ECSE 551 Fall 2020 Mini-project 3 Report

Fei Peng
260712440
fei.peng@mail.mcgill.ca

Yukai Zhang
260710915
yukai.zhang@mail.mcgill.ca

Yi Zhu
260716006
yi.zhu6@mail.mcgill.ca

Abstract

Image identification and classification are two of the most popular research areas in machine learning as well as computer vision. In this mini-project, we chose to investigate and evaluate the performance of different Convolutional Neural Network (i.e., CNN) models on a large image dataset. To have a better understanding of the structure of CNN, we implemented a model from scratch and analyzed its behavior. In addition, we tested different CNN layer arrangements and examined the effect of hyperparameters (e.g., batch size, number of epochs, and learning rate) before the optimum combination was selected, which yields the highest accuracy in the validation set. The performance of other CNN implementations provided by PyTorch and TensorFlow were also investigated, from which we found that VGG has the highest accuracy without any pre-trained data. According to our testing result, correctly choosing a proper CNN layer arrangement can greatly boost the result accuracy, avoid allocating too much memory as well as make sure the training process converges after reasonable iterations. Details will be presented in the following parts of this report.

1 Introduction

The increasing popularity of computer vision and the advance of machine learning technique have brought the opportunities to perform apparel classification [1]. The main objective of this mini-project is to apply machine learning methods on classification of clothes from five different categories where each has a distinctive price from $1 to $5. There are 60,000 monochrome images provided in the training dataset for this mini-project, and each image consists of three clothes. Therefore, their total price lies between $5 and $13 which has 9 different values. After applying machine learning methods on the training dataset, an additional 10,000 images are used to test the accuracy of the classification techniques and the results are submitted on Kaggle competition.

In this project, we are allowed to freely choose any machine learning methods, so as to explore their performance. However, the three clothes could appear in any random place in each image, and most of the classification methods we have been using so far in this course are sensitive to this position shift. Thus, a fully connected layer analysis (e.g., fully connected neural networks) is preferred as it can be used to achieve shift invariance, but performing this method requires a large number of training instances (i.e., weight parameters) as well as a long learning period, whereas we have limited computational resources and we are forbidden to import any pre-trained data [2]. Convolutional Neural Network, on the other hand, conserves the benefit of shift invariant by applying the same weight configuration across the image space. A convolution neural network consists of one or multiple convolutional layers, which are followed by fully connected layers. This network arrangement allows CNN to process/analyze an image in a similar manner as the biological neural systems.

Since the CNN method is wildly applied to solve many image-recognition problems, different types of network designs are available. Choosing the one that best fits this mini-project is essential. Thus, we investigated the performance of the CNN we implemented ourselves, and ResNet18 [3], AlexNet [4], VGG16 as well as VGG19 (both versions provided by PyTorch and TensorFlow) [5].

After comparing the validation accuracy of each implementation of CNN, VGG19 (by TensorFlow) was found to show a promising result. It yielded an accuracy of 99% in the validation dataset and 97% in the Kaggle competition.

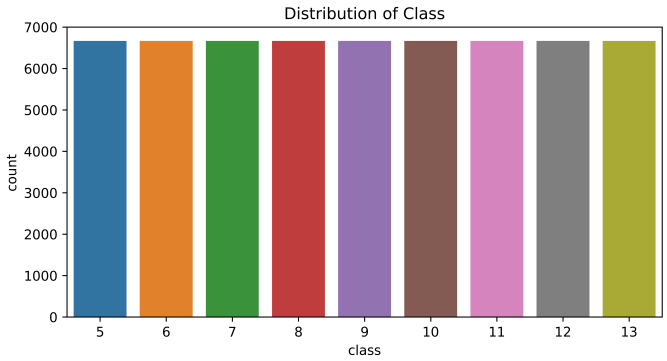

Figure 1: Class distribution of train data

2 Datasets

Before performing any investigation on the supervised machine learning and image classification models, it is important that the characteristics of the dataset (i.e., both features and classes) are properly analyzed and understood. Aside from that, data preprocessing lays the groundwork for the following data analysis, as its product is the final training set [6].

The training dataset we used is in $Train.pkl$ which contains 60,000 grayscale image samples, and their corresponding classes are imported from $TrainLabels.csv$. The test dataset is 10,000 grayscale image samples contained in $Test.pkl$, from which we predicted the classes and submitted the result to Kaggle competition.

In this project, the following data analyzation and preprocessing had been conducted.

2.1 Feature Analysis

The dataset is a modified version of the Fashion-MNIST dataset, which consists of samples with 641281 pixel values as features representing grayscale images. Therefore, the total number of features in each sample is 8192. As the number of samples in this dataset is numerous (i.e., 60,000 training samples), we considered using deep learning to achieve optimum accuracy with large dataset [7], from which we chose Convolutional Neural Networks (i.e., CNN) instead of fully connected neural networks for fewer memory requirements and lower latency [2]. Moreover, based on our previous experience, the images are relatively large and complex, therefore we anticipated to use deeper neural networks (i.e., more layers of neurons) for better accuracy. Details will be discussed later in Section 3.

2.2 Target Analysis

In this dataset, each image contains three articles, and our goal is to predict the total price of all articles presented in the image. The prices (i.e., classes) range from 5 to 13, therefore, it is a nine-way classification problem. The distribution of each class is shown in Figure 1. The uncertainty in prediction can be quantified by the entropy of the dataset with Equation 1, and the calculated data entropy is 3.17 which stands for an even distribution. Therefore, the number of samples in each class is equal. Moreover, before feeding the dataset to any training process, we need to make sure that the class values are encoded. Since the prices are integer numbers range from 5 to 13, we could simply encode them by subtracting the values by 5, thus the obtained target values would range from 0 to 8.

$$H(X) = -\sum_{x \epsilon X} p(x) log_2 p(x) \tag{1}$$

2.3 Normalization

The original pixel value of the images in the dataset is between 0 and 255, to centralize the value distribution while keeping its precision, we normalize the data with mean and standard division both equal to 0.5. Therefore, after normalization, the pixel values range from -1 to 1, which removes any scale factors that might influence the classification result [8].

3 Proposed Approach

In this section, we are going to discuss the models we implemented and the methods we used for this project, as well as the model and method selection in terms of accuracy, robustness/consistency, and speed. However, due to time constraint of this project, the reported accuracy and time might not be based on the absolute optimum hyperparameters, but rather try and error.

3.1 Training/Validation Splits

As mentioned previously in Section 2, the training dataset contains 60,000 samples, while the testing dataset contains 10,000 samples, however, only the training dataset contains the corresponding target classes. Therefore, for model selection, we chose to split the training dataset to 50,000 training samples and 10,000 testing samples, in order to perform validation. In this way, we could ensure consistency between the validation and testing process, and increase the possibility to reach higher accuracy on predicting the testing dataset. Due to the fact that deep learning methods are relatively more time consuming than normal machine learning algorithms, k-fold validation technique is not as common, thus, we compared the accuracy of different models based only on those 10,000 testing samples that are separated from the training dataset.

3.2 Hyperparameters

It is common practice to decay the learning rate in order to achieve better accuracy. However, one can usually obtain the same learning curve on both training and test sets by instead increasing the batch size during training [9]. As will be explained detaily later in Section 4, a smaller learning rate leads to a longer training time to approach the same accuracy. In CNN, it means that more epochs are required for training the network to preserve the same accuracy. Since the time consumed in each epoch is relatively constant, there is a linear relationship between number of epochs and total training time. However, increasing the learning rate reduces the number of parameter updates, thus, leads to greater parallelism and shorter training times. To compensate the influence of increasing the learning rate (e.g., by a factor of ϵ), we could increase the batch size by a factor (e.g., B) that is linearly proportional to the increase of learning rate (i.e., $B \propto \epsilon$) [9]. However, increasing the batch size alone could not ensure the same accuracy [10], whereas in our case decrease the accuracy as will be explained later in Section 4. Therefore, by tuning the learning rate and batch size together, we could select the most suitable combination to achieve higher accuracy and shorter training time. Moreover, we could increase the momentum coefficient (i.e., β) in gradient descent and scale $B \propto \frac{1}{1-\beta}$, while maintaining the same accuracy [9].

On the other hand, although increasing the number of epochs leads to longer training time, it also helps with achieving higher accuracy. In real life, it is usually a trade-off between choosing shorter training time and higher accuracy. In our case, since this is a competition on test accuracy, we would choose higher accuracy by sacrificing training time. However, accuracy will eventually converge, and continuing to train the model for more epochs will cause overtraining.

3.3 Algorithm Selection

At the beginning of this project, we started with building a Convolutional Neural Network (i.e., CNN) from scratch, in order to explore the structure and characteristics of the network. At the beginning, we configured a rather simple network, which starts with only four 2D convolution layers, each followed by a ReLU activation function as well as a 2D max pooling, and ends with two fully connected layers. However, we underestimated the requirement of the feature size (i.e., number of pixels in each image) on the depth of the network. The accuracy did not advance with the increasing number of epochs, but remained no difference from random classification, and could easily end up with data underflow/overflow, no matter how we changed the gradient descent and image normalization parameters. Therefore, we decided to increase the depth of our CNN model by learning and imitating one of the existing successful models provided by PyTorch [11], which is ResNet18 (i.e., residual network) [3], and tweaked the structure to suit our use. As a result, our CNN model contains twenty 2D convolution layers, followed by batch normalisation and ReLU activation functions, as well as one fully connected layer at the end. With this model, the running loss (i.e., cross entropy loss) started to converge as the number of epochs increased, and the accuracy began to rise.

Next, we continued to investigate the behavior of CNN models by performing experiments on ResNet18 [3], AlexNet [4], VGG16, VGG19 as well as VGG19 with batch normalization (i.e., VGG-19 BN) [5] provided by PyTorch [11], as well as VGG19 from TensorFlow (i.e., VGG-19 TF) [12], with minor adjustment on the models to fit our situation (i.e., input color channel equals one, and number of output class equals nine). AlexNet is one of the first Deep Convolutional Networks to achieve considerable accuracy, which consists of five convolutional layers and three fully connected layers, with ReLU as the activation function. VGG16 and VGG19 are both VGGNet, which differ only in the total number of layers in the network (i.e., 16 and 19 layers respectively). The idea of VGG is to use fixed size kernels so that all the variable size convolutional kernels used in AlexNet can be replicated by making use of multiple 3x3 kernels as building blocks. As a result, VGG reduces the number of parameters and the training time. ResNet18 are 18-layer residual nets, which have shortcut connection added to each pair of 3x3 filters compared to plain nets that are inspired by VGGNet. Thus, ResNet is able to find simpler mappings when they exist. Later in the next section (i.e., Section 4), we are going to discuss in detail the experiment results.

4 Results

When performing machine learning with CNN, multiple epochs are needed in order to obtain the highest accuracy. After evaluating the time and accuracy with respect to the number of epochs for the CNN implemented by ourselves, we found that the time required for each epoch is approximately constant and the total time is linearly proportional to the number of epochs. Accuracy, on the other hand, has an abrupt increase at the first several epochs. The slope of the curve gradually reduces as the accuracy approaches 1.0. Details are shown in Figure 2. After training for 32 epochs, our model achieved an accuracy of 84% and the training time is 995.74 seconds, which are shown in Table 1.

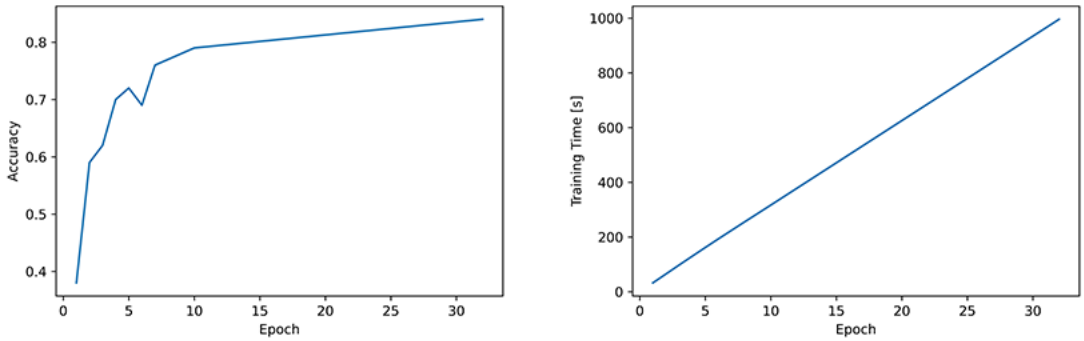

Figure 2: Accuracy (left) and training time (right) with respect to number of epochs

Table 1: Accuracy and training time for seven CNN models (32 epochs)

	CNN	ResNet-18	AlexNet	VGG-16	VGG-19	VGG-19 BN	VGG-19 TF
Accuracy	84%	87%	84%	94%	98%	98%	99%
Time (s)	995.74	1598.56	1275.41	5399.81	6151.02	6758.24	6085.63

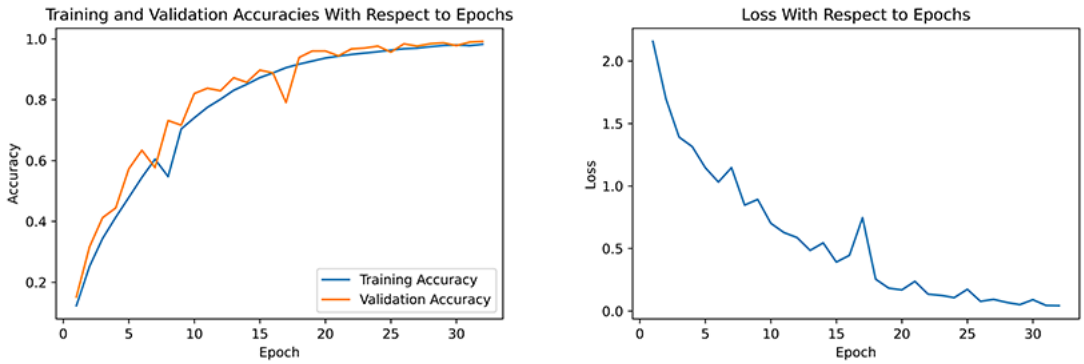

Figure 3: Training/validation accuracy (left) and loss (right) with respect to number of epochs

We then conducted experiments with other CNN models. The performances are recorded in Table 1 where 32 epochs of training are executed for each model. Despite having a slightly lower accuracy (84%), the CNN implemented by ourselves has demonstrated an incredible efficiency which takes only 15% of the training time compared to more complex models such as VGG. The VGG19 provided by TensorFlow, on the other hand, can yield the highest accuracy (99%) with a proper combination of hyperparameters. In our case, we chose SGD as our gradient descent method with a learning rate of 0.001 and a momentum of 0.7. We also set the loss function to $sparse\_categorical\_crossentropy$ and the metrics we use to evaluate the model is $accuracy$. We disabled the usage of pre-train data by setting the weight of VGG19 to be $None$ and we specified a batch size of 16. Its result was used in Kaggle competition and achieved a leaderboard accuracy of 0.98366.

To have a better understanding of this model (i.e., VGG-19 TF), we did further investigation. We picked 10,000 of the 60,000 training dataset as the validation set and the relationship between training/validation accuracies as well as loss with respect to epochs are shown in Figure 3. It is noticeable that the curve of validation accuracy keeps rising in the last few epochs and we can infer that overtraining has not occurred within the first 32 epoches. The loss decreases and approaches 0 as the number of epochs increases.

The accuracies and convergence time depend on not only the training models, but also the hyperparameters. The effects of the learning rate/momentum as well as batches size are investigated. We used VGG19 and VGG19 with batch normalization to test the effect of learning rate and momentum. Their accuracies after 32 epochs are shown in Table 2 and their loss curves are plotted in Figure 4. It appears that with a higher learning rate, the loss will have a more rapid drop in the first several epochs. The loss difference between different learning rates will decrease as the number of epochs gets larger.

Table 2: Accuracy performance of two CNN models under different hyperparameter sets (32 epochs)

	VGG-19	VGG-19 BN
Learning Rate = 0.001 & Momentum = 0	56%	68%
Learning Rate = 0.005 & Momentum = 0.5	98%	98%

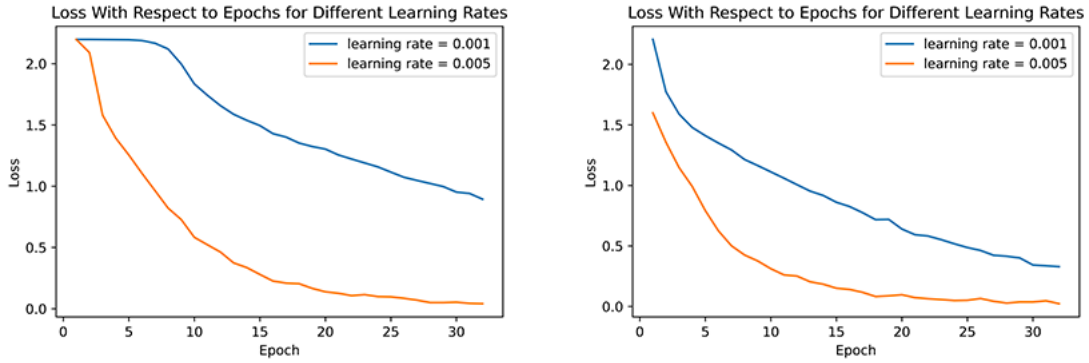

Figure 4: Loss with respect to epochs in different learning rates in VGG19 (left) and VGG19 BN (right)

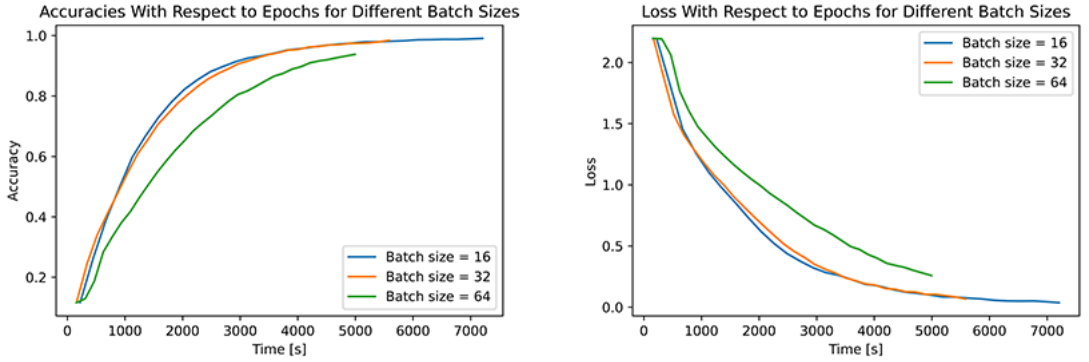

Figure 5: Accuracy (left) and loss (right) with respect to number of epochs for different batch sizes

5 Discussion and Conclusion

In conclusion, the Convolutional Neural Network model we implemented from scratch in this project performed as expected for classifying the clothes images. It achieved a validation accuracy of 84% after training for 32 epochs and the training process takes 995.74 seconds. However, the CNN models provided by TensorFlow yields the highest validation accuracy (99%) without using any pretrainined data. The validation accuracy is found to rise at a decreasing rate as the number of epochs grows. There does not exist a sign of overtraining within the first 32 epochs when the learning rate is set to be no larger than 0.005.

The future investigation could be to improve the layer arrangement of our self-implemented CNN and to find the best combination of hyperparameters in order to achieve a higher accuracy with less training time/epochs.

On the other hand, we achieved a leaderboard accuracy of 0.98366 using the VGG19 provided by TensorFlow without pre-trained data. 64 epochs of training were performed with a batch size of 16 in order to achieve this result as the validation accuracy barely changes (i.e., converges) at this point. In the future, we will take a closer look at the hyperparameters of the VGG19 algorithm to find the best combination that suits this machine learning problem.

6 Statement of Contributions

Yukai Zhang was in charge of researching the structure of the Convolutional Neural Network and implementing a CNN from scratch. Fei Peng browsed the models in PyTorch and TensorFlow, modified and trained the selected base models, as well as investigated the relationship between accuracy and number of epochs. Yi Zhu was responsible for analyzing the correlation between time and number of epochs, as well as proofreading the report.

References

[1] L. Bossard, M. Dantone, C. Leistner, C. Wengert, T. Quack, and L. Van Gool, "Apparel Classification with Style," Berlin, Heidelberg, 2013: Springer Berlin Heidelberg, in Computer Vision – ACCV 2012, pp. 321-335.

[2] S. Hijazi, R. Kumar, and C. Rowen, "Using Convolutional Neural Networks for Image Recognition By," 2015.

[3] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* Las Vegas, NV, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.

[4] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *ArXiv,* vol. abs/1404.5997, 2014.

[5] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *CoRR,* vol. abs/1409.1556, 2015.

[6] S. Kotsiantis, D. Kanellopoulos, and P. Pintelas, "Data Preprocessing for Supervised Leaning," *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering,* vol. 1, pp. 4104-4109, 2007.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature,* vol. 521, no. 7553, pp. 436-444, 2015/05/01 2015, doi: 10.1038/nature14539.

[8] G. Finlayson, B. Schiele, and J. Crowley, "Comprehensive Colour Image Normalization," in *ECCV,* 1998.

[9] S. L. Smith, P. Kindermans, and Q. V. Le, "Don't Decay the Learning Rate, Increase the Batch Size," *ArXiv,* vol. abs/1711.00489, 2018.

[10] Y. You, I. Gitman, and B. Ginsburg, "Scaling SGD Batch Size to 32K for ImageNet Training," *ArXiv,* vol. abs/1708.03888, 2017.

[11] A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *ArXiv,* vol. abs/1912.01703, 2019.

[12] M. Abadi et al., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *ArXiv,* vol. abs/1603.04467, 2016.

Appendix

(Please see following pages attached for code implementations)

```
# -*- coding: utf-8 -*-
"""Data_Preprocessing.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1ALcO_L0DWWHcrm6Qcxv-KH4TzpSpAlak

<center><h1>Mini Project 3 - Convolutional Neural Network</h1>
<h3>Data Preprocessing</h3>
<h4>This file performs some of the operations on Data Preprocessing and Analysis.</h4></center>

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

# Importations
"""

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

# make path = './' in-case you are running this locally
path = '/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_03/'

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from google.colab import files
from sklearn.preprocessing import LabelEncoder
from scipy.stats import entropy

"""# Data Preprocessing"""

dataset = pd.read_csv(path+"TrainLabels.csv")
# reddit_test = pd.read_csv(path+"test.csv")

y = dataset['class']

class Data_Processing:
    def __init__(self, data, name='New Data'):
        self.data = data
        self.name = name

    def show_y_dist(self, ydata):
        plt.figure(figsize=(8,4))
        plt.subplot(111), sns.countplot(x='class', data=ydata)
        plt.title('Distribution of Class')
        plt.savefig("Distribution of Class.png", dpi=1200)
        files.download("Distribution of Class.png")
        plt.show()

data_analysis = Data_Processing(dataset.values, 'TrainLabels.csv')
data_analysis.show_y_dist(dataset)

# calculate the data entropy
le = LabelEncoder()  # encoder for classes
```

```
63 le.fit(y)
64 y_label = le.transform(y)
65 n_k = len(le.classes_)
66 N = len(y)
67 theta_k = np.zeros(n_k)  # probability of class k
68 # compute theta values
69 for k in range(n_k):
70     count_k = (y_label==k).sum()
71     theta_k[k] = count_k / N
72
73 print("Data entropy is", entropy(theta_k, base=2))
```

```
# -*- 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).

```

```
# 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

```

```
113 import torch.optim as optim
114
115 # cnn
116 # This cnn is based on the structure of resnet18
117 class Net(nn.Module):
118     def __init__(self):
119         super(Net, self).__init__()
120         self.conv1 = nn.Conv2d(1, 32, kernel_size=(7,7), stride=(2, 2),
    padding=(3, 3), bias=False)
121         self.bn1 = nn.BatchNorm2d(32, eps=1e-5, momentum=0.1, affine=True,
    track_running_stats=True)
122         self.relu = nn.ReLU(inplace=True)
123         self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1,
    dilation=1, ceil_mode=False)
124
125         # Layer 1
126         self.layer1block1conv1 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride
    =(1, 1), padding=(1, 1), bias=False)
127         self.layer1block1bn1 = nn.BatchNorm2d(32, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
128         self.layer1block1relu = nn.ReLU(inplace=True)
129         self.layer1block1conv2 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride
    =(1, 1), padding=(1, 1), bias=False)
130         self.layer1block1bn2 = nn.BatchNorm2d(32, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
131
132         self.layer1block2conv1 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride
    =(1, 1), padding=(1, 1), bias=False)
133         self.layer1block2bn1 = nn.BatchNorm2d(32, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
134         self.layer1block2relu = nn.ReLU(inplace=True)
135         self.layer1block2conv2 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride
    =(1, 1), padding=(1, 1), bias=False)
136         self.layer1block2bn2 = nn.BatchNorm2d(32, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
137
138         # Layer 2
139         self.layer2block1conv1 = nn.Conv2d(32, 64, kernel_size=(3, 3), stride
    =(2, 2), padding=(1, 1), bias=False)
140         self.layer2block1bn1 = nn.BatchNorm2d(64, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
141         self.layer2block1relu = nn.ReLU(inplace=True)
142         self.layer2block1conv2 = nn.Conv2d(64, 64, kernel_size=(3, 3), stride
    =(1, 1), padding=(1, 1), bias=False)
143         self.layer2block1bn2 = nn.BatchNorm2d(64, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
144         self.layer2block1conv3 = nn.Conv2d(64, 64, kernel_size=(1, 1), stride
    =(2, 2), bias=False)
145         self.layer2block1bn3 = nn.BatchNorm2d(64, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
146
147         self.layer2block2conv1 = nn.Conv2d(64, 64, kernel_size=(3, 3), stride
    =(1, 1), padding=(1, 1), bias=False)
148         self.layer2block2bn1 = nn.BatchNorm2d(64, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
149         self.layer2block2relu = nn.ReLU(inplace=True)
150         self.layer2block2conv2 = nn.Conv2d(64, 64, kernel_size=(3, 3), stride
    =(1, 1), padding=(1, 1), bias=False)
151         self.layer2block2bn2 = nn.BatchNorm2d(64, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
152
153         # Layer 3
154         self.layer3block1conv1 = nn.Conv2d(64, 128, kernel_size=(3, 3), stride
```

```
154 =(2, 2), padding=(1, 1), bias=False)
155         self.layer3block1bn1 = nn.BatchNorm2d(128, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
156         self.layer3block1relu = nn.ReLU(inplace=True)
157         self.layer3block1conv2 = nn.Conv2d(128, 128, kernel_size=(3, 3),
    stride=(1, 1), padding=(1, 1), bias=False)
158         self.layer3block1bn2 = nn.BatchNorm2d(128, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
159         self.layer3block1conv3 = nn.Conv2d(128, 128, kernel_size=(1, 1),
    stride=(2, 2), bias=False)
160         self.layer3block1bn3 = nn.BatchNorm2d(128, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
161
162         self.layer3block2conv1 = nn.Conv2d(128, 128, kernel_size=(3, 3),
    stride=(1, 1), padding=(1, 1), bias=False)
163         self.layer3block2bn1 = nn.BatchNorm2d(128, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
164         self.layer3block2relu = nn.ReLU(inplace=True)
165         self.layer3block2conv2 = nn.Conv2d(128, 128, kernel_size=(3, 3),
    stride=(1, 1), padding=(1, 1), bias=False)
166         self.layer3block2bn2 = nn.BatchNorm2d(128, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
167
168         # Layer 4
169         self.layer4block1conv1 = nn.Conv2d(128, 256, kernel_size=(3, 3),
    stride=(2, 2), padding=(1, 1), bias=False)
170         self.layer4block1bn1 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
171         self.layer4block1relu = nn.ReLU(inplace=True)
172         self.layer4block1conv2 = nn.Conv2d(256, 256, kernel_size=(3, 3),
    stride=(1, 1), padding=(1, 1), bias=False)
173         self.layer4block1bn2 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
174         self.layer4block1conv3 = nn.Conv2d(256, 256, kernel_size=(1, 1),
    stride=(2, 2), bias=False)
175         self.layer4block1bn3 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
176
177         self.layer4block2conv1 = nn.Conv2d(256, 256, kernel_size=(3, 3),
    stride=(1, 1), padding=(1, 1), bias=False)
178         self.layer4block2bn1 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
179         self.layer4block2relu = nn.ReLU(inplace=True)
180         self.layer4block2conv2 = nn.Conv2d(256, 256, kernel_size=(3, 3),
    stride=(1, 1), padding=(1, 1), bias=False)
181         self.layer4block2bn2 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1,
    affine=True, track_running_stats=True)
182
183         self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1,1))
184         self.fc = nn.Linear(in_features=256, out_features=9, bias=True)
185
186     def forward(self, x):
187         x = self.conv1(x)
188         x = self.bn1(x)
189         x = self.relu(x)
190         x = self.maxpool(x)
191
192         # Layer 1
193         x = self.layer1block1conv1(x)
194         x = self.layer1block1bn1(x)
195         x = self.layer1block1relu (x)
196         x = self.layer1block1conv2(x)
197         x = self.layer1block1bn2(x)
```

```
198
199         x = self.layer1block2conv1(x)
200         x = self.layer1block2bn1(x)
201         x = self.layer1block2relu (x)
202         x = self.layer1block2conv2(x)
203         x = self.layer1block2bn2(x)
204
205         # Layer 2
206         x = self.layer2block1conv1(x)
207         x = self.layer2block1bn1(x)
208         x = self.layer2block1relu(x)
209         x = self.layer2block1conv2(x)
210         x = self.layer2block1bn2(x)
211         x = self.layer2block1conv3(x)
212         x = self.layer2block1bn3(x)
213
214         x = self.layer2block2conv1(x)
215         x = self.layer2block2bn1(x)
216         x = self.layer2block2relu(x)
217         x = self.layer2block2conv2(x)
218         x = self.layer2block2bn2(x)
219
220         # Layer 3
221         x = self.layer3block1conv1(x)
222         x = self.layer3block1bn1(x)
223         x = self.layer3block1relu(x)
224         x = self.layer3block1conv2(x)
225         x = self.layer3block1bn2(x)
226         x = self.layer3block1conv3(x)
227         x = self.layer3block1bn3(x)
228
229         x = self.layer3block2conv1(x)
230         x = self.layer3block2bn1(x)
231         x = self.layer3block2relu(x)
232         x = self.layer3block2conv2(x)
233         x = self.layer3block2bn2(x)
234
235         # Layer 4
236         x = self.layer4block1conv1(x)
237         x = self.layer4block1bn1(x)
238         x = self.layer4block1relu(x)
239         x = self.layer4block1conv2(x)
240         x = self.layer4block1bn2(x)
241         x = self.layer4block1conv3(x)
242         x = self.layer4block1bn3(x)
243
244         x = self.layer4block2conv1(x)
245         x = self.layer4block2bn1(x)
246         x = self.layer4block2relu(x)
247         x = self.layer4block2conv2(x)
248         x = self.layer4block2bn2(x)
249
250         x = self.avgpool(x)
251         x = torch.flatten(x, 1)
252         x = self.fc(x)
253
254         return x
255
256 train_index = np.arange(50000)
257 test_index = np.arange(50000, 60000)
258 batch_size = 32 #feel free to change it
259
260 # Read image data and their label into a Dataset class
```

```
261 train_set = MyDataset('./Train.pkl', './TrainLabels.csv', transform=
    img_transform, idx=train_index)
262 train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True,
    num_workers=2)
263 test_set = MyDataset('./Train.pkl', './TrainLabels.csv', transform=
    img_transform, idx=test_index)
264 test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=True,
    num_workers=2)
265
266 net = Net()
267 # if there is a available cuda device, use GPU, else, use CPU
268 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
269 net = net.to(device)
270
271 # set criterion to cross entropy loss
272 criterion = nn.CrossEntropyLoss()
273 # set learning rate to 0.001
274 optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.5)
275
276 running_loss = 0.0
277 num_epochs = 32
278 for epoch in range(num_epochs):
279     for i, data in enumerate(train_loader,0):
280         img, label = data
281         img = img.to(device)
282         label = label.to(device)
283
284         # zero the parameter gradients
285         optimizer.zero_grad()
286
287         # forward + backward + optimize
288         outputs = net(img)
289         loss = criterion(outputs, label)
290         loss.backward()
291         optimizer.step()
292
293         # print statistics
294         running_loss += loss.item()
295         if i % 320 == 319:  # print every 320 mini-batches
296             print('[%d, %5d] loss: %.3f' %
297                 (epoch + 1, i + 1, running_loss / 320))
298             running_loss = 0.0
299
300             torch.save(net.state_dict(), '/model.pth')
301             torch.save(optimizer.state_dict(), '/optimizer.pth')
302
303 print('Finished Training')
304
305 correct = 0
306 total = 0
307
308 # calculate accuracy
309 with torch.no_grad():
310     for data in test_loader:
311         images, labels = data
312         images = images.to(device)
313         labels = labels.to(device)
314         outputs = net(images)
315         # get the index of the max output
316         _, predicted = torch.max(outputs.data, 1)
317         total += labels.size(0)
318         correct += (predicted == labels).sum().item()
319
```

```
320 print('Accuracy of the network: %d %%' % (
321     100 * correct/ total))
```

File - E:\Study\ECSE551\Mini_Project_3\Code\pytorch_net.py

```
# -*- coding: utf-8 -*-
"""PyTorch_Net.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1VaRJXCTRYPnSrfmQcq6PVJuMZ5xxB250

<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>
"""

# Commented out IPython magic to ensure Python compatibility.
from google.colab import drive
drive.mount("/content/drive")

# %cd '/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_03/'

import numpy as np
import pandas as pd
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import pickle

from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from PIL import Image

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

import torch.optim as optim

import torchvision.models as models

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

img_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

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=None)
test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=True,
    num_workers=2)

"""# Notice: In case the code blocks are not in the right order, please run
    the blocks containing train() and test() functions every time after [model,
    criterion and optimizer] definition and before calling them.

# ResNet18
"""

resnet18 = models.resnet18()

net = resnet18
net.conv1 = nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3
    ), bias=False)
net.fc = nn.Linear(512, 9)
# 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=0.001)
# optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.5)

epoch = 32
train(epoch)

test()

"""# AlexNet"""

alexnet = models.alexnet()

net = alexnet
net.features[0] = nn.Conv2d(1, 64, kernel_size=(11, 11), stride=(4, 4),
    padding=(2, 2))
net.classifier[6] = nn.Linear(in_features=4096, out_features=9, bias=True)
# 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)
```

```
118
119 # set criterion to cross entropy loss
120 criterion = nn.CrossEntropyLoss()
121
122 # set learning rate to 0.001
123 # optimizer = optim.SGD(net.parameters(), lr=0.001)
124 optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.8)
125
126 epoch = 32
127 train(epoch)
128
129 test()
130
131 """# VGG16"""
132
133 vgg16 = models.vgg16()
134
135 net = vgg16
136 net.features[0] = nn.Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding
    =(1, 1))
137 net.classifier[6] = nn.Linear(in_features=4096, out_features=9, bias=True)
138 # if there is a available cuda device, use GPU, else, use CPU
139 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
140 net = net.to(device)
141
142 # set criterion to cross entropy loss
143 criterion = nn.CrossEntropyLoss()
144
145 # set learning rate to 0.001
146 optimizer = optim.SGD(net.parameters(), lr=0.001)
147 # optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.8)
148
149 epoch = 32
150 train(epoch)
151
152 test()
153
154 """# VGG19"""
155
156 vgg19 = models.vgg19()
157
158 net = vgg19
159 net.features[0] = nn.Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding
    =(1, 1))
160 net.classifier[6] = nn.Linear(in_features=4096, out_features=9, bias=True)
161 # if there is a available cuda device, use GPU, else, use CPU
162 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
163 net = net.to(device)
164
165 # set criterion to cross entropy loss
166 criterion = nn.CrossEntropyLoss()
167
168 # set learning rate to 0.001
169 # optimizer = optim.SGD(net.parameters(), lr=0.001)
170 optimizer = optim.SGD(net.parameters(), lr=0.005, momentum=0.5)
171
172 # with: optimizer = optim.SGD(net.parameters(), lr=0.001)
173 epoch = 32
174 train(epoch)
175
176 test()
177
178 # with: optimizer = optim.SGD(net.parameters(), lr=0.005, momentum=0.5)
```

```
epoch = 32
train(epoch)

test()

"""# VGG19-bn"""

vgg19bn = models.vgg19_bn()

net = vgg19bn
net.features[0] = nn.Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
net.classifier[6] = nn.Linear(in_features=4096, out_features=9, bias=True)
# 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=0.001)
optimizer = optim.SGD(net.parameters(), lr=0.005, momentum=0.5)

def train(num_epochs=2): # Feel free to change it
    net.train()

    running_loss = 0.0

    # Here is a piece of code that reads data in batch.
    # In each epoch all samples are read in batches using dataloader
    for epoch in range(num_epochs):
        for i, data in enumerate(train_loader):
            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 = F.nll_loss(outputs, label)
            loss.backward()
            optimizer.step()

            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')

def test():
    net.eval()

```

```
241     correct = 0
242     total = 0
243 
244     # calculate accuracy
245     with torch.no_grad():
246         for data in test_loader:
247             images, labels = data
248 
249             images = images.to(device)
250             labels = labels.to(device)
251 
252             outputs = net(images)
253             # get the index of the max output
254             _, predicted = torch.max(outputs.data, 1)
255             total += labels.size(0)
256             correct += (predicted == labels).sum().item()
257 
258     print('Accuracy of the network: %d %%' % (
259         100 * correct / total))
260 
261 # with: optimizer = optim.SGD(net.parameters(), lr=0.001)
262 epoch = 32
263 train(epoch)
264 
265 test()
266 
267 # with: optimizer = optim.SGD(net.parameters(), lr=0.005, momentum=0.5)
268 epoch = 32
269 train(epoch)
270 
271 test()
```

```
# -*- coding: utf-8 -*-
"""TensorFlow_Model.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1io3Hj9bFn56RH23DcKvgA2545zUIGxoc

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

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

import torch
import torchvision
import tensorflow as tf
from sklearn.model_selection import train_test_split
import time
import pickle
import numpy as np
import pandas as pd
from PIL import Image
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

from torch.utils.data import Dataset
from torch.utils.data import DataLoader

import tensorflow.keras as keras
from tensorflow.keras.applications.vgg19 import VGG19
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout
from tensorflow.keras.optimizers import SGD

# Commented out IPython magic to ensure Python compatibility.
from google.colab import drive
drive.mount("/content/drive")

# %cd '/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_03/'

"""# Import and Preprocess Dataset"""

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

img_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

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=None)

train_set_data = np.repeat(train_set.data[..., np.newaxis], 3, -1)
# train_x, x_validation, train_y, y_validation = train_test_split(
train_set_data, train_set.targets, test_size=0.20, random_state=0)
train_x = train_set_data
train_y = train_set.targets
print(train_x.shape, train_y.shape)

"""# Define Model"""

vgg19 = VGG19(weights=None, include_top=False)

# add a global spatial average pooling layer
x = vgg19.output
x = GlobalAveragePooling2D()(x)
x = Dropout(0.3)(x)
x = Dense(128, activation='relu')(x)
predictions = Dense(9, activation='softmax')(x)

# this is the model to train
model = Model(inputs=vgg19.input, outputs=predictions)

# make all layers trainable
for layer in vgg19.layers:
    layer.trainable = True

"""# Train"""

model.compile(keras.optimizers.SGD(learning_rate=0.001, momentum=0.7), loss='
sparse_categorical_crossentropy', metrics=['accuracy'])

# model.fit(train_x, train_y, epochs=45, validation_data=(x_validation,
y_validation), batch_size=32)
model.fit(train_x, train_y, epochs=65, batch_size=32)

"""# Output Export"""

test_data = pickle.load(open( './Test.pkl', 'rb' ), encoding='bytes')
test_data = np.repeat(test_data[..., np.newaxis], 3, -1)

from google.colab import files
import pandas as pd

y_test = model.predict(test_data, batch_size=32)
X_id = np.arange(y_test.shape[0])

```

```
y_class = np.argmax(y_test,axis=1) + 5
print(y_class)
result = {'id': X_id, 'class': y_class}

df = pd.DataFrame(data=result)
df.to_csv('result.csv', index=False)
files.download('result.csv')
```