```python
# -*- coding: utf-8 -*-
"""CNN.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1YiJluZWoyHPQNAVdLCPD7eLf5TlBQHTn

<center><h1>Mini Project 3 - Convolutional Neural Network</h1>
<h4>The PyTorch File.</h4></center>

<h3>Team Members:</h3>
<center>
Yi Zhu, 260716006<br>
Fei Peng, 260712440<br>
Yukai Zhang, 260710915
</center>

# Importation and import dataset
"""

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

# Commented out IPython magic to ensure Python compatibility.
# %cd '/content/gdrive/My Drive/ECSE_551_F_2020/Mini_Project_03/'
#!ls './data/'

import pickle
import matplotlib.pyplot as plt
import numpy as np
from torchvision import transforms
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from keras.layers import Flatten, Dense, Dropout, Activation, Conv2D, MaxPool2D, BatchNormalization
from PIL import Image
import torch

# Read a pickle file and disply its samples
# Note that image data are stored as unit8 so each element is an integer value between 0 and 255
data = pickle.load(open('./Train.pkl', 'rb'), encoding='bytes')
targets = np.genfromtxt('./TrainLabels.csv', delimiter=',', skip_header=1)[:,1:]
plt.imshow(data[1234,:,:],cmap='gray', vmin=0, vmax=256)
print(data.shape, targets.shape)

# get train size
train_size = data.shape[0]

# get test size
test_data = pickle.load(open('./Test.pkl', 'rb'), encoding='bytes')
print(test_data.shape)
test_size = test_data.shape[0]

"""# Dataset class
*Dataset* class and the *Dataloader* class in pytorch help us to feed our own training data into the network. Dataset class is used to provide an interface for accessing all the training or testing samples in your dataset. For your convinance, we provide you with a custom Dataset that reads the provided data including images (.pkl file) and labels (.csv file).

```

```python
# Dataloader class
Although we can access all the training data using the Dataset class, for neural networks, we would need batching, shuffling, multiprocess data loading, etc. DataLoader class helps us to do this. The DataLoader class accepts a dataset and other parameters such as batch_size.
"""

# Transforms are common image transformations. They can be chained together using Compose.
# Here we normalize images img=(img-0.5)/0.5
img_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# img_file: the pickle file containing the images
# label_file: the .csv file containing the labels
# transform: We use it for normalizing images (see above)
# idx: This is a binary vector that is useful for creating training and validation set.
# It return only samples where idx is True

class MyDataset(Dataset):
    def __init__(self, img_file, label_file, transform=None, idx = None):
        self.data = pickle.load(open( img_file, 'rb' ), encoding='bytes')
        self.targets = np.genfromtxt(label_file, delimiter=',', skip_header=1)[:,1:]
        if idx is not None:
          self.targets = self.targets[idx]
          self.data = self.data[idx]
        self.transform = transform
        self.targets -= 5

    def __len__(self):
        return len(self.targets)

    def __getitem__(self, index):
        img, target = self.data[index], int(self.targets[index])
        img = Image.fromarray(img.astype('uint8'), mode='L')

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

        return img, target

# Read image data and their label into a Dataset class
dataset = MyDataset('./Train.pkl', './TrainLabels.csv', transform=img_transform, idx=None)

batch_size = 32 #feel free to change it
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Read a batch of data and their labels and display them
# Note that since data are transformed, they are between [-1,1]
imgs, labels = (next(iter(dataloader)))
imgs = np.squeeze(imgs)
plt.imshow(imgs[5].cpu().numpy(),cmap='gray', vmin=-1, vmax=1) #.transpose()

"""# CNN"""

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

```

```python
import torch.optim as optim

# cnn
# This cnn is based on the structure of resnet18
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=(7,7), stride=(2, 2), padding=(3, 3), bias=False)
        self.bn1 = nn.BatchNorm2d(32, eps=1e-5, momentum=0.1, affine=True, track_running_stats=True)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)

        # Layer 1
        self.layer1block1conv1 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer1block1bn1 = nn.BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.layer1block1relu = nn.ReLU(inplace=True)
        self.layer1block1conv2 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer1block1bn2 = nn.BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

        self.layer1block2conv1 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer1block2bn1 = nn.BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.layer1block2relu = nn.ReLU(inplace=True)
        self.layer1block2conv2 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer1block2bn2 = nn.BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

        # Layer 2
        self.layer2block1conv1 = nn.Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        self.layer2block1bn1 = nn.BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.layer2block1relu = nn.ReLU(inplace=True)
        self.layer2block1conv2 = nn.Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer2block1bn2 = nn.BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.layer2block1conv3 = nn.Conv2d(64, 64, kernel_size=(1, 1), stride=(2, 2), bias=False)
        self.layer2block1bn3 = nn.BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

        self.layer2block2conv1 = nn.Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer2block2bn1 = nn.BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.layer2block2relu = nn.ReLU(inplace=True)
        self.layer2block2conv2 = nn.Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer2block2bn2 = nn.BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

        # Layer 3
        self.layer3block1conv1 = nn.Conv2d(64, 128, kernel_size=(3, 3), stride
```

```
=(2, 2), padding=(1, 1), bias=False)
        self.layer3block1bn1 = nn.BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.layer3block1relu = nn.ReLU(inplace=True)
        self.layer3block1conv2 = nn.Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer3block1bn2 = nn.BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.layer3block1conv3 = nn.Conv2d(128, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        self.layer3block1bn3 = nn.BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

        self.layer3block2conv1 = nn.Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer3block2bn1 = nn.BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.layer3block2relu = nn.ReLU(inplace=True)
        self.layer3block2conv2 = nn.Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer3block2bn2 = nn.BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

        # Layer 4
        self.layer4block1conv1 = nn.Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        self.layer4block1bn1 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.layer4block1relu = nn.ReLU(inplace=True)
        self.layer4block1conv2 = nn.Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer4block1bn2 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.layer4block1conv3 = nn.Conv2d(256, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        self.layer4block1bn3 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

        self.layer4block2conv1 = nn.Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer4block2bn1 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.layer4block2relu = nn.ReLU(inplace=True)
        self.layer4block2conv2 = nn.Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.layer4block2bn2 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1,1))
        self.fc = nn.Linear(in_features=256, out_features=9, bias=True)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        # Layer 1
        x = self.layer1block1conv1(x)
        x = self.layer1block1bn1(x)
        x = self.layer1block1relu (x)
        x = self.layer1block1conv2(x)
        x = self.layer1block1bn2(x)
```

```
        x = self.layer1block2conv1(x)
        x = self.layer1block2bn1(x)
        x = self.layer1block2relu (x)
        x = self.layer1block2conv2(x)
        x = self.layer1block2bn2(x)

        # Layer 2
        x = self.layer2block1conv1(x)
        x = self.layer2block1bn1(x)
        x = self.layer2block1relu(x)
        x = self.layer2block1conv2(x)
        x = self.layer2block1bn2(x)
        x = self.layer2block1conv3(x)
        x = self.layer2block1bn3(x)

        x = self.layer2block2conv1(x)
        x = self.layer2block2bn1(x)
        x = self.layer2block2relu(x)
        x = self.layer2block2conv2(x)
        x = self.layer2block2bn2(x)

        # Layer 3
        x = self.layer3block1conv1(x)
        x = self.layer3block1bn1(x)
        x = self.layer3block1relu(x)
        x = self.layer3block1conv2(x)
        x = self.layer3block1bn2(x)
        x = self.layer3block1conv3(x)
        x = self.layer3block1bn3(x)

        x = self.layer3block2conv1(x)
        x = self.layer3block2bn1(x)
        x = self.layer3block2relu(x)
        x = self.layer3block2conv2(x)
        x = self.layer3block2bn2(x)

        # Layer 4
        x = self.layer4block1conv1(x)
        x = self.layer4block1bn1(x)
        x = self.layer4block1relu(x)
        x = self.layer4block1conv2(x)
        x = self.layer4block1bn2(x)
        x = self.layer4block1conv3(x)
        x = self.layer4block1bn3(x)

        x = self.layer4block2conv1(x)
        x = self.layer4block2bn1(x)
        x = self.layer4block2relu(x)
        x = self.layer4block2conv2(x)
        x = self.layer4block2bn2(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x

train_index = np.arange(50000)
test_index = np.arange(50000, 60000)
batch_size = 32 #feel free to change it

# Read image data and their label into a Dataset class
```

```
train_set = MyDataset('./Train.pkl', './TrainLabels.csv', transform=img_transform, idx=train_index)
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True, num_workers=2)
test_set = MyDataset('./Train.pkl', './TrainLabels.csv', transform=img_transform, idx=test_index)
test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=True, num_workers=2)

net = Net()
# if there is a available cuda device, use GPU, else, use CPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net = net.to(device)

# set criterion to cross entropy loss
criterion = nn.CrossEntropyLoss()
# set learning rate to 0.001
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.5)

running_loss = 0.0
num_epochs = 32
for epoch in range(num_epochs):
    for i, data in enumerate(train_loader,0):
        img, label = data
        img = img.to(device)
        label = label.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(img)
        loss = criterion(outputs, label)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 320 == 319:  # print every 320 mini-batches
            print('[%d, %5d] loss: %.3f' %
                (epoch + 1, i + 1, running_loss / 320))
            running_loss = 0.0

            torch.save(net.state_dict(), '/model.pth')
            torch.save(optimizer.state_dict(), '/optimizer.pth')

print('Finished Training')

correct = 0
total = 0

# calculate accuracy
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        images = images.to(device)
        labels = labels.to(device)
        outputs = net(images)
        # get the index of the max output
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

```

```
print('Accuracy of the network: %d %%' % (
    100 * correct/ total))
```