Convolution Neural Network

The goal of this assignment is to get hands-on experience designing and training deep convolutional neural networks using PyTorch. Starting from a baseline architecture we provided, you will design an improved deep net architecture to classify (small) images into 10 construction equipment categories. You will evaluate the performance of your architecture by uploading your predictions to this Kaggle competition and submit your code and report describing your implementation choices to NTU COOL.

Google Colab setup with Google Drive folder

This notebook provides the code you need to set up Google Colab to run and import files from within a Google Drive folder.

This will allow you to upload assignment code to your Google Drive and then run the code on Google Colab machines (with free GPUs if needed).

You will need to create a folder in your Google Drive to hold your assignments and you will need to open Colaboratory within this folder before running the set up code (check the link above to see how).

Mount Google Drive

This will allow the Colab machine to access Google Drive folders by mounting the drive on the machine. You will be asked to copy and paste an authentication code.

```
#from google.colab import drive
#drive.mount('/content/gdrive/')
# !gdown --id '1Ln9gskUvFdg3Y27lmPHPelbGRUleSjCs' --output train
# !gdown --id '15e6gxX3FRIQ53e_Zsq9iGp8U8kkXvMZm' --output val
# !gdown --id '1MELlUwxd24iH1wY7TjgWigHFXhSZh5xd' --output test
# Dropbox
# !wget https://www.dropbox.com/s/qk9av5laeu817nf/train?dl=0 -O train
# !wget https://www.dropbox.com/s/z1gm6w8wtiwx3xs/val?dl=0 -O val
# !wget https://www.dropbox.com/s/wkcsa8ah5gtg7uq/test?dl=0 -O test
```

Change directory to allow imports

As noted above, you should create a Google Drive folder to hold all your assignment files. You will need to add this code to the top of any python notebook you run to be able to import python files from your drive assignment folder (you should change the file path below to be your own assignment folder).

Copy data to local dir

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
ls
```

```
gdrive/  sample_data/
```

```
import os
os.chdir("/content/gdrive/My Drive/Colab Notebooks/0520/")
```

```
#!mkdir /data
#!cp train /data/
#!cp val /data/
#!cp test /data/
```

```
ls ./data
```

```
test  train  val
```

Set up GPU and PyTorch

First, ensure that your notebook on Colaboratory is set up to use GPU. After opening the notebook on Colaboratory, go to Edit>Notebook settings, select Python 3 under "Runtime type," select GPU under "Hardware accelerator," and save.

Next, install PyTorch:

```
!pip3 install torch torchvision
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-packages (1.
Requirement already satisfied: torchvision in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-
Requirement already satisfied: pillow>=4.1.1 in /usr/local/lib/python3.7/dist-pack
```

Make sure that pytorch is installed and works with GPU:

```
import torch
a = torch.Tensor([1]).cuda()
print(a)
!nvidia-smi
```

```
tensor([1.], device='cuda:0')
Wed May 26 14:32:39 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 465.19.01    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
```

```
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   42C    P0    25W /  70W |   1054MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

Part 1

```
"""Headers"""

from __future__ import print_function
from PIL import Image
import os
import os.path
import numpy as np
import sys
import cv2
if sys.version_info[0] == 2:
    import cPickle as pickle
else:
    import pickle

import torch.utils.data as data
from torchvision.datasets.utils import download_url, check_integrity

import csv
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import os.path
import sys
import torch
import torch.utils.data
import torchvision
import torchvision.transforms as transforms

from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
from tqdm.auto import tqdm
np.random.seed(111)
torch.cuda.manual_seed_all(111)
torch.manual_seed(111)

import IPython.display
from google.colab.patches import cv2_imshow
```

Just execute the cell below. This is the dataloader. DO NOT CHANGE ANYTHING IN HERE!

```
"""""""
class ACID100_CIE5141(data.Dataset):
    """`ACID <https://www.acidb.ca/dataset>`_ Dataset.
    Randomly pick 1000 train, val images and 200 test images for each class that only c
    code adapted from CIFAR 100

    Args:
        root (string): Root directory of dataset where directory
            ``cifar-10-batches-py`` exists or will be saved to if download is set to Tr
        train (bool, optional): If True, creates dataset from training set, otherwise
            creates from test set.
        transform (callable, optional): A function/transform that  takes in an PIL imag
            and returns a transformed version. E.g, ``transforms.RandomCrop``
        target_transform (callable, optional): A function/transform that takes in the
            target and transforms it.
        download (bool, optional): If true, downloads the dataset from the internet and
            puts it in root directory. If dataset is already downloaded, it is not
            downloaded again.

    """

    def __init__(self, root, fold="train",
                 transform=None, target_transform=None,
                 download=False):

        fold = fold.lower()

        self.train = False
        self.test = False
        self.val = False

        if fold == "train":
            self.train = True
        elif fold == "test":
            self.test = True
        elif fold == "val":
            self.val = True
        else:
            raise RuntimeError("Not train-val-test")


        self.root = os.path.expanduser(root)
        self.transform = transform
        self.target_transform = target_transform

        # fpath = os.path.join(root, self.filename)
        # if not self._check_integrity():
        #     raise RuntimeError('Dataset not found or corrupted.' +
        #                        ' Download it and extract the file again.')

        # now load the picked numpy arrays
```

```
        if self.train:
            self.train_data = []
            self.train_labels = []
            file = os.path.join(self.root, 'train')
            print(file)
            fo = open(file, 'rb')
            if sys.version_info[0] == 2:
                entry = pickle.load(fo)
            else:
                entry = pickle.load(fo, encoding='latin1')

            #print(entry['images'][0].shape)
            #image = cv2.cvtColor(entry['images'][0], cv2.COLOR_BGR2RGB)
            #print(image)
            #cv2_imshow(image)
            #plt.imshow(image)
            #plt.show()

            self.train_data.append(entry['images'])
            # make labels from 0-9
            self.train_labels = [x-1 for x in entry['labels']]
            fo.close()

            self.train_data = np.concatenate(self.train_data)
            #print(self.train_data.shape)
            self.train_data = self.train_data.reshape((1000, 3, 256, 256))
            self.train_data = self.train_data.transpose((0, 2, 3, 1))  # convert to HWC
            #print(self.train_data.shape)

            p = np.arange(0,1000,1)
            mask_train = np.ones((1000,), dtype=bool)
            mask_train[p] = True

            copy_all_data = np.array(self.train_data)
            self.train_data = np.array(copy_all_data[mask_train])

            copy_all_labels = np.array(self.train_labels)
            self.train_labels = np.array(copy_all_labels[mask_train])

        elif self.val:
            self.val_data = []
            self.val_labels = []
            file = os.path.join(self.root, 'val')
            fo = open(file, 'rb')
            if sys.version_info[0] == 2:
                entry = pickle.load(fo)
            else:
                entry = pickle.load(fo, encoding='latin1')

            self.val_data.append(entry['images'])
            # make labels from 0-9
            self.val_labels = [x-1 for x in entry['labels']]
            fo.close()

            self.val_data = np.concatenate(self.val_data)
            self.val_data = self.val_data.reshape((1000, 3, 256, 256))
            self.val_data = self.val_data.transpose((0, 2, 3, 1))  # convert to HWC

            p = np.arange(0,1000,1)
```

```
            p = np.arange(0,1000,1)
            mask_val = np.ones((1000,), dtype=bool)
            mask_val[p] = True

            copy_all_data = np.array(self.val_data)
            self.val_data = np.array(copy_all_data[mask_val])

            copy_all_labels = np.array(self.val_labels)
            self.val_labels = np.array(copy_all_labels[mask_val])

        elif self.test:
            # f = self.test_list[0][0]
            file = os.path.join(self.root, 'test')
            fo = open(file, 'rb')
            if sys.version_info[0] == 2:
                entry = pickle.load(fo)
            else:
                entry = pickle.load(fo, encoding='latin1')
            self.test_data = entry['images']
            # make labels from 0-9
            self.test_labels = [x-1 for x in entry['labels']]

            fo.close()
            self.test_data = self.test_data.reshape((200, 3, 256, 256))
            self.test_data = self.test_data.transpose((0, 2, 3, 1))  # convert to HWC

    def __getitem__(self, index):
        """
        Args:
            index (int): Index

        Returns:
            tuple: (image, target) where target is index of the target class.
        """
        if self.train:
            img, target = self.train_data[index], self.train_labels[index]
        elif self.test:
            img, target = self.test_data[index], self.test_labels[index]
        elif self.val:
            img, target = self.val_data[index], self.val_labels[index]

        # doing this so that it is consistent with all other datasets
        # to return a PIL Image
        img = Image.fromarray(cv2.cvtColor(img,cv2.COLOR_BGR2RGB))

        if self.transform is not None:
            img = self.transform(img)

        if self.target_transform is not None:
            target = self.target_transform(target)

        return img, target

    def __len__(self):
        if self.train:
            return len(self.train_data)
        elif self.test:
            return len(self.test_data)
        elif self.val:
```

```
        elif self.val:
            return len(self.val_data)

    def _check_integrity(self):
        root = self.root
        for fentry in (self.train_list + self.test_list):
            filename, md5 = fentry[0], fentry[1]
            fpath = os.path.join(root, self.base_folder, filename)
            if not check_integrity(fpath, md5):
                return False
        return True

    def __repr__(self):
        fmt_str = 'Dataset ' + self.__class__.__name__ + '\n'
        fmt_str += '    Number of datapoints: {}\n'.format(self.__len__())
        tmp = 'train' if self.train is True else 'test'
        fmt_str += '    Split: {}\n'.format(tmp)
        fmt_str += '    Root Location: {}\n'.format(self.root)
        tmp = '    Transforms (if any): '
        fmt_str += '{0}{1}\n'.format(tmp, self.transform.__repr__().replace('\n', '\n'
        tmp = '    Target Transforms (if any): '
        fmt_str += '{0}{1}'.format(tmp, self.target_transform.__repr__().replace('\n',
        return fmt_str
```

This file has been adapted from the easy-to-use tutorial released by PyTorch:
http://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

Training an image classifier

We will do the following steps in order:

1. Load the CIFAR100_CS543 training, validation and test datasets using torchvision. Use torchvision.transforms to apply transforms on the dataset.
2. Define a Convolution Neural Network - BaseNet
3. Define a loss function and optimizer
4. Train the network on training data and check performance on val set. Plot train loss and validation accuracies.
5. Try the network on test data and create .csv file for submission to kaggle

```
# <<TODO#5>> Based on the val set performance, decide how many
# epochs are apt for your model.
# ---------
EPOCHS = 200
# ---------

IS_GPU = True
TEST_BS = 256
TOTAL_CLASSES = 10
TRAIN_BS = 16
PATH_TO_ACID100_CIE5141 = "./data/"

# Transform:
# Stats needed to normalize images
input_size = 128
```

```
means = [0.485, 0.456, 0.406]
std_devs = [0.229, 0.224, 0.225]

# Other transforms parameters
rotation = 30
p=0.5
```

```
ls /data/
```

```
ls: cannot access '/data/': No such file or directory
```

```
def calculate_val_accuracy(valloader, is_gpu):
    """ Util function to calculate val set accuracy,
    both overall and per class accuracy
    Args:
        valloader (torch.utils.data.DataLoader): val set
        is_gpu (bool): whether to run on GPU
    Returns:
        tuple: (overall accuracy, class level accuracy)
    """
    correct = 0.
    total = 0.
    predictions = []

    class_correct = list(0. for i in range(TOTAL_CLASSES))
    class_total = list(0. for i in range(TOTAL_CLASSES))

    for data in valloader:
        images, labels = data
        if is_gpu:
            images = images.cuda()
            labels = labels.cuda()
        outputs = net(Variable(images))
        _, predicted = torch.max(outputs.data, 1)
        predictions.extend(list(predicted.cpu().numpy()))
        total += labels.size(0)
        correct += (predicted == labels).sum()

        c = (predicted == labels).squeeze()
        for i in range(len(labels)):
            label = labels[i]
            class_correct[label] += c[i]
            class_total[label] += 1

    class_accuracy = 100 * np.divide(class_correct, class_total)
    return 100*correct/total, class_accuracy
```

1.** Loading CIFAR100_CS543**

We modify the dataset to create CIFAR100_CS543 dataset which consist of 45000 training images (450 of each class), 5000 validation images (50 of each class) and 10000 test images (100 of each class). The train and val datasets have labels while all the labels in the test set are set to 0.

```
# The output of torchvision datasets are PILImage images of range [0, 1].
# Using transforms.ToTensor(), transform them to Tensors of normalized range
```

```
# [-1, 1].


# <<TODO#1>> Use transforms.Normalize() with the right parameters to
# make the data well conditioned (zero mean, std dev=1) for improved training.
# <<TODO#2>> Try using transforms.RandomCrop() and/or transforms.RandomHorizontalFlip()
# to augment training data.
# After your edits, make sure that test_transform should have the same data
# normalization parameters as train_transform
# You shouldn't have any data augmentation in test_transform (val or test data is never
# ----------------------

train_transform = transforms.Compose([
    transforms.RandomResizedCrop(input_size),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(rotation),
    transforms.RandomGrayscale(p),
    transforms.ToTensor(),
    transforms.Normalize(means, std_devs)])

test_transform = transforms.Compose([
    transforms.Resize(input_size),
    transforms.ToTensor(),
    transforms.Normalize(means, std_devs)])
# ----------------------

trainset = ACID100_CIE5141(root=PATH_TO_ACID100_CIE5141, fold="train",
                                        download=True, transform=train_transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=TRAIN_BS,
                                          shuffle=True, num_workers=2)
print("Train set size: "+str(len(trainset)))

valset = ACID100_CIE5141(root=PATH_TO_ACID100_CIE5141, fold="val",
                                       download=True, transform=test_transform)
valloader = torch.utils.data.DataLoader(valset, batch_size=TEST_BS,
                                          shuffle=False, num_workers=2)
print("Val set size: "+str(len(valset)))

testset = ACID100_CIE5141(root=PATH_TO_ACID100_CIE5141, fold="test",
                                       download=True, transform=test_transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=TEST_BS,
                                          shuffle=False, num_workers=2)
print("Test set size: "+str(len(testset)))

# The 100 classes for CIFAR100
classes =['backhoe_loader', 'excavator', 'dump_truck', 'concrete_mixer_truck', 'mobile_
```

```
./data/train
Train set size: 1000
Val set size: 1000
Test set size: 200
```

```
########################################################################
# 2. Define a Convolution Neural Network
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# We provide a basic network that you should understand, run and
# eventually improve
# <<TODO>> Add more conv layers
```

```
    # <<TODO>> Add more fully connected (fc) layers
    # <<TODO>> Add regularization layers like Batchnorm.
    #          nn.BatchNorm2d after conv layers:
    #          http://pytorch.org/docs/master/nn.html#batchnorm2d
    #          nn.BatchNorm1d after fc layers:
    #          http://pytorch.org/docs/master/nn.html#batchnorm1d
    # This is a good resource for developing a CNN for classification:
    # http://cs231n.github.io/convolutional-networks/#layers

    import math
    import torch.nn as nn
    import torch.nn.functional as F

    class BaseNet(nn.Module):
        def __init__(self, img_rows = input_size):
            super(BaseNet, self).__init__()

            # <<TODO#3>> Add more conv layers with increasing
            # output channels
            # <<TODO#4>> Add normalization layers after conv
            # layers (nn.BatchNorm2d)

            # Also experiment with kernel size in conv2d layers (say 3
            # inspired from VGGNet)
            # To keep it simple, keep the same kernel size
            # (right now set to 5) in all conv layers.
            # Do not have a maxpool layer after every conv layer in your
            # deeper network as it leads to too much loss of information.

            # self.conv1 = nn.Conv2d(3, 6, 5,1,2)
            # self.pool = nn.MaxPool2d(2, 2)
            # self.conv2 = nn.Conv2d(6, 16, 5,1,2)

            self.out_rows = int(img_rows / 16)
            self.cnn = nn.Sequential(
                nn.Conv2d(3, 16, 3, 1, 1),
                nn.BatchNorm2d(16),
                nn.ReLU(),
                nn.MaxPool2d(2, 2, 0),

                nn.Conv2d(16, 32, 3, 1, 1),
                nn.BatchNorm2d(32),
                nn.ReLU(),
                nn.MaxPool2d(2, 2, 0),

                nn.Conv2d(32, 64, 3, 1, 1),
                nn.BatchNorm2d(64),
                nn.ReLU(),
                nn.MaxPool2d(2, 2, 0),

                nn.Conv2d(64, 128, 3, 1, 1),
                nn.BatchNorm2d(128),
                nn.ReLU(),
                nn.MaxPool2d(2, 2, 0),
            )

            # <<TODO#3>> Add more linear (fc) layers
            # <<TODO#4>> Add normalization layers after linear and
```

```
        # experiment inserting them before or after ReLU (nn.BatchNorm1d)
        # More on nn.sequential:
        # http://pytorch.org/docs/master/nn.html#torch.nn.Sequential

        self.fc_net = nn.Sequential(
            nn.Linear(128 * self.out_rows * self.out_rows, 256),
            nn.ReLU(),
            #nn.Dropout(0.5),
            nn.Linear(256, TOTAL_CLASSES),

        )
        self._initialize_weights()


    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_uniform_(m.weight)

                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)

            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

            elif isinstance(m, nn.Linear):
                nn.init.kaiming_uniform_(m.weight)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):

        # <<TODO#3&#4>> Based on the above edits, you'll have
        # to edit the forward pass description here.

        #x = self.pool(F.relu(self.conv1(x)))
        # Output size = 28//2 x 28//2 = 14 x 14
        # 252/2 * 252/2 = 126 * 126

        #x = self.pool(F.relu(self.conv2(x)))
        # Output size = 10//2 x 10//2 = 5 x 5
        # 122/2 * 122/2 = 61 * 61

        # See the CS231 link to understand why this is 16*5*5!
        # This will help you design your own deeper network
        x = self.cnn(x)
        x = x.view(x.size()[0], -1)
        x = self.fc_net(x)

        # No softmax is needed as the loss function in step 3
        # takes care of that

        return x


# Create an instance of the nn.module class defined above:

net = BaseNet()
```

```
# For training on GPU, we need to transfer net and data onto the GPU
# http://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#training-on-gpu
if IS_GPU:
    net = net.cuda()



######################################################################
# 3. Define a Loss function and optimizer
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Here we use Cross-Entropy loss and SGD with momentum.
# The CrossEntropyLoss criterion already includes softmax within its
# implementation. That's why we don't use a softmax in our model
# definition.

import torch.optim as optim
criterion = nn.CrossEntropyLoss()

# Tune the learning rate.
# See whether the momentum is useful or not
optimizer = torch.optim.Adam(net.parameters(), lr=0.001,weight_decay=0.001)

plt.ioff()
fig = plt.figure()
device='cuda'
min_acc=0


val_loss_over_epochs = []
train_loss_over_epochs = []
val_accuracy_over_epochs = []
train_accuracy_over_epochs = []
for epoch in range(EPOCHS):

    # ---------- Training ----------
    # Make sure the model is in train mode before training.
    net.train()

    # These are used to record information in training.
    train_loss = []
    train_accs = []

    # Iterate the training set by batches.
    for batch in tqdm(trainloader):

        # A batch consists of image data and corresponding labels.
        imgs, labels = batch

        # Forward the data. (Make sure data and model are on the same device.)
        logits = net(imgs.to(device))

        # Calculate the cross-entropy loss.
        # We don't need to apply softmax before computing cross-entropy as it is done a
        loss = criterion(logits, labels.to(device))

        # Gradients stored in the parameters in the previous step should be cleared out
        optimizer.zero_grad()
```

```
        # Compute the gradients for parameters.
        loss.backward()

        # Update the parameters with computed gradients.
        optimizer.step()

        # Compute the accuracy for current batch.
        acc = (logits.argmax(dim=-1) == labels.to(device)).float().mean()

        # Record the loss and accuracy.
        train_loss.append(loss.item())
        train_accs.append(acc)

    # The average loss and accuracy of the training set is the average of the recorded
    train_loss = sum(train_loss) / len(train_loss)
    train_acc = sum(train_accs) / len(train_accs)

    # Print the information.
    print(f"[ Train | {epoch + 1:03d}/{EPOCHS:03d} ] loss = {train_loss:.5f}, acc = {tr

    # ---------- Validation ----------
    # Make sure the model is in eval mode so that some modules like dropout are disable
    net.eval()

    # These are used to record information in validation.
    valid_loss = []
    valid_accs = []

    # Iterate the validation set by batches.
    for batch in tqdm(valloader):

        # A batch consists of image data and corresponding labels.
        imgs, labels = batch

        # We don't need gradient in validation.
        # Using torch.no_grad() accelerates the forward process.
        with torch.no_grad():
          logits = net(imgs.to(device))

        # We can still compute the loss (but not the gradient).
        loss = criterion(logits, labels.to(device))

        # Compute the accuracy for current batch.
        acc = (logits.argmax(dim=-1) == labels.to(device)).float().mean()

        # Record the loss and accuracy.
        valid_loss.append(loss.item())
        valid_accs.append(acc)

    # The average loss and accuracy for entire validation set is the average of the rec
    valid_loss = sum(valid_loss) / len(valid_loss)
    valid_acc = sum(valid_accs) / len(valid_accs)
    if valid_acc > min_acc:
        # Save model if your model improved
        min_acc = valid_acc
        torch.save(net.state_dict(),'best')  # Save model to specified path
        print('Saving model (epoch = {:4d}, val_acc = {:.4f})'
              .format(epoch + 1, min_acc))
```

```
            .format(epoch + 1, min_acc));
        val_loss_over_epochs.append(valid_loss)
        train_loss_over_epochs.append(train_loss)
        val_accuracy_over_epochs.append(valid_acc)
        train_accuracy_over_epochs.append(train_acc)
        # Print the information.
        print(f"[ Valid | {epoch + 1:03d}/{EPOCHS:03d} ] loss = {valid_loss:.5f}, acc = {va

    # ------------------------------


    # Plot train loss over epochs and val set accuracy over epochs
    # Nothing to change here
    # -------------

    plt.ylabel('loss')
    plt.plot(np.arange(EPOCHS), train_loss_over_epochs, 'r-',label='train_loss')
    plt.plot(np.arange(EPOCHS), val_loss_over_epochs, 'b-',label='val_loss')
    plt.xticks(np.arange(EPOCHS, dtype=int))
    plt.show()

    plt.plot(np.arange(EPOCHS), val_accuracy_over_epochs, 'b-',label='val_acc')
    plt.plot(np.arange(EPOCHS), train_accuracy_over_epochs, 'r-',label='train_acc')
    plt.ylabel('accuracy')
    plt.xlabel('Epochs')
    plt.xticks(np.arange(EPOCHS, dtype=int))
    plt.show()
    plt.savefig("plot.png")
    plt.close(fig)

    print('Finished Training')
    print('Saving model ( val_acc = {:.4f})'
              .format( min_acc))
    # # -------------
```

```
100%                                      63/63 [00:16<00:00, 3.86it/s]

[ Train | 001/200 ] loss = 4.82272, acc = 0.10516
100%                                      4/4 [00:03<00:00, 1.20it/s]

Saving model (epoch =    1, val_acc = 0.1569)
[ Valid | 001/200 ] loss = 2.44370, acc = 0.15686
100%                                      63/63 [00:11<00:00, 5.27it/s]

[ Train | 002/200 ] loss = 2.50728, acc = 0.12103
100%                                      4/4 [00:02<00:00, 1.66it/s]

[ Valid | 002/200 ] loss = 2.49207, acc = 0.13157
100%                                      63/63 [00:08<00:00, 7.65it/s]

[ Train | 003/200 ] loss = 2.42008, acc = 0.12996
100%                                      4/4 [00:02<00:00, 1.92it/s]

Saving model (epoch =    3, val_acc = 0.1577)
[ Valid | 003/200 ] loss = 2.25053, acc = 0.15770
100%                                      63/63 [00:04<00:00, 14.06it/s]

[ Train | 004/200 ] loss = 2.29955, acc = 0.12599
100%                                      4/4 [00:02<00:00, 1.66it/s]
```

```
del net
net = BaseNet().to(device)
ckpt = torch.load('best', map_location='cpu')  # Load your best model
net.load_state_dict(ckpt)
```

```
100%                                      4/4 [00:02<00:00, 1.56it/s]
```

```
###############################################################################
# 5. Try the network on test data, and create .csv file
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
###############################################################################

# Check out why .eval() is important!
# https://discuss.pytorch.org/t/model-train-and-model-eval-vs-model-and-model-eval/5744
net.eval()

total = 0
predictions = []
for data in testloader:
    images, labels = data

    # For training on GPU, we need to transfer net and data onto the GPU
    # http://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#training-on-gpu
    if IS_GPU:
        images = images.cuda()
        labels = labels.cuda()

    outputs = net(Variable(images))
    _, predicted = torch.max(outputs.data, 1)
    predictions.extend(list(predicted.cpu().numpy()))
```

```
        predictions.extend(list(predicted.cpu().numpy()))
        total += labels.size(0)

    with open('submission_r08521609.csv', 'w') as csvfile:
        wr = csv.writer(csvfile, quoting=csv.QUOTE_ALL)
        wr.writerow(["Id", "Prediction1"])
        for l_i, label in enumerate(predictions):
            wr.writerow([str(l_i), str(label)])
```