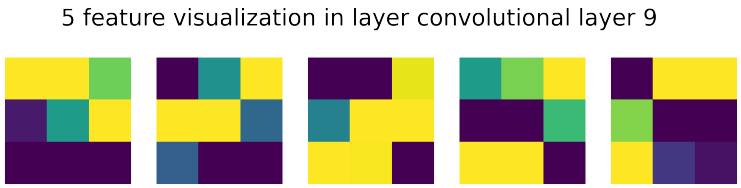
**Solution:**
ResNet-18 has 17 Convolutional layers and 1 fully connected layer. Before visualizing the feature maps in the network, we pull out the parameters in each convolutional layer and the layers by looping through the network structure and examine type of each layer. For each convolutional layer, the extracted parameters are the weights of its filter that defines the feature of which it extracts. And by first normalizing those filters, we visualizes the feature maps at a specific convolutional layer by simply calling the plt.imshow function. We selected first, 9th, 16th convolutional layer in ResNet-18 and visualized 5 feature maps from each of those layers.

- Layer 1 has 64 hidden neurons, each applies a 7X7 feature filter.

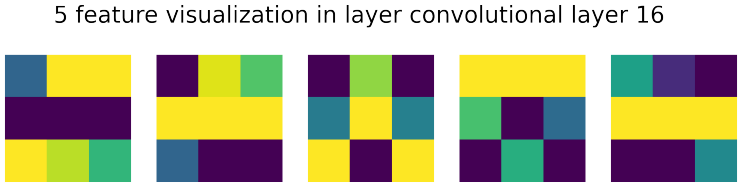



The last 2 feature maps from this map seems to find some cutting edges in the image

- Layer 9 has 128 hidden neurons, each applies a 3X3 feature filter.




- Layer 16 has 512 hidden neurons, each applies a 3X3 feature filter.




The structure of the network is that as the network go deeper (later layers), it has more neurons with each layer yet applies smaller filters to each neuron (more features get extracted in the deeper network).
It's especially hard to interpret the feature maps in the deeper layers, since the image lost much of its pixel spacial representation after several layers of convolution and activation.

## 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 500–6000 dimensions). You will need to resize the images to a size compatible with your network (usually $224 \times 224 \times 3$, but look at the documentation for the pre-trained system you selected). You should grab the output just after the last hidden layer or after global pooling (if it is 1000-dimensional, you will know you did it wrong).

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:**
Firstly, following the image file classification specified in trainval.txt and test.txt file, we loaded in each image using PIL, applied the previously defined transform(including Resize, CenterCrop,ToTensor and Normalize), and saved those tensors to train and test files(including feature and labels).The training set is of size 3680 and the test set is of size 3669.
Then we loaded the pre-trained ResNet-18, and built the feature_extractor by retaining the all model layers but the last one. Running the training and testing image features through the feature_extractor, we attained a feature vector of dimension 512 for each image input. Normalizing each feature vector to unit length by dividing by $L_2$ norm, we chose Logistic Regression as the linear classifier (from sklearn package) and trained the model by fitting the train feature vectors with the train labels. Besides, we also need to apply regularization to the model since 512 is a rather dimension (L2 regularization applied here).Applying .score method on testing features and labels from the trained classifier, we directly attained the mean-per-class accuracy on training set as 0.8529891304347826, and that on testing set as 0.42927228127555195.
Logistic Regression uses softmax function to produce output in probability format among classes, and applies Cross entropy as the loss function. It makes no assumptions about the class distributions in feature space and can be applied to multiple-class classification cases, like in our case. However, since the dataset has 512 dimension, which is rather high, even with the regularization, the model overfits (large difference between the accuracy in training and testing dataset), suggesting that a linear classifier can't capture the true feature complexities extracted by the pre-trained ResNet and thus generalize it on test dataset.

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

## Problem 3 - Training a Small CNN

## Part 1 (25 points)

For this problem you must use a toolbox. Train a CNN with three hidden convolutional layers that use the Mish activation function. Use 64 $11 \times 11$ filters for the first layer, followed by $2 \times 2$ max pooling (stride of 2). The next two convolutional layers will use 128 $3 \times 3$ filters followed by the Mish 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 across RGB channels and then divide by the new largest value across RGB channels.

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

**Solution:**
Based on the described network architecture, we applied PyTorch modules to build the NN as:

```
Net(
  (conv1): Conv2d(3, 64, kernel_size=(11, 11), stride=(1, 1))
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (conv3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1))
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc1): Linear(in_features=128, out_features=10, bias=True)
)
```

Since we use nn.CrossEntropyLoss() as the loss function, which already combines nn.LogSoftmax() and nn.NLLLoss(), we don't need to add another individual softmax layer in the network architecture.We use Mish activation function from [2]. The training set is of size 50000, and the test set is of size 10000. To tune the hyper-parameters (epoch, learning rate, batch size, optimizer), we randomly split 20% of the original training set as

the validation set. All trainings are conducted on GPU [3].
Initially, we trained on the NN with 80 epochs, learning rate of 0.0001, "Adam" optimizer, we plot the training and validation loss with each epoch on each batch size in the collection of [200,400,600, 800, 1000, 1500, 2000, 3000].

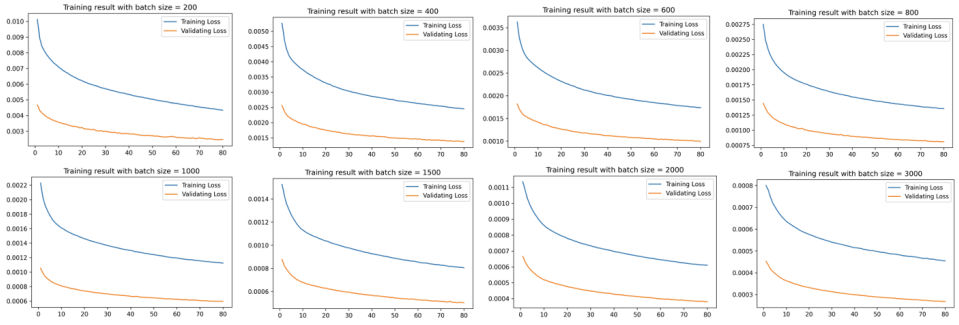



Figure 1: Training and Validation loss with different batch size

The Training loss is consistently bigger than that of the Validation loss for all batch sized trained, which means that the model hasn't really learned much from the data(underfitting), so we increase the learning rate to 0.0005 and set the epochs to 100, repeating the training and validating process for different batch size.

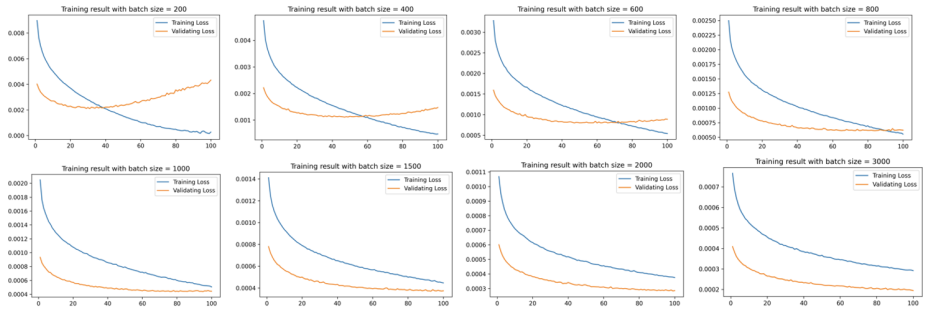



Figure 2: Training and Validation loss with different batch size

The network learns more as we can see convergence in training and validation loss. Also, we can see the model goes from over-fitting to under-fitting with larger batch sizes. However, training process still converge very slowly and converge incompletely even with 100 epochs. So we decided to switch to even larger learning rate (1e-3) and increase the epoch to 200.

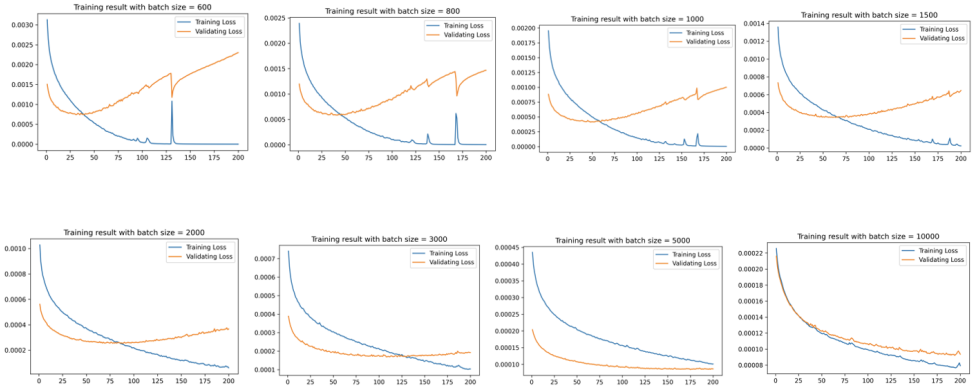



Figure 3: Training and Validation loss with different batch size

The model converges pretty well and have good training and validation loss with learning rate of 1e-3 and batch size of 10000.
Training with such hyper-parameters on the entire training dataset of size 50000 and training with 500 epoches for full convergence, we have the result as the following using Adam optimizer.
Storing the weight of the first convolutional layer in a collection, we normalized each of the 64 filters and visualized them in one large image
IMAGE SHOWING THE FILTERS

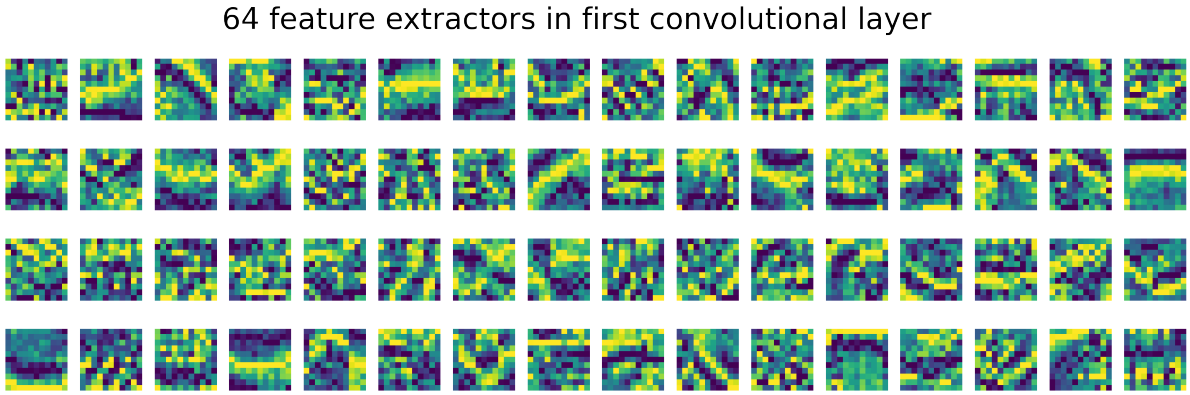



Figure 4: 64 Filters in the first convolutional layer

TRAINING LOSS AS FUNCTION OF EPOCHS

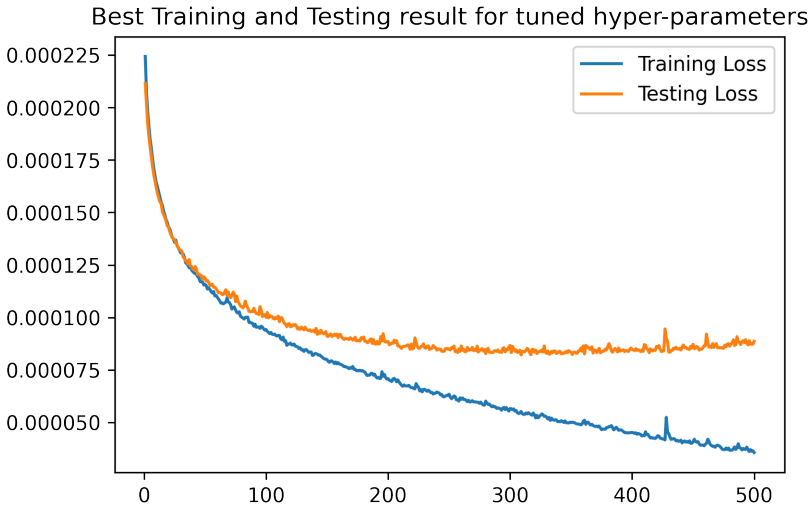



Figure 5: Training and testing loss with tuned model

Training loss on last epoch: 3.556086599826813e-05
Testing loss on last epoch: 8.859020471572876e-05
TEST DATA ACCURACY
Accuracy of the network on the test set: 67 %
WEIGHT INITIALIZATION INFORMATION
We use all the default weight initialization from PyTorch, and we have 2 basic modules that requires parameters: Convolutional module and Linear module(with Bias).
Convolutional module:The weight follows uniform distribution from (-stdv, stdv), with stdv $= \frac{1}{\sqrt{weight.size(1)}}$ [4]
Linear module:The weight and bias follows uniform distribution from (-stdv, stdv), with stdv $= \frac{1}{\sqrt{weight.size(1)}}$ [5]
DESCRIBE HYPER-PARAMETERS
The tuned model has batch size of 10000, optimizer = Adam,epoch number = 500.

## 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:**

Adding a batch norm operation at the end of each hidden layer(after the non-linear relu function) by using the nn.BatchNorm2d, we retrain the training dataset using the previously tuned hyper-parameters(batch size of 10000, optimizer = Adam,epoch number = 500).From the training and testing loss graph, we see a much better training loss convergence but the model over-fitted since the testing loss is going up after certain epochs. This is expected since with batch normalization, larger batch size should be applied. However, to set the benchmark for training loss comparison between the batched-normalized and non-batch-normalized NN, we still trained this model using the previously tuned hyper-parameters. The training and testing loss of each epoch is as follows:
Training loss on last epoch: 2.050423389300704e-07
Testing loss on last epoch: 0.00013541976213455201
Accuracy of the network on the test set using batch normalization is reported as 72%, a 7.5% increase from 67% of the one without batch normalization. Plotting the training loss

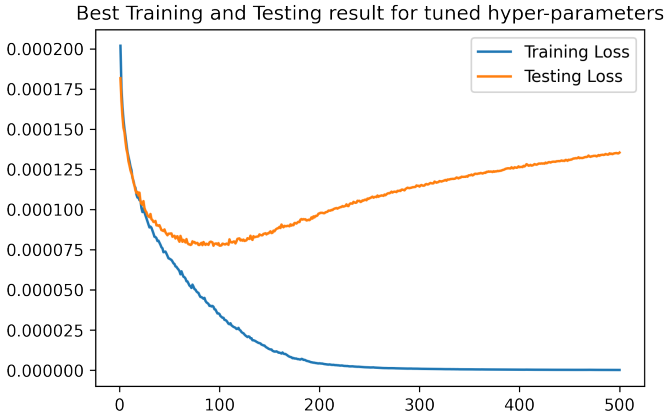



Figure 6: Training and testing loss batch normalized model

of the NN model with and without batch normalization, we clearly see that the training loss converges much quicker in the NN architecture with batch normalization.

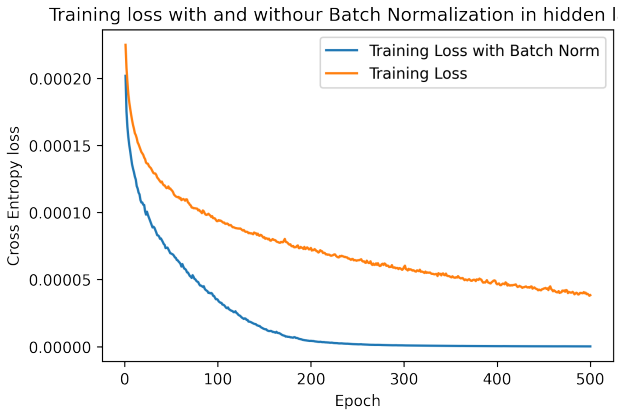



Figure 7: Training loss with and withour Batch Normalization in hidden layers

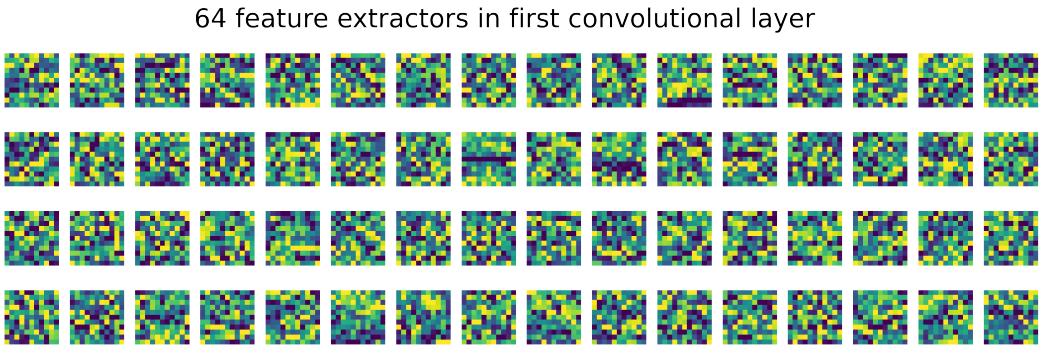



Figure 8: 64 filters visualized in the first convolutional layer with batch normalization

## 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?

And the dropout is conducted after the 5 Note: Your model should perform better than the one in Part 1 and Part 2.

**Solution:**

Trying a bunch of kernel sizes, number of convolutional layers, number of hidden unit, dropout rates in designing architecture, and tuned each of them on different batch size and learning rate and select the final architecture with the lowest validation set loss.
The final architecture is as follows:

```
(conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1))
(conv12): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
(conv1_bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
(conv2_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(conv3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1))
(conv3_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
(conv4_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(conv5): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1))
(conv5_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(conv6): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1))
(conv6_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(fc1): Linear(in_features=512, out_features=256, bias=True)
(fc2): Linear(in_features=256, out_features=128, bias=True)
(fc3): Linear(in_features=128, out_features=10, bias=True)
(dropout): Dropout(p=0.5, inplace=False)
```

Training such a model with batch size of 1000, learning rate of 1e-1 with Adam optimizer, The last epoch training and testing losses are
Training loss on last epoch: 3.311768591403961e-05
Testing loss on last epoch: 0.0005452457904815673
It achieves an test set accuracy of 76%, which is higher than the 72% reported by the batch normalized shallow network. This indicates that the well-regularized deeper network does extract informative features from the image data, thus leading to better generalization result on the test dataset.

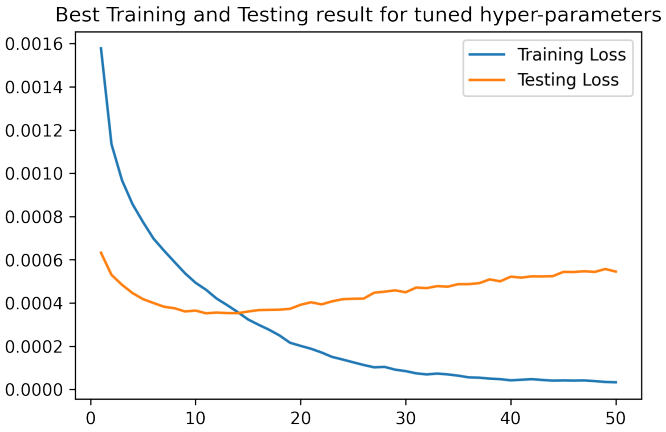



Figure 9: Training loss testing loss with deeper NN

## 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 $\epsilon$ 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 $\epsilon$ 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 $4 \times 3$ 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:**
The pre-trained model we use is ResNet18, and we choose four pictures that can be correctly classified by the pre-trained model on 1000 ImageNet classes. In attaining the adversarial image, we set the updating rate $\lambda = 0.01$ and the difference threshold $\epsilon = 0.1$ so that the adversarial image output would classify the first image (pepper) as image index of 812(space shuttle). The adversarial noise is the difference between the adversarial image and the original image. The result is as follows.
Although not visually obvious, the image that's previously classified as bell pepper with 99.7% are now classified as a space shuttle with 99.8% probability. The image that's previously classified as hot dog with 99.9% is now classified as pretzel with 10.9% probability. The image that's previously classified as cheeseburger with 99.6% is now classified as white shark with 74.7% probability.The image that's previously classified as lemon with 69.4% is now classified as tiger shark with 5.4% probability.

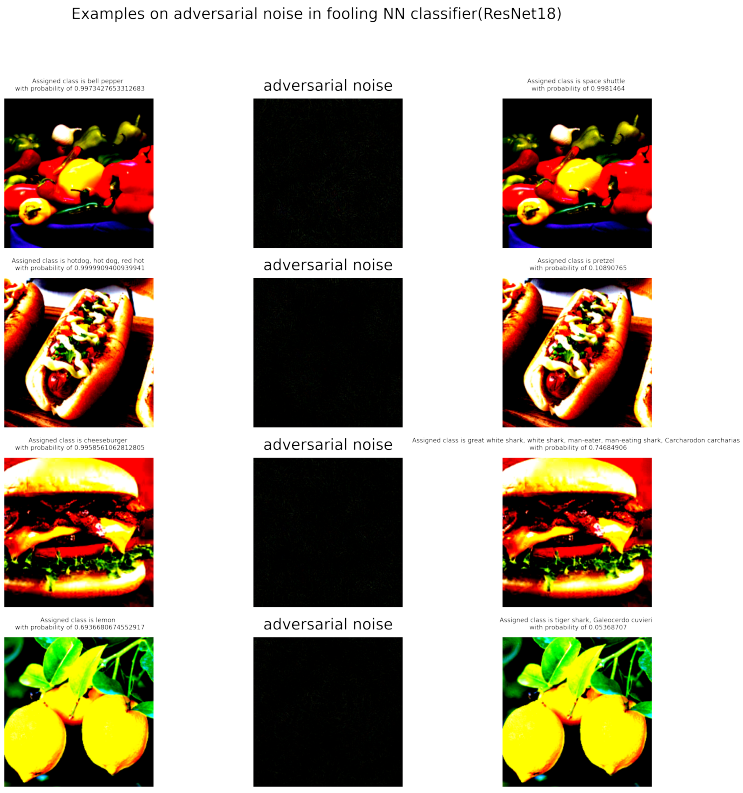



Figure 10: Examples on adversarial noise in fooling NN classifier(ResNet18)

## 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 to make a faux RGB image. 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:**

We conducted four separate transformations to the adversarial images computed in part 1(flipping, cropping, rotation, and gray-scaling). The flipping conducts horizontal flipping on the width axis to attain a mirror image by using torch.flip. The cropping con-

ducts center cropping by taking 89.5% of its original width and height (approximately 80% of the original image area), using transforms.CenterCrop(0.895*data.shape[2]. The rotation conducts 30 degree counter-clockwise rotation on the image by using textttransforms.functional.rotate(data,30). The grayscaling converts image to gray-scale and refills RGB channel by using transforms.Grayscale(num_output_channels=3). Then for each transformed image with different transformation applied, we predict their class and probability by running through the pre-trained ResNet18. The result is as follows.

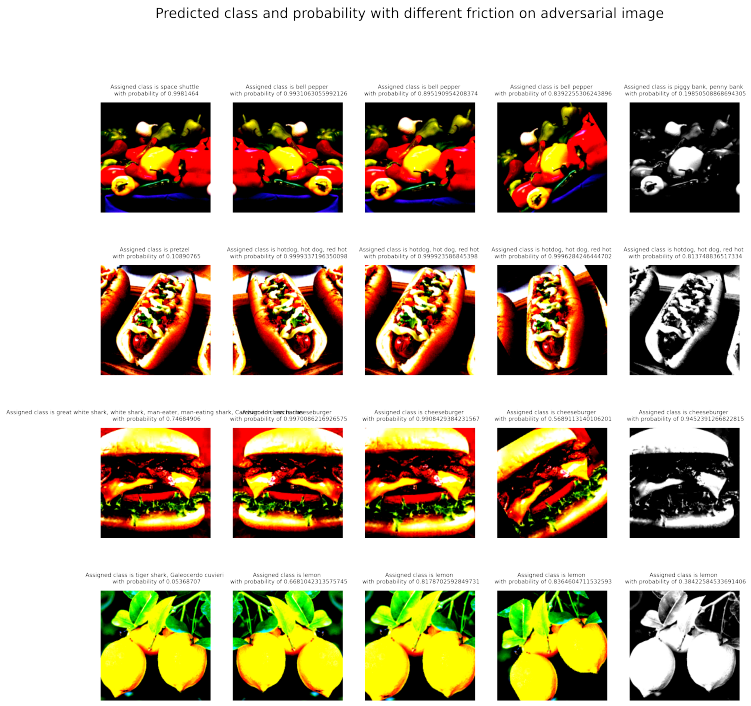



Figure 11: Predicted class and probability with different friction on adversarial image

For the flipping, cropping, rotation transformation on the adversarial image, the pre-trained ResNet showed robustness in dealing with the noise inserted into the adversarial image, and still classified them to the original, true classes with high probability. Only the pepper adversarial image with gray-scaling transformation shows mis-classification compared to its original true class, but it's still different from the adversarial image (then classified as space shuttle but now classified as piggy bank). In all, with all four transformations conducted on the adversarial image, the adversarial images shows weak robustness in mis-classifying image into certain classes from our four image examples selected.
The adversarial examples are especially week when transformations such as flipping, cropping and rotation are done to them because the pre-trained CNN-based ResNet18 model

is featured as invariant as such transformations. This means that the feature extractor of the model and recognize the transformed features on a larger scale to accurately classify the image, diminishing the effect of previously added-in noises. For the gray-scaling transformation, partial information is lost(including the noise) is lost in the transformation procedure, making the adversarial example not guaranteed to work.

# Code Appendix

## 0.1 Problem 1 Using a Pre-Trained CNN

### 0.1.1 Part 1 - Using Pre-Trained Deep CNN

```
import torch
import torchvision.models as models
from matplotlib import image
from matplotlib import pyplot
from torchvision import transforms
#use the resnet18 pre-trained
resnet18 = models.resnet18(pretrained=True)
from PIL import Image
import numpy as np
# load image and convert to matrix
img = Image.open('peppers.jpg')
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]
 )])
img_t = transform(img)
batch_t = torch.unsqueeze(img_t, 0)
resnet18.eval()
out = resnet18(batch_t)
import ast
# with open('imagenet_classes.txt') as f:
file = open("imagenet_classes.txt", "r")
contents = file.read()
dictionary = ast.literal_eval(contents)
file.close()
```

```
classes = list(dictionary.values())
_, index = torch.topk(out, 3)
percentage = torch.nn.functional.softmax(out, dim=1)[0] * 100
print(classes[index[0][0]], percentage[index[0][0]].item())
print(classes[index[0][1]], percentage[index[0][1]].item())
print(classes[index[0][2]], percentage[index[0][2]].item())
```

### 0.1.2 Part 2 - Visualizing Feature Maps

```
import torch
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import cv2 as cv
import argparse
resnet18 = models.resnet18(pretrained=True)  parameters = []
layers = []
model_component = list(resnet18.children())
count_conv = 0
for i in range(len(model_component)):
  if type(model_component[i]) == nn.Conv2d:
    count_conv += 1
    parameters.append(model_component[i].weight)
    layers.append(model_component[i])
  elif type(model_component[i]) == nn.Sequential:
    for j in range(len(model_component[i])):
            for p in model_component[i][j].children():
                if type(p) == nn.Conv2d:
                    count_conv += 1
                    parameters.append(p.weight)
                    layers.append(p)
import random
def visualize_feature_layer(layer_num):
  plt.figure(figsize=(20, 17))
  all = []
  five_visualize = []
  for i, filter in enumerate(parameters[layer_num]):

      tensor_col = filter[0, :, :].detach()
```

```
      tensor_col1 = tensor_col - torch.min(tensor_col, 0,keepdim=True)[0]
      tensor_col2 =  tensor_col1 / torch.max(tensor_col1, 0,keepdim=True)[0]
      all.append(tensor_col2)
  #sample 5
  five_visualize = random.choices(all, k=5)
  fig=plt.figure(dpi = 400, figsize = (8,2))

  for i in range(5):
    ax=fig.add_subplot(1,5,i+1)
    ax.imshow(five_visualize[i])
    ax.axis('off')

  fig.suptitle("5 feature visualization in layer convolutional layer %s"%
  (layer_num+1),fontsize = 14) # or plt.suptitle('Main title')
  plt.show()
visualize_feature_layer(0)
visualize_feature_layer(8)
visualize_feature_layer(15)
```

### 0.2 Problem 2 - Transfer Learning with a Pre-Trained CNN

```
from skimage.io import imread_collection
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
from matplotlib import image
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]
 )])
label_train = open("trainval.txt","r")
label_test = open("test.txt","r")
label1 = label_train.readlines()
label_train.close()
label2 = label_test.readlines()
label_test.close()
train_label = []
```

```
test_label = []
for i in range(len(label1)):
  train_label.append(label1[i].strip().split())
for i in range(len(label2)):
  test_label.append(label2[i].strip().split())

train_label = np.array(train_label)
test_label = np.array(test_label)
import glob
pre = "HW/HW2/images/images/"
image_name_col = glob.glob("HW/HW2/images/images/*.jpg")
image_train_col = []
image_test_col = []
label_train_ordered = []
label_test_ordered = []
non_refer = []
for i in range(len(image_name_col)):
    cur_image_name = image_name_col[i]
    cur_image = Image.open(cur_image_name ).convert("RGB")
    refer_name =cur_image_name[len(pre):-4]
    print(refer_name)
    cur_image_tensor = transform(cur_image)
    train_ref = train_label[train_label[:,0] == refer_name]
    test_ref = test_label[test_label[:,0] == refer_name]
    if len(train_ref) != 0 :
      image_train_col.append(cur_image_tensor)
      label_train_ordered.append(int(train_ref[0][1]))
    elif len(test_ref) != 0 :
      image_test_col.append(cur_image_tensor)
      label_test_ordered.append(int(test_ref[0][1]))
    else:
        non_refer.append(cur_image_name)
    cur_image.close()
import torch
dic_train = {}
dic_test = {}
for i in range(len(image_train_col)):
  dic_train[i] = image_train_col[i]
for i in range(len(image_test_col)):
  dic_test[i] = image_test_col[i]
torch.save(dic_train, 'HW/HW2/transformed_train_image_tensors.pt')
```

```
torch.save(dic_test, 'HW/HW2/transformed_test_image_tensors.pt')
import pandas as pd
label_train_dict = {"Label":label_train_ordered}
pd.DataFrame(label_train_dict).to_csv("HW/HW2/LabelImageTrain.csv", index = False)
label_test_dict = {"Label":label_test_ordered}
pd.DataFrame(label_test_dict).to_csv("HW/HW2/LabelImageTest.csv", index = False)
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torchvision.models as models
from matplotlib import image
from matplotlib import pyplot
from torchvision import transforms
from torch.utils.data import DataLoader, TensorDataset
image_train_col = torch.load('HW/HW2/transformed_train_image_tensors.pt')
image_test_col = torch.load('HW/HW2/transformed_test_image_tensors.pt')

input_train = torch.stack(list(image_train_col.values()))
input_test = torch.stack(list(image_test_col.values()))
model_ft = models.resnet18(pretrained=True)
torch.no_grad()
feature_extractor = torch.nn.Sequential(*list(model_ft.children())[:-1])
def feature_extraction(input, train_or_test):
  index = 0
  count = 0
  while count < len(input):
    subcount = 0
    dic = {}
    while (subcount < 500) and (count <len(input) ):

      outputs = torch.flatten(feature_extractor(input[count:count+1]),
      start_dim=2, end_dim=3)
      dic[count] = outputs
      count += 1
      subcount += 1
    torch.save(dic, 'HW/HW2/feature_extracted_%s_%s.pt'%(train_or_test, index))
    index += 1
feature_extraction(input_train, "train")
feature_extraction(input_test, "test")
train_feature =[]
```

```
for i in range(8):
  cur_feature =  list(torch.load("HW/HW2/feature_extracted_train_%s.pt"%(i)).values())
  train_feature.extend(cur_feature)
test_feature =[]
for i in range(8):
  cur_feature =  list(torch.load("HW/HW2/feature_extracted_test_%s.pt"%(i)).values())
  test_feature.extend(cur_feature)
train_feature_tensor = torch.stack(train_feature)
test_feature_tensor = torch.stack(test_feature)
print(train_feature_tensor.shape,test_feature_tensor.shape)
def flatten_L2_norm(feature_tensor):
  feature_tensor_flatten1 = torch.flatten(feature_tensor,start_dim=1, end_dim=2)
  feature_tensor_flatten = torch.flatten(feature_tensor_flatten1, start_dim = 1)
  feature_normalized = [feature_tensor_flatten[i].div(torch.norm(feature_tensor_flatten[i]
  , p=2).detach().expand_as(feature_tensor_flatten[i])) for i in
  range(len(feature_tensor_flatten))]
  feature_normalized_tensor = torch.stack(feature_normalized)
  return feature_normalized_tensor
train_feature_normalized_tensor = flatten_L2_norm(train_feature_tensor)
test_feature_normalized_tensor = flatten_L2_norm(test_feature_tensor)
dic_train = {}
for i in range(len(train_feature_normalized_tensor)):
  dic_train[i] = train_feature_normalized_tensor[i]
torch.save(dic_train, 'HW/HW2/normalized_train_feature_tensors.pt')
dic_test = {}
for i in range(len(test_feature_normalized_tensor)):
  dic_test[i] = test_feature_normalized_tensor[i]
torch.save(dic_test, 'HW/HW2/normalized_test_feature_tensors.pt')
train_x = torch.load('HW/HW2/normalized_train_feature_tensors.pt')
test_x = torch.load('HW/HW2/normalized_test_feature_tensors.pt')
train_y = pd.read_csv('HW/HW2/LabelImageTrain.csv')["Label"]
test_y = pd.read_csv('HW/HW2/LabelImageTest.csv')["Label"]
train_x =torch.stack(list(train_x.values())).detach().numpy()
test_x = torch.stack(list(test_x.values())).detach().numpy()
print(train_x.shape, test_x.shape, train_y.shape, test_y.shape)
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=42,penalty = 'l2',C = 530)
clf.fit(train_x,train_y)
print("The train set mean-per-class accuracy is %s"%(clf.score(train_x, train_y)))
print("The test set mean-per-class accuracy is %s"%(clf.score(test_x, test_y)))
```

## 0.3 Problem 3 - Training a Small CNN

### 0.3.1 Part 1

```
import torch
import torchvision
import torchvision.transforms as transforms
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='HW/HW2/data', train=True,
                                        download=True, transform=transform)
testset = torchvision.datasets.CIFAR10(root='HW/HW2/data', train=False,
                                       download=True, transform=transform)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
import torch.nn.functional as F
import torch
def mish(input):
    return input * torch.tanh(F.softplus(input))
from torch import nn
class Mish(nn.Module):
    def __init__(self):
        super().__init__()
    def forward(self, input):
        return mish(input)
torch.cuda.is_available()
def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')
device = get_default_device()
def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')
```

```
device = get_default_device()
def to_device(data, device):
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list,tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=(11, 11))
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=(3, 3))
        self.conv3 = nn.Conv2d(128, 128, kernel_size=(3, 3))
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc1 = nn.Linear(128, 10)
    def forward(self, x):
        x = self.maxpool(Mish()(self.conv1(x)))
        x = Mish()(self.conv2(x))
        x = Mish()(self.conv3(x))
        x = self.avgpool(x)
        x = x.view(-1, 128 * 1 * 1)
        x =  self.fc1(x)
        return x
import torch.optim as optim
criterion = nn.CrossEntropyLoss()
 def one_training_dev(train_ds, val_ds, batch_size, optimizer = "Adam",epoch_num = 50,
 lr = 1e-3):
  train_loss = []
  valid_loss = []
  train_loader = torch.utils.data.DataLoader(train_ds, batch_size, shuffle=True,
  num_workers=4, pin_memory=True)
  val_loader = torch.utils.data.DataLoader(val_ds, batch_size*2, num_workers=4,
  pin_memory=True)
  net = Net()
  to_device(net, device)
  #set up loss
  criterion = nn.CrossEntropyLoss()
  #set up optimizer
  if optimizer == "Adam":
    optimizer = optim.Adam(net.parameters(), lr = lr)
  elif optimizer == "Adadelta":
```

```
    optimizer = optim.Adadelta(net.parameters(), lr = lr)
  elif optimizer == "Adagrad":
    optimizer = optim.Adagrad(net.parameters(), lr = lr)
  elif optimizer == "AdamW":
    optimizer = optim.AdamW(net.parameters(), lr = lr)
  for epoch in range(epoch_num):  # loop over the dataset multiple times
      running_loss = 0.0
      for i, data in enumerate(train_loader, 0):

          inputs, labels = data
          inputs, labels =inputs.to(device),labels.to(device)
          optimizer.zero_grad()
          outputs = net(inputs)
          loss = criterion(outputs, labels)
          loss.backward()
          optimizer.step()
          running_loss += loss.item()
      running_loss /= len(train_ds)
      train_loss.append(running_loss)
      with torch.no_grad():
          loss_valid = 0.0
          for i, data in enumerate(val_loader, 0):
            inputs, labels = data
            inputs, labels =inputs.to(device),labels.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss_valid += loss.item()
          loss_valid /= len(val_ds)
          valid_loss.append(loss_valid)
  plt.figure(dpi = 300)
  plt.plot(np.arange(1,epoch_num +1),np.array(train_loss), label = "Training Loss" )
  plt.plot(np.arange(1,epoch_num +1),np.array(valid_loss), label = "Validating Loss" )
  plt.title("Training result with batch size = %s"%( batch_size))
  print('Training loss on last epoch:',train_loss[-1])
  print('Validating loss on last epoch:',valid_loss[-1])
  plt.legend()
  plt.show()
val_size = 10000 #20% as the val
train_size = len(trainset) - val_size
train_ds, val_ds = torch.utils.data.random_split(trainset, [train_size, val_size])
def tune_batch(batch_col ):
```

```
  print("Training with Adam parameters and epoch of 80, optimizer Adam, and
  learning rate of 1e-4")
  for i in batch_col:
    one_training_dev(train_ds, val_ds, i, optimizer = "Adam",
    epoch_num = 80, lr = 1e-4)
batch_col = [200,400,600, 800, 1000, 1500, 2000, 3000]
tune_batch(batch_col )
 def final_training_testing_evaluation(trainset, testset, batch_size,
 optimizer = "Adam",epoch_num = 500, lr = 1e-3):
  train_loss = []
  test_loss = []
  correct = 0
  total = 0
  trainloader = torch.utils.data.DataLoader(trainset, batch_size= batch_size,
                                            shuffle=True, num_workers=4)

  testloader = torch.utils.data.DataLoader(testset, batch_size= batch_size*2,
                                           shuffle=False, num_workers=4)
  net = Net()
  to_device(net, device)
  #set up loss
  criterion = nn.CrossEntropyLoss()#combines nn.LogSoftmax() and nn.NLLLoss()
  #set up optimizer
  if optimizer == "Adam":
    optimizer = optim.Adam(net.parameters(), lr = lr)
  for epoch in range(epoch_num):  # loop over the dataset multiple times
      running_loss = 0.0
      for i, data in enumerate(trainloader, 0):
          inputs, labels = data
          inputs, labels =inputs.to(device),labels.to(device)
          optimizer.zero_grad()
          outputs = net(inputs)
          loss = criterion(outputs, labels)
          loss.backward()
          optimizer.step()
          running_loss += loss.item()
      running_loss /= len(trainset)
      train_loss.append(running_loss)
      with torch.no_grad():
          running_loss = 0.0
          for i, data in enumerate(testloader, 0):
```

```
            inputs, labels = data
            inputs, labels =inputs.to(device),labels.to(device)
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            loss = criterion(outputs, labels)
            running_loss += loss.item()
          running_loss /= len(testset)
          test_loss.append(running_loss)

  plt.figure(dpi = 300)
  plt.plot(np.arange(1,epoch_num +1),np.array(train_loss), label = "Training Loss" )
  plt.plot(np.arange(1,epoch_num +1),np.array(test_loss), label = "Testing Loss" )
  plt.title("Best Training and Testing result for tuned hyper-parameters")
  print('Training loss on last epoch:',train_loss[-1])
  print('Testing loss on last epoch:',test_loss[-1])
  print('Accuracy of the network on the test set: %d %%' % (
    100 * correct / total))
  plt.legend()
  plt.show()
  return net,train_loss, test_loss
best_model,train_loss, test_loss = final_training_testing_evaluation(trainset,
testset, 12000, optimizer = "Adam",epoch_num = 500, lr = 1e-3)
parameters = []
layers = []
model_component = list(best_model.children())
count_conv = 0
for i in range(len(model_component)):
  if type(model_component[i]) == nn.Conv2d:
    count_conv += 1
    parameters.append(model_component[i].weight)
    layers.append(model_component[i]) #only need the first convolutional layer
    break
def visualize_conv_layer(parameters):
  # plt.figure(figsize=(20, 17))
  all = []
  visualize = []
  for i, filter in enumerate(parameters[0]):
      tensor_col = filter.cpu()[0, :, :].detach()
      tensor_col1 = tensor_col - torch.min(tensor_col, 0,keepdim=True)[0]
```

```
      tensor_col2 =  tensor_col1 / torch.max(tensor_col1, 0,keepdim=True)[0]
      all.append(tensor_col2)
  fig=plt.figure(dpi = 400, figsize = (10,3))
  for i in range(64):
    ax=fig.add_subplot(4,16,i+1)
    ax.imshow(all[i])
    ax.axis('off')
  fig.suptitle("64 feature extractors in first convolutional layer",fontsize = 14)
  plt.show()
visualize_conv_layer(parameters)
```

### 0.3.2 Part 2

```
class NetBatchNorm(nn.Module):
    def __init__(self):
        super(NetBatchNorm, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=(11, 11))
        self.conv1_bn=nn.BatchNorm2d(64)
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=(3, 3))
        self.conv2_bn=nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 128, kernel_size=(3, 3))
        self.conv3_bn=nn.BatchNorm2d(128)
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc1 = nn.Linear(128, 10)
    def forward(self, x):
        x = self.maxpool(self.conv1_bn(Mish()(self.conv1(x))))
        x = self.conv2_bn(Mish()(self.conv2(x)))
        x = self.conv3_bn(Mish()(self.conv3(x)))
        x = self.avgpool(x)
        x = x.view(-1, 128 * 1 * 1)
        x =  self.fc1(x)
        return x
 def final_training_batchnorm_testing_evaluation(trainset, testset, batch_size,
 optimizer = "Adam",epoch_num = 500, lr = 1e-3):
  train_loss = []
  test_loss = []
  correct = 0
  total = 0
  trainloader = torch.utils.data.DataLoader(trainset, batch_size= batch_size,
                                            shuffle=True, num_workers=0)
```

```
testloader = torch.utils.data.DataLoader(testset, batch_size= batch_size*2,
                                         shuffle=False, num_workers=0)
net = NetBatchNorm()
to_device(net, device)
#set up loss
criterion = nn.CrossEntropyLoss()
if optimizer == "Adam":
  optimizer = optim.Adam(net.parameters(), lr = lr)
for epoch in range(epoch_num):

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):

        inputs, labels = data
        inputs, labels =inputs.to(device),labels.to(device)
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    running_loss /= len(trainset)
    train_loss.append(running_loss)
    with torch.no_grad():
        running_loss = 0.0
        for i, data in enumerate(testloader, 0):
          inputs, labels = data
          inputs, labels =inputs.to(device),labels.to(device)
          outputs = net(inputs)
          _, predicted = torch.max(outputs.data, 1)
          total += labels.size(0)
          correct += (predicted == labels).sum().item()
          loss = criterion(outputs, labels)
          running_loss += loss.item()
        running_loss /= len(testset)
        test_loss.append(running_loss)
plt.figure(dpi = 300)
plt.plot(np.arange(1,epoch_num +1),np.array(train_loss), label = "Training Loss" )
plt.plot(np.arange(1,epoch_num +1),np.array(test_loss), label = "Testing Loss" )
plt.title("Best Training and Testing result for tuned hyper-parameters")
```

```
  print('Training loss on last epoch:',train_loss[-1])
  print('Testing loss on last epoch:',test_loss[-1])
  print('Accuracy of the network on the test set: %d %%' % (
    100 * correct / total))
  plt.legend()
  plt.show()
  return net,train_loss, test_loss
best_batchnorm_model,train_batchnorm_loss, test_batchnorm_loss =
final_training_batchnorm_testing_evaluation(trainset, testset, 12000,
optimizer = "Adam",epoch_num = 500, lr = 1e-3)
parameters_batchnorm = []
layers_batchnorm = []
model_component = list(best_batchnorm_model.children())
count_conv = 0
for i in range(len(model_component)):
  if type(model_component[i]) == nn.Conv2d:
    count_conv += 1
    parameters_batchnorm.append(model_component[i].weight)
    layers_batchnorm.append(model_component[i])
    break
visualize_conv_layer(parameters_batchnorm)
plt.figure(dpi = 300)
epoch_num = 500
plt.plot(np.arange(1,epoch_num +1), train_batchnorm_loss,
label = "Training Loss with Batch Norm")
plt.plot(np.arange(1,epoch_num +1),train_loss, label = "Training Loss")
plt.title("Training loss with and withour Batch Normalization in hidden layers")
plt.xlabel("Epoch")
plt.ylabel("Cross Entropy loss")
plt.show
```

### 0.3.3 Part 3

```
class NetOptimize(nn.Module):
    def __init__(self):
        super(NetOptimize, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=(3,3))
        self.conv1_bn=nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 128,kernel_size=(3,3))
        self.conv2_bn=nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128,128, kernel_size=(3,3))
```

```
        self.conv3_bn=nn.BatchNorm2d(128)
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=(3, 3))
        self.conv4_bn=nn.BatchNorm2d(256)

        self.conv5 = nn.Conv2d(256, 512, kernel_size=(3, 3))
        self.conv5_bn=nn.BatchNorm2d(512)
        self.conv6 = nn.Conv2d(512, 512, kernel_size=(3, 3))
        self.conv6_bn=nn.BatchNorm2d(512)

        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc1 = nn.Linear(512, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)
        self.dropout = nn.Dropout(0.5)
        # torch.cuda.empty_cache()

    def forward(self, x):
        x = self.conv1_bn(Mish()(self.conv1(x)))
        x = self.conv2_bn(Mish()(self.conv2(x)))
        x = self.maxpool(self.conv3_bn(Mish()(self.conv3(x))))

        x = self.dropout(self.conv4_bn(Mish()(self.conv4(x))))
        x = self.dropout(self.conv5_bn(Mish()(self.conv5(x))))
        x = self.conv6_bn(Mish()(self.conv6(x)))
        x = self.conv6_bn(Mish()(self.conv6(x)))
        x = self.conv6_bn(Mish()(self.conv6(x)))

        x = self.avgpool(x)
        x = x.view(-1, 512 * 1 * 1)
        x = self.dropout(self.fc1(x))
        x = self.dropout(self.fc2(x))
        x =  self.fc3(x)
        return x
def one_training_dev_optimize(train_ds, val_ds, batch_size, optimizer = "Adam",
epoch_num = 50, lr = 1e-3):
  torch.cuda.empty_cache()
  train_loss = []
  valid_loss = []
  train_loader = torch.utils.data.DataLoader(train_ds, batch_size, shuffle=True,
  num_workers=4, pin_memory=True)
```

```
val_loader = torch.utils.data.DataLoader(val_ds, batch_size*2, num_workers=4,
pin_memory=True)
net = NetVGGModify()
to_device(net, device)
criterion = nn.CrossEntropyLoss()
if optimizer == "Adam":
  optimizer = optim.Adam(net.parameters(), lr = lr)
elif optimizer == "Adadelta":
  optimizer = optim.Adadelta(net.parameters(), lr = lr)
elif optimizer == "Adagrad":
  optimizer = optim.Adagrad(net.parameters(), lr = lr)
elif optimizer == "AdamW":
  optimizer = optim.AdamW(net.parameters(), lr = lr)
for epoch in range(epoch_num):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        inputs, labels =inputs.to(device),labels.to(device)
        optimizer.zero_grad()
        outputs = net(inputs)
        inputs.detach()
        labels.detach()
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    running_loss /= len(train_ds)
    train_loss.append(running_loss)
    with torch.no_grad():
        loss_valid = 0.0
        for i, data in enumerate(val_loader, 0):
          inputs, labels = data
          inputs, labels =inputs.to(device),labels.to(device)
          outputs = net(inputs)
          loss = criterion(outputs, labels)
          inputs.detach()
          labels.detach()
          loss_valid += loss.item()
        loss_valid /= len(val_ds)
        valid_loss.append(loss_valid)
plt.figure(dpi = 300)
```

```
  plt.plot(np.arange(1,epoch_num +1),np.array(train_loss), label = "Training Loss" )
  plt.plot(np.arange(1,epoch_num +1),np.array(valid_loss), label = "Validating Loss" )
  plt.title("Training result with batch size = %s"%( batch_size))
  print('Training loss on last epoch:',train_loss[-1])
  print('Validating loss on last epoch:',valid_loss[-1])
  plt.legend()
  plt.show()
def tune_batch_optimize(batch_col ):
  for i in batch_col:
    one_training_dev_optimize(train_ds, val_ds, i, optimizer = "Adam",epoch_num = 50,
    lr = 1e-4)
tune_batch_optimize(batch_col )
 def final_training_batchnorm_testing_evaluation(trainset, testset, batch_size,
 optimizer = "Adam",epoch_num = 500, lr = 1e-3):
  train_loss = []
  test_loss = []
  correct = 0
  total = 0
  trainloader = torch.utils.data.DataLoader(trainset, batch_size= batch_size,
                                            shuffle=True, num_workers=4)

  testloader = torch.utils.data.DataLoader(testset, batch_size= batch_size*2,
                                           shuffle=False, num_workers=4)
  net = NetVGGModify()
  to_device(net, device)
  #set up loss
  criterion = nn.CrossEntropyLoss()
  if optimizer == "Adam":
    optimizer = optim.Adam(net.parameters(), lr = lr)
  for epoch in range(epoch_num):  # loop over the dataset multiple times
      running_loss = 0.0
      for i, data in enumerate(trainloader, 0):
          inputs, labels = data
          inputs, labels =inputs.to(device),labels.to(device)
          optimizer.zero_grad()
          outputs = net(inputs)
          loss = criterion(outputs, labels)
          loss.backward()
          optimizer.step()
          running_loss += loss.item()
      running_loss /= len(trainset)
```

```
      train_loss.append(running_loss)
      with torch.no_grad():
          running_loss = 0.0
          for i, data in enumerate(testloader, 0):
            inputs, labels = data
            inputs, labels =inputs.to(device),labels.to(device)
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            loss = criterion(outputs, labels)
            running_loss += loss.item()
          running_loss /= len(testset)
          test_loss.append(running_loss)
  plt.figure(dpi = 300)
  plt.plot(np.arange(1,epoch_num +1),np.array(train_loss), label = "Training Loss" )
  plt.plot(np.arange(1,epoch_num +1),np.array(test_loss), label = "Testing Loss" )
  plt.title("Best Training and Testing result for tuned hyper-parameters")
  print('Training loss on last epoch:',train_loss[-1])
  print('Testing loss on last epoch:',test_loss[-1])
  print('Accuracy of the network on the test set: %d %%' % (
    100 * correct / total))
  plt.legend()
  plt.show()
  return net,train_loss, test_loss
net,train_loss, test_loss =
final_training_batchnorm_testing_evaluation(trainset, testset, 1000,
optimizer = "Adam",epoch_num = 50, lr = 1e-4)
```

## 0.4 Problem 4

### 0.4.1 Part 1

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

```
import pandas as pd
from PIL import Image
import ast
import torch.nn.functional as F
resnet18 = models.resnet18(pretrained=True)
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]
 )])
with open('imagenet_classes.txt') as f:
  file = open("imagenet_classes.txt", "r")
  contents = file.read()
  dictionary = ast.literal_eval(contents)
file.close()
classes = list(dictionary.values())
def original_class_attain(pre_trained_model, image,classes):
  batch_t = torch.unsqueeze(image, 0)
  pre_trained_model.eval()
  out = pre_trained_model(batch_t)
  _, index = torch.max(out, 1)
  print(_, index)
  probability = torch.nn.functional.softmax(out, dim=1)[0]
  class_assigned = classes[index]
  return classes[index[0]], probability[index[0]].item()
import matplotlib.pyplot as plt
def image_show(input_image):
  img = input_image.numpy().transpose(1, 2, 0)
  plt.imshow(img)
  plt.show()
img1 = transform(Image.open('peppers.jpg'))
img2 = transform(Image.open('hotdog.jpg'))
img3 = transform(Image.open('burger.jpg'))
img4 = transform(Image.open('lemon.jpg'))
x_data = [img1.numpy(),img2.numpy(),img3.numpy(),img4.numpy()]
labels = [812,1,2,3]
train_data = []
for i in range(len(x_data)):
```

```
   train_data.append([x_data[i], labels[i]])
trainloader = torch.utils.data.DataLoader(train_data, shuffle=False, batch_size=4)
prev_index = [945,934,933,951]
class_assigned, probability  = original_class_attain(resnet18, img1,classes)
print("Assigned class is %s, with probability of %s"%(class_assigned, probability ))
image_show(img1)
class_assigned, probability  = original_class_attain(resnet18, img2,classes)
print("Assigned class is %s, with probability of %s"%(class_assigned, probability ))
image_show(img2)
class_assigned, probability  = original_class_attain(resnet18, img3,classes)
print("Assigned class is %s, with probability of %s"%(class_assigned, probability ))
image_show(img3)
class_assigned, probability  = original_class_attain(resnet18, img4,classes)
print("Assigned class is %s, with probability of %s"%(class_assigned, probability ))
image_show(img4)
def adversarial_attack(image,original_image,  lamda,epsilon, data_grad):
    perturbed_image = image - lamda*data_grad
    clipped_change =torch.clamp(perturbed_image-original_image, -epsilon,  epsilon)
    new_perturbed_image = original_image + clipped_change
    return new_perturbed_image
def train_four(model,   device, trainloader,  lamda,epsilon ):
    for data, target in trainloader:
        data.requires_grad = True
        output = model(data)
        pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability
        perturbed_data = data.clone()
        while (pred.flatten().cpu().numpy()[0] != target.cpu().numpy()[0])
        or (pred.flatten().cpu().numpy()[1] == prev_index[1])
        or (pred.flatten().cpu().numpy()[2] == prev_index[2])
        or  (pred.flatten().cpu().numpy()[3] == prev_index[3]):
          loss = F.nll_loss(output, target)
          model.zero_grad()
          loss.backward(retain_graph=True)
          data_grad = data.grad.data
          # print(data_grad)
          perturbed_data =  adversarial_attack(perturbed_data,data,
          lamda,epsilon, data_grad)
          # print(perturbed_data)
          output = model(perturbed_data)
          pred = output.max(1, keepdim=True)[1]
          print(pred.flatten())
```

```
        return perturbed_data
model = models.resnet18(pretrained=True)
use_cuda=True
device = torch.device("cuda" if (use_cuda and torch.cuda.is_available()) else "cpu")
perturbed_data= train_four(model,'cuda', trainloader, 0.01, 1)
output = model(perturbed_data)
pred = output.max(1, keepdim=True)[1]
probability = torch.nn.Softmax(dim=1)(output).max(1, keepdim=True)[0]
perturbed_idx = pred.flatten().detach().numpy()
perturbed_probability = probability.flatten().detach().numpy()
print(perturbed_idx ,perturbed_probability)
img_col = [img1,img2,img3, img4]
def plot(x_data,perturbed_data,img_col,perturbed_idx ,perturbed_probability):
  fig, axs = plt.subplots(4,3,dpi = 400, figsize = (10,10))
  [axi.set_axis_off() for axi in axs.ravel()]
  fig.suptitle('Examples on adversarial noise in fooling NN classifier(ResNet18)')
  for i in range(4):
    class_assigned, probability  = original_class_attain(resnet18, img_col[i],classes)
    axs[i, 0].set_title("Assigned class is %s \n with probability of %s"%
    (class_assigned, probability ),fontdict = {'fontsize': 5} )
    axs[i][0].imshow(x_data[i].transpose(1, 2, 0))
    axs[i, 1].set_title("adversarial noise" )
    axs[i][1].imshow((perturbed_data.detach().numpy()[i] - x_data[i]).transpose(1, 2, 0))
    axs[i, 2].set_title("Assigned class is %s \n with probability of %s"%
    (classes[perturbed_idx[i]], perturbed_probability[i] ),fontdict = {'fontsize': 5} )
    axs[i][2].imshow(perturbed_data.detach().numpy()[i].transpose(1, 2, 0))
plot(x_data,perturbed_data,img_col,perturbed_idx ,perturbed_probability)
```

### 0.4.2 Part 2

```
def reflection(data):
  return torch.flip(data, [3])
def crop(data):
  transform = transforms.Compose([transforms.CenterCrop(0.895*data.shape[2])])
  return transform(data)
def rotation(data):
  # transform = transforms.functional.rotate(data,30)
  return transforms.functional.rotate(data,30)
def grayscaled(data):
  transform = transforms.Grayscale(num_output_channels=3)
  return transform(data)
```

```
def plot_friction(perturbed_data,perturbed_idx,perturbed_probability):
  flipped = reflection(perturbed_data)
  cropped = crop(perturbed_data)
  rotated = rotation(perturbed_data)
  model = models.resnet18(pretrained=True)
  grayscale = grayscaled(perturbed_data)
  fig, axs = plt.subplots(4,5,dpi = 400, figsize = (10,10))
  [axi.set_axis_off() for axi in axs.ravel()]
  fig.suptitle('Predicted class and probability with different friction on
  adversarial image')
  for i in range(4):
    axs[i, 0].set_title("Assigned class is %s \n with probability of %s"%
    (classes[perturbed_idx[i]], perturbed_probability[i] ),fontdict = {'fontsize': 5} )
    axs[i][0].imshow(perturbed_data.detach().numpy()[i].transpose(1, 2, 0))

    cur_class, cur_prob = original_class_attain(model, flipped[i],classes)
    axs[i, 1].set_title("Assigned class is %s \n with probability of %s"%
    (cur_class, cur_prob),fontdict = {'fontsize': 5} )
    axs[i][1].imshow(flipped.detach().numpy()[i].transpose(1, 2, 0))

    cur_class, cur_prob = original_class_attain(model, cropped[i],classes)
    axs[i, 2].set_title("Assigned class is %s \n with probability of %s"%
    (cur_class, cur_prob),fontdict = {'fontsize': 5} )
    axs[i][2].imshow(cropped.detach().numpy()[i].transpose(1, 2, 0))

    cur_class, cur_prob = original_class_attain(model, rotated[i],classes)
    axs[i, 3].set_title("Assigned class is %s \n with probability of %s"%
    (cur_class, cur_prob),fontdict = {'fontsize': 5} )
    axs[i][3].imshow(rotated.detach().numpy()[i].transpose(1, 2, 0))

    cur_class, cur_prob = original_class_attain(model, grayscale[i],classes)
    axs[i, 4].set_title("Assigned class is %s \n with probability of %s"%
    (cur_class, cur_prob),fontdict = {'fontsize': 5} )
    axs[i][4].imshow(grayscale.detach().numpy()[i].transpose(1, 2, 0))
plot_friction(perturbed_data,perturbed_idx,perturbed_probability)
```

## References


[1] URL: https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a.

[2] URL: https://github.com/digantamisra98/Mish/tree/master/Mish/Torch.

[3] URL: https://medium.com/analytics-vidhya/training-deep-neural-networks-on-a-gpu-with-pytorch-2851ccfb6066.
[4] URL: https://blog.csdn.net/ys1305/article/details/94332007.
[5] URL: https://discuss.pytorch.org/t/how-are-layer-weights-and-biases-initialized-by-default/13073.