Mini Project 3

Anuradha Colombathanthri
Department of Mathematics and Industrial Engineering
Polytechnique Montreal, UdeM
perera.colombathanthri@mail.mcgill.ca

Oren Gurevitch
Department of Physiology
McGill University
oren.gurevitch@mail.mcgill.ca

Sara Yabesi
Department of Computer Engineering
Polytechnique Montreal, UdeM
sara.yabesi@mail.mcgill.ca

Abstract

Convolutional Neural Network (CNN) is a supervised neural network model mainly used for image classification. The CNN architecture consists of layers of convolution using filters and pooling, which are then fed to the output layer through fully connected feed-forward neural network layers. The objective of this project was to build a 10-class CNN classifier using PyTorch libraries to classify a given image dataset as per the Japanese number included in each image. For this purpose, we used three types of CNN models namely ResNet18, ResNet34 and VGG16. The dataset that was given to us consists of image files. Each image file is a combination of Fashion-MNIST data and Japanese integer digits. The training set has 60,000 labeled grayscale images with an input size of 28*28, test set has 10,000 grayscale images with similar pixel sizes. The best ResNet18 model and VGG16 model could give a 91 and 86.6 accuracy respectively to classify the test set images as per their Japanese integer. The classified file 'Test.csv' provides the class number (the Japanese integer) in front of each test image ID. The results were also entered into the Kaggle competition, the group managed to be the seventh positioned with 91.00% best accuracy figure among all the submissions. We could gain knowledge on image classifiers and performance enhancements of neural networks through this project. The results are specific to this dataset, however, the developed CNN classifiers can be used by future learners for their image classification purposes.

1 Introduction

The expectation of this project was to develop a Convolutional Neural Network (CNN) classifier using PyTorch to identify the Japanese integer digit that consists in each pickle image file of the test data set.
In CNN architecture, Convolutional layers are used to extract the features of images using Kernels/filters, and pooling is used as a down-sampling method for the image features [1]. This

method of CNN architecture provides far less number of parameters per hidden layer of the neural network, unlike fully connected neural networks, therefore providing efficiency in classification.
Once the image features are well extracted through hidden convolutional + pooling layers, the data is fed forwarded to the output layer through fully connected neural network layers and activation functions.
The cited references were used throughout the project to gather the required knowledge. The approach we followed for this project is described below.

1.1 Analysis of the dataset

First and foremost, we went through the Train_labels.csv file to understand the labeling structure of the training dataset. We observed that there are no null values. Then we loaded the Train.pkl and Test.pkl files to Google Colab to extract their file sizes. There are 60,000 grayscale pickle images in the Training set file and each image is 28*28 pixels in size. Similarly, Test.pkl file also consists of 10,000 28*28 grayscale images.
An image file consists of figures extracted from a popular image classifier dataset 'Fashion-MNIST' and a Japanese integer digit. Each file consists of only one Japanese integer (from 0 to 9) and the Train_labels.csv file uses that integer to label each image, therefore this becomes a 10-class classification model.
For this machine learning report, data pre-processing was deemed unnecessary, as standard techniques such as normalization and image augmentation were employed, consistent with typical approaches to image classification tasks. Further details on these steps can be found in Chapter 2.

1.2 Development of CNN architecture

In deep neural network architecture such as CNN, we can encounter mainly two types of issues, degradation, and vanishing gradient.
Degradation means the problem of training error increase with the increase of layers, which affects the model accuracy.
A vanishing gradient means the problem of the gradient not being large enough to update the initial layers through backpropagation, resulting in the model not converging to a global minimum but to a local minimum of the loss function [2].
We developed a CNN classifier using Pytorch libraries that can mitigate these two issues in CNN.

ResNet18 architecture with modification (ModifiedResNet):
ResNet architectures are residual networks that use residual block stacking, and these models can address the aforementioned two issues by the use of the ReLu activation function [2].
We developed an ModifiedResNet model with 18 layers and eight residual blocks in total. Activation was done after every two residual blocks. ResNet18 architecture is meant for RGB image files. Therefore to feed in our grayscale images, we modified the model by adding 64 channels as the output of the first layer.
The filter size used was 7*7, and we also introduced size 1 stride and size 3 padding to retain the dimensions of the image.
We used 'drop-out' regularization method to remove random nodes with 0.2,0.25 and 0.5 probabilities as a method to mitigate the overfitting of our model.
This output was then fed to an output layer with 10 nodes, to denote a class by each node. The optimizer, activation function, drop out, batch size, and learning rate were the hyper-parameters that we experimented with to achieve the highest performance level. These results are elaborated on in Chapter 3 [figure 1].

1.3 Performance comparison experiment

The purpose of this experiment was to compare the developed CNN classifier performance against another image classifier.
We selected the Python in-built VGG16 classifier to classify the given dataset. The data pre-processing

step was the same as for the ResNet18 architecture that we developed. The results of this experiment are described in Chapter 3.

1.4 Output file

We created an output file with two columns, the first with the ID given in the test file, and the second column with the classified Japanese integer digit of each image.

2 Datasets

The training and testing data provided for this project were images consisting of Fashion-MNIST and Japanese integer digits. Fashion-MNIST is an open-source image dataset that is commonly used for image classifier developments, the Japanese integer file was specially developed for this project and was provided.
Each image file consists of three or four random items extracted from the Fashion-MNIST dataset and only one Japanese integer digit. The objective was to classify each image file as per the Japanese integer consisting of that file. Therefore, this classification is a multi-class classification with a class size of 10.
This chapter elaborates on the pre-processing and feature extraction methods that we used before training our model.

2.1 Training Dataset

This dataset consists of 60,000 grayscale images. Each image is of 28*28 pixel size. The images are converted to serialized pickle images [3] and saved in 'Train.pkl' file. These images are all labeled, the label file is 'Train_labels.csv' with no null values.

2.2 Test Dataset

This dataset consists of 10,000 grayscale images. Each image is of 28*28 pixel size. The images are converted to serialized pickle images [3] and saved in 'Test.pkl' file. The purpose of the project is to label each of these images as per the Japanese integer digit contained in it and create the output file 'Test_labels.csv' as afore-described.

2.3 Data pre-processing

As with any machine learning model, we initiated some useful pre-processing techniques to extract the features that are useful for our model to be trained on.

Tensor Transform:

We used transforms.Totensor() function to transform image data to PyTorch tensors.

Normalization:

We assessed the impact of preprocessing methods on the input data by utilizing the img_transform variable. We attempted normalization using a mean value of 0.5, the dataset's mean (0.00080), and standard deviation (STD) values of 0.5 or the dataset's STD (0.0011). In each instance, the cross-validation accuracy remained steady at approximately 10%. As a result, we determined that the data was already normalized, and additional normalization was not required.

Image augmentation:

We then investigated image transformations that preserved center invariance by employing the RandomAffine function from PyTorch [4]. We tested a range of transformation degrees, including 5, 10, 15, 20, 25, and 30 degrees. Despite this, the cross-validation accuracy consistently remained at 10%. Based on these findings, we hypothesized that the application of transformations to the dataset

might be redundant, given that all data digits used the same font.

These pre-processing steps were done for both training and test sets before training the model.

3 Results

As described, we developed an ModifiedResNet (modified ResNet18) model to classify the image dataset. The model was trained for five epochs with a batch size of 1000.

3.1 Selection of hyper-parameters of ModifiedResNet model

We experimented with four hyper-parameters in the model, optimizer, activation function, the learning rate and drop-out rate.

Selection of the optimizer: Optimizer selection was done by experimenting Adam and Stochastic gradient descent (SGD). We achieved to the highest accuracy with Adam optimizer. We used Adam (Adaptive Moment Estimation) function, due to its inherent properties that are useful for large-scale image data classifications such as straightforward implementation, efficiency in converging, and less memory usage to name a few [5].
Selection of activation functions: We used two activation functions such as ReLu and Leaky ReLu. However, in this implementation, ReLU gave us a higher accuracy on our prediction.
Selection of the learning rate: We performed a brief experiment to select the best-performing learning rate from 0.01 to 0.001 for both activation functions.
The results we achieved by these hyper-parameter changes are shown in Figure 1.

As per the achieved results, ModifiedResNet with ReLU activation, and 0.001 learning rate could give the highest accuracy of 91% for this classification, we reported the output file related to this performance into the Kaggle competition.

We also analyzed the training loss of each epoch and the graph in the code (see code in appendix)

3.2 Additional experiment for performance comparison

We used Python built-in VGG16 classifier on our dataset to compare the results with the ModifiedResNet model we developed. The main difference of this CNN architecture is that it does not use the residual block stacking method that is used by ResNet18 [6].
Figure 1 below shows the comparison of achieved results. As per this comparison, our ModifiedResNet model could give a 4.4% higher accuracy than the VGG 16 model.

4 Discussion and conclusion

This project was to gather knowledge on image classification using a Convolutional Neural Network (CNN), to classify a training data set of 60,000 labeled grayscale images with 28*28 pixel size. An image file consists of Fashion-MNIST images and a Japanese integer.
The Japanese integer was the basis for the classification, hence this was a 10-class classification problem. The model was tested by using 10,000 grayscale images with 28*28 pixel size, which was provided as the test dataset. Data pre-processing steps were performed as described to enable useful features for the model training.
We developed a CNN classifier with residual block stacking namely ModifiedResNet (ResNet18 with modifications). The intention of selecting this particular CNN architecture was to mitigate two inherent issues of CNN, degradation and vanishing gradient.
Developed ModifiedResNet uses eight residual blocks in total, an activation function was introduced after every two blocks. However, as ResNet18 is meant for RGB files, to make it grayscale-friendly, we modified the output of the first layer to 64 channels. This ModifiedResNet model uses a 7*7 filter, with size 1 stride and size 3 padding to retain the dimensions of the image.

The model was trained for five epochs with a 1000 batch size, this parameter selection was random. We selected the Adam optimizer due to its inherent properties that are useful for such classifications. Activation function and learning parameter selection were done as per trial-and-error experiments, as shown in Figure 1.
Out of these results, we observed that the best accuracy was achieved when the model was trained with ReLU activation function and when the learning rate was 0.001 The achieved accuracy was 91% to classify this image data set. This result was logged into the Kaggle competition and the leaderboard result which was taken at 5pm on 13/4/2023 shows that Group 6 could log in the 7th best accuracy level among all the teams participating.

Additionally, we performed a performance comparison experiment using Python built-in VGG16 classifier. The main difference between VGG16 and ModifiedResNet architecture is that VGG16 does not use the residual block stacking method.
As per the reported observations, it was clear that our ModifiedResNet (ResNet18 with modifications) model could outperform the VGG16 model by 86.6% in accuracy. ResNet18 (and other ResNet models) are meant to handle degradation and vanishing gradient issues, which are inherent in deep neural networks. This is the reason behind this performance improvement in classification by using ModifiedResNet over VGG16 [figure 1].

With this, it can be concluded that the project objective of developing a CNN classifier for a 10-class classification of an image dataset was achieved. We also could achieve our additional objective of analyzing the best performer for such image classifications.
The results generated are specific to the selected datasets, however, the developed model can be used by future learners for the classification of their own datasets by changing the learning parameters accordingly.
Due to the time constraints of this project, we compared our model against a single built-in model only, comparing this ResNet model against other available built-in classifiers remains a potential avenue for future researchers.

5 Statement of Contributions

Anuradha Colombathanthri - Report writing, code preprocessing, algorithm implementation, model training, model evaluation.
Oren Gurevitch - Report writing, code preprocessing, algorithm implementation, model training, model evaluation.
Sara Yabesi - Report writing, code preprocessing, algorithm implementation, model training, model evaluation.

attemps	CNN classifier	cross validation accuracy %	kaggle test accuracy %	layers	iterations	learning rate	batch size	dropout rate	optimizer
1	ResNet 18	83.25	80.07	16 ReLU	5	0.001	1000	0.5	adam
2	ResNet 34	85.09	79.06	32 ReLU	20	0.01	100	0.5	SGD
3	ResNet 18	93.442	87.43	16 ReLU	5	0.001	64	0.25	adam
4	ResNet 18	93.267	86.33	8 LeakyReLU	5	0.001	64	0.25	adam
5	**ResNet 18**	**95.63**	**91**	**16 ReLU**	**50**	**0.001**	**32**	**0.2**	**adam**
6	VGG 16	92.147	86.6	16 ReLU	20	0.001	1000	0.5	adam

Figure 1: Results of model comparison - Kaggle uploads

Appendix: References and code - Also given in the code.zip file:

References

[1] N. Armanfard, "Lecture 20, cnn," 2023.

[2] M. Siddharth, "Understanding resnet and analyzing various models on the cifar-10 dataset," 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/06/understanding-resnet-and-analyzing-various-models-on-the-cifar-10-dataset/

[3] A. Agrawal, "Understanding python pickling and how to use it securely," 2014. [Online]. Available: https://www.synopsys.com/blogs/software-security/python-pickling/#:~:text=Pickle%20in%20Python%20is%20primarily,transport%20data%20over%20the%20network.

[4] PyTorch, "Randomaffine." [Online]. Available: https://pytorch.org/vision/main/generated/torchvision.transforms.RandomAffine.html

[5] B. Jason, "Gentle introduction to the adam optimization algorithm for deep learning," 2017. [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[6] S. Mascarenhas and M. Agarwal, "A comparison between vgg16, vgg19 and resnet50 architecture frameworks for image classification," 2021.

[7] J. Brownlee, "How to develop a cnn for mnist handwritten digit classification," 2021. [Online]. Available: https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/

[8] S. Rajwal, "Classification of handwritten digits using cnn," 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/07/classification-of-handwritten-digits-using-cnn/

[9] A. Pandian, "Digit recognition using cnn (99 accuracy)," 2020. [Online]. Available: https://www.kaggle.com/code/arunrk7/digit-recognition-using-cnn-99-accuracy

[10] K. Patel, "Mnist handwritten digits classification using a convolutional neural network (cnn)," 2019. [Online]. Available: https://towardsdatascience.com/mnist-handwritten-digits-classification-using-a-convolutional-neural-network-cnn-af5fafbc35e9

[11] T. Pakhale, "Transfer learning using vgg16 in pytorch," 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/06/transfer-learning-using-vgg16-in-pytorch/

[12] T. Laplanche, "How to train your resnet: The jindo dog," 2020. [Online]. Available: https://medium.com/analytics-vidhya/how-to-train-your-resnet-the-jindo-dog-50551117381d

[13] G. Singhal, "Transfer learning with resnet in pytorch," 2020. [Online]. Available: https://www.pluralsight.com/guides/introduction-to-resnet

[7] [8] [9] [10] [11] [12] [4] [13]

```
# -*- coding: utf-8 -*-
"""mp3_final.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1GYyUAnyws3KGHnV6j3Bzaq7JZA6oHrY9
"""

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 PIL import Image
import torch

from sklearn.model_selection import train_test_split
from torchvision.datasets import KMNIST
import matplotlib.pyplot as plt

import time
import pandas as pd
from tqdm.notebook import tqdm_notebook
import random

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import cv2
from skimage.util import img_as_ubyte
from torchvision.models import VGG, vgg16

# Check if using GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
if torch.cuda.is_available():
  print(f"Nvidia Cuda/GPU is available!")

gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
```

```
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Select the Runtime > "Change runtime type" menu to enable a GPU accelerator, ')
  print('and then re-execute this cell.')
else:
  print(gpu_info)

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

# Commented out IPython magic to ensure Python compatibility.
#Folder of the  drive is accessed
# %cd '/content/gdrive/MyDrive/ecse551-mp3'

# 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
train_data  = pickle.load( open( './Train.pkl', 'rb' ), encoding='bytes').reshape(-1,28,28)
train_targets   = np.genfromtxt('./Train_labels.csv', delimiter=',', skip_header=1)[:,1:]
test_data  = pickle.load( open( './Test.pkl', 'rb' ), encoding='bytes')
plt.imshow(train_data[0,:,:].squeeze(),cmap='gray')

print('Training data shape:', train_data.shape)
print('Training targets shape:', train_targets.shape)
print('Test data shape:', test_data.shape)

"""#Dataset class
#Dataloader class
"""

#Compute training data's mean and standard deviation
#Divide by 255 to scale grayscale values (0-255) to 0-1 range for neural network input

train_data = np.random.random((100, 100)) # Example data

mean_val = np.mean(train_data) / 255 # Compute the mean of the pixel values
std_val = np.std(train_data) / 255 # Compute the standard deviation of the pixel values

print(f"Mean: {mean_val}, Standard Deviation: {std_val}")

"""**img_transform preprocessing**"""

# Transforms are common image transformations. They can be chained together using Compose.
```

```
"""**img_transform preprocessing**"""

# Transforms are common image transformations. They can be chained together using Compose.
''' Doesnt work
img_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.RandomAffine(degrees=5), ### hyperparameter "degrees" ###
])
'''
img_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.RandomAffine(degrees=5), ### hyperparameter "degrees" ###
    transforms.Normalize([mean_val], [std_val]) ### hyperparameters "mean" and "std" ###
])
''' Doesnt work
img_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.RandomAffine(degrees=5), ### hyperparameter "degrees" ###
    transforms.Normalize(0.5, 0.5) ### hyperparameters "mean" and "std" ###
])
'''

class CustomDatasetTrain(Dataset):
  def __init__(self, image_file, label_file, data_transform=None, idx = None):
    self.image_data = pickle.load( open( image_file, 'rb' ), encoding='bytes') # load the image data
    self.label_data = np.genfromtxt(label_file, delimiter=',', skip_header=1)[:,1:] # skip the first row
    self.data_transform = data_transform

    if idx is not None:
      self.label_data = self.label_data[idx] # select the idx-th row
      self.image_data = self.image_data[idx] # select the idx-th row

  def __len__(self):
    return len(self.label_data) # return the number of samples

  def __getitem__(self, idx):
    image, label = self.image_data[idx], int(self.label_data[idx]) # get the idx-th sample

    if self.data_transform is not None: # if there is a transform given, apply this transform to the image
      image = image.reshape(28,28) # reshape the image to 28x28
      image = Image.fromarray(image.astype('float'), mode='1') # convert the image to PIL image, mode='1' means the image is binary
      image = self.data_transform(image) # apply the transform to the image
```

```
    return image, label

class CustomDatasetTest(Dataset):

    def __init__(self, img_path, transform=None, indices=None):
        self.images = pickle.load(open(img_path, 'rb'))

        if indices is not None:
            self.images = self.images[indices]

        self.transform = transform

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

    def __getitem__(self, index):
        image = self.images[index]
        image = Image.fromarray(image.astype('uint8'), mode='L')

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

        return image

## ---- Load data for entire training set ---- ##
# Read image data and their label into a Dataset object #

## Note:- Selecting approximately 1000 test data from the set of 60000 as well.

training_dataset = CustomDatasetTrain('./Train.pkl', './Train_labels.csv', idx=None)

# Load data using DataLoader
batch_size = 32                                                    # Set batch size based on the size of the training or test data
training_data = DataLoader(training_dataset, batch_size=batch_size, shuffle=True)

data = enumerate(training_data)
idx, (inputs, targets) = next(data)
print(inputs.shape)

fig = plt.figure()
for i in range(6):
  plt.subplot(2, 3, i+1)
  plt.tight_layout()
```

```
for i in range(6):
  plt.subplot(2, 3, i+1)
  plt.tight_layout()
  plt.imshow(inputs[i][0], cmap='gray', interpolation='none')
  plt.title("Ground Truth: {}".format(targets[i]))
  plt.xticks([])
  plt.yticks([])
fig.show()

# Read a batch of data and their labels and display them
data_batch, target_labels = next(iter(training_data))
image = np.squeeze(data_batch)
plt.imshow(image[1].reshape(28, 28).cpu().numpy(), cmap='binary', vmin=0, vmax=1)

# Load test data
test_file = 'Test.pkl'
test_batch_size = 32

# Load test data using pickle
with open(test_file, 'rb') as file:
    test_data = DataLoader(pickle.load(file, encoding='bytes'), batch_size=test_batch_size)

# Shape when dataloader is used.
print(data_batch.shape)
print(target_labels.shape)

"""#ResNet - CNN"""

import torch.nn as nn
import torch.optim as optim
from torchvision.models.resnet import ResNet, BasicBlock, Bottleneck

# ---- Modified ResNet 18 ----- #
class ModifiedResNet(ResNet):
    def __init__(self):
        super(ModifiedResNet, self).__init__(BasicBlock, [2, 2, 2, 2], num_classes=10)
        self.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=1, padding=3, bias=False)

model_resnet = ModifiedResNet()

adam_optimizer = optim.Adam(model_resnet.parameters(), lr=0.001)
loss_criterion = nn.CrossEntropyLoss()
```

```
"""#Training and Testing on ResNet"""

# Define dropout probability
dropout_prob = 0.2

# Clone ResNet model
num_ft = model_resnet.fc.in_features

# Replace fully connected layer with a new one with dropout
new_fc = nn.Sequential(nn.Dropout(p=dropout_prob), nn.Linear(num_ft, 10))
model_resnet.fc = new_fc

# Move the model to GPU
model_resnet.to('cuda')

"""ResNet Model - Training & Validation"""

# Load training data and labels
train_data_source = CustomDatasetTrain('./Train.pkl', './Train_labels.csv', idx=None) # Load training data and labels

# Define sizes of training and validation sets
training_portion = int(0.8 * len(train_data_source)) # 80% of the data is used for training
validation_portion = len(train_data_source) - training_portion # 20% of the data is used for validation

# Split data into training and validation sets, using random_split
train_subset, validation_subset = torch.utils.data.random_split(train_data_source, [training_portion, validation_portion])

# Load data using DataLoader
data_batch_size = 100
train_loader = DataLoader(train_subset, batch_size=data_batch_size, shuffle=True)
validation_loader = DataLoader(validation_subset, batch_size=data_batch_size, shuffle=True)

## ----- Training Function for the ResNet ----- ##

def resnet_training(epoch, train_loader):
  model_resnet.train() # Set the model to training mode
  correct_predictions = 0 # Initialize the number of correct predictions
  total_predictions = 0 # Initialize the number of total predictions
  for idx, (data, label) in enumerate(train_loader): # Iterate over the training data
    data, label = data.cuda(), label.cuda() # Move the data and labels to the GPU
    adam_optimizer.zero_grad() # Clear the gradients
    model_output = model_resnet(data).cuda() # Forward pass
    prediction = torch.max(model_output.data, 1) # Get the predictions
```

```
 correct_predictions = 0 # Initialize the number of correct predictions
 total_predictions = 0 # Initialize the number of total predictions
 for idx, (data, label) in enumerate(train_loader): # Iterate over the training data
   data, label = data.cuda(), label.cuda() # Move the data and labels to the GPU
   adam_optimizer.zero_grad() # Clear the gradients
   model_output = model_resnet(data).cuda() # Forward pass
   _, prediction = torch.max(model_output.data, 1) # Get the predictions

   correct_predictions += (prediction == label).sum().item() # Update the number of correct predictions
   loss = loss_criterion(model_output, label) # Calculate the loss
   total_predictions += label.size(0) # Update the number of total predictions
   accuracy_pct = (correct_predictions / total_predictions) * 100.0 # Calculate the accuracy
   progress_visual.set_description(f"Loss : {loss.item():.3f}, Training Accuracy : {accuracy_pct:.3f}") # Update the progress bar

   loss.backward() # Backward pass
   adam_optimizer.step() # Update the weights

 print(f'Epoch : {epoch}; Training Accuracy: {accuracy_pct:.3f}') # Print the training accuracy

## ----- Validation Function for the ResNet ----- ##

def resnet_validatation(epoch, validation_loader): # Validation function
 correct_predictions = 0 # Initialize the number of correct predictions
 total_predictions = 0 # Initialize the number of total predictions
 highest_accuracy = 0 # Initialize the highest accuracy
 for idx, (data, label) in enumerate(validation_loader): # Iterate over the validation data
   data, label = data.cuda(), label.cuda() # Move the data and labels to the GPU
   model_output = model_resnet(data).cuda() # Forward pass
   _, prediction = torch.max(model_output.data, 1) # Get the predictions
   correct_predictions += (prediction == label).sum().item() # Update the number of correct predictions
   total_predictions += label.size(0) # Update the number of total predictions
   accuracy_pct = (correct_predictions / total_predictions) * 100.0 # Calculate the accuracy
   progress_visual.set_description(f"Validation Accuracy : {accuracy_pct:.3f}") # Update the progress bar

   if accuracy_pct > highest_accuracy:
     highest_accuracy = accuracy_pct

 print(f'Epoch : {epoch}; Validation Accuracy: {accuracy_pct:.3f}')
 return highest_accuracy


## ----- Running the epochs of ResNet ----- ##
```

```
## ----- Running the epochs of ResNet ----- ##

iteration_count = 5 # Set the number of iterations to run
best_epoch = 0
progress_visual = tqdm_notebook(iterable=range(iteration_count), position=0, leave=True)
highest_accuracy = 0
begin_time = time.time()

for idx in progress_visual: # Iterate over the number of iterations
  resnet_training(idx + 1, train_loader) # Run the training function
  curr_val_acc = resnet_validatation(idx + 1, validation_loader) # Get the current validation accuracy
  if curr_val_acc > highest_accuracy: # Check if the current validation accuracy is higher than the highest accuracy
    highest_accuracy = curr_val_acc # Store the highest accuracy
    best_epoch = idx # Store the epoch with the highest accuracy
    torch.save(model_resnet.state_dict(), '/model.pth') # Save the model
    torch.save(adam_optimizer.state_dict(), '/optimizer.pth') # Save the optimizer

finish_time = time.time() # Get the time when the training is finished

"""#Test and Save data"""

test_loader_final = test_data # This is the test data loader
predictions = []

def make_prediction(test_data=test_data):
    # Set the model to evaluation mode
    model_resnet.eval()

    test_loss = 0
    correct_count = 0

    with torch.no_grad():
        for single_data in test_data:
            # Transfer data to the device (CPU or GPU)
            single_data = single_data.to(device)

            model_output = model_resnet(single_data)
            prediction = model_output.data.max(1, keepdim=True)[1]
            predictions.append(prediction.tolist())

    flattened_predictions = [item for sublist in predictions for item in sublist] # Flatten the list, since the predictions are in a list of lists
    final_predictions = [item for sublist in flattened_predictions for item in sublist] # Flatten the list, since the predictions are in a list of lists
```

```
    final_predictions = [item for sublist in flattened_predictions for item in sublist] # Flatten the list, since the predictions are in a list of lists
    return final_predictions

def save_csv(predicted_data, output_file_name): # Save the predictions to a csv file
    formatted_data = {'id': np.arange(10000), 'class': predicted_data} # Create a dictionary with the data
    df = pd.DataFrame(formatted_data, columns=['id', 'class']) # Create a pandas dataframe with the data
    df.to_csv(output_file_name, index=False, header=True) # Save the dataframe to a csv file
    print("Data saved successfully! Check file name submit.csv") # Print a message to the user

# Record the start time of the program, make predictions, and record the end time
start_time = time.time()
predictions_made = make_prediction(test_data=test_data) # Make predictions on the test set

print(f"Execution Time: {time.time() - start_time:.2f} seconds") # Print the execution time

save_csv(predictions_made, "submit.csv") # Save the predictions to a csv file

"""# **Other training and validation codes, including ResNet 18 loss graph, ResNet 34, leakyReLU (with SGD optimizer) layers and VGG 16**

**ResNet 18 loss graph:**
"""

'''
#showing loss in graph fo report


sequence = list(range(0, 10000, 1)) # Create a sequence of numbers from 0 to 10000
batch_size = 32 # Set the batch size
grouped_sequence = [sequence[i:i + batch_size] for i in range(0, len(sequence), batch_size)] # Group the sequence into batches of size 32

test_idx = [i * len(training_data.dataset) for i in range(3)] # Create a list of indices to sample from the test set
peak_accuracy = 0 # Initialize peak accuracy to zero
average_accuracy = 0 # Initialize average accuracy to zero

num_iterations = 5 # Setting the number of epochs to run
progress_indicator = tqdm_notebook(iterable=range(num_iterations), position=0, leave=True) # Create a progress indicator

init_time = time.time() # Record the time at the start of training

loss_history = [] # Create an empty list to store training losses after each epoch

for current_epoch in progress_indicator: # Loop over the number of epochs
  model_resnet.train() # Set the model to training mode
```

```
for current_epoch in progress_indicator: # Loop over the number of epochs
  model_resnet.train() # Set the model to training mode
  correct_predictions = 0 # Initialize correct predictions to zero
  total_predictions = 0 # Initialize total predictions to zero
  epoch_losses = 0 # Initialize epoch loss to zero
  for batch_idx, (data_points, labels) in enumerate(training_data): # Loop over the training data
    data_points, labels = data_points.cuda(), labels.cuda() # Move data to GPU
    adam_optimizer.zero_grad() # Clear the gradients
    model_output = model_resnet(data_points).cuda() # Forward pass
    _, predicted_labels = torch.max(model_output, 1) # Get the predicted labels
    correct_predictions += (predicted_labels == labels).sum().item() # Update the number of correct predictions
    batch_loss = loss_criterion(model_output, labels) # Compute the loss
    total_predictions += labels.size(0) # Update the total number of predictions
    accuracy_pct = (correct_predictions / total_predictions) * 100.0 # Compute the accuracy
    progress_indicator.set_description(f"Loss : {batch_loss.item():.3f}, Main Data Accuracy : {accuracy_pct:.3f}") # Update the progress indicator
    epoch_losses += batch_loss.item() # Add batch loss to epoch loss
    batch_loss.backward() # Backpropagate the loss
    adam_optimizer.step() # Update the model parameters

  loss_history.append(epoch_losses / len(training_data))  # Save average training loss after each epoch

  print(f'Epoch : {current_epoch + 1} ; Training Loss on main data: {epoch_losses / len(training_data):.3f}') # Print the training loss after each epoch


plt.plot(range(1, num_iterations + 1), loss_history) # Plot the training loss vs epoch
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Training Loss vs Epoch')
plt.show()
'''

"""**ResNet 34 CNN model**"""

'''
class CustomResNet34(ResNet):
  def __init__(self):
    super(CustomResNet, self).init(BasicBlock, [3, 4, 6, 3], num_classes=10)
    self.initial_conv = nn.Conv2d(1, 64, kernel_size=7, stride=1, padding=3, bias=False)
'''

"""**LeakyReLU & SGD optimizer**"""
```

```
"""**LeakyReLU & SGD optimizer**"""

''' ignore to try leakyRelu
# ---- Modified ResNet 18 ----- #
class ModifiedResNet(ResNet):
    def __init__(self):
        super(ModifiedResNet, self).__init__(BasicBlock, [2, 2, 2, 2], num_classes=10)
        self.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=1, padding=3, bias=False)

model_resnet = ModifiedResNet()

adam_optimizer = optim.Adam(model_resnet.parameters(), lr=0.001)
loss_criterion = nn.CrossEntropyLoss()
'''
''' Ignored for leakyRelu
# Define dropout probability
dropout_prob = 0.2

# Clone ResNet model
num_ft = model_resnet.fc.in_features

# Replace fully connected layer with a new one with dropout
new_fc = nn.Sequential(nn.Dropout(p=dropout_prob), nn.Linear(num_ft, 10))
model_resnet.fc = new_fc

# Move the model to GPU
model_resnet.to('cuda')
'''

''' Leaky ReLU
from torch.nn import LeakyReLU
class LeakyBasicBlock(BasicBlock):
    def __init__(self, *args, **kwargs):
        super(LeakyBasicBlock, self).__init__(*args, **kwargs)
        self.relu = LeakyReLU(negative_slope=0.01, inplace=True)

# ---- LeakyRELU ----- #
class MFResNetLeaky(ResNet):
    def __init__(self):
        super(MFResNetLeaky, self).__init__(LeakyBasicBlock, [2, 2, 2, 2], num_classes=10) # Based on ResNet18
        self.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=1, padding=3, bias=False)
```

```
model_resnet = MFResNetLeaky()

#Create separate optimizers and criteria for Resnet
adam_optimizer = optim.Adam(model_resnet.parameters(), lr=0.001) #adam optimizer
#SGD_optimizer = optim.SGD(model_resnet.parameters(), lr=0.01, momentum=0.9) #SGD optimizer
loss_criterion = nn.CrossEntropyLoss()

num_ft = model_resnet.fc.in_features
model_resnet.fc = nn.Sequential(nn.Dropout(p=0.25), nn.Linear(num_ft, 10))
model_resnet = model_resnet.to('cuda')
'''

"""**VGG 16**

VGG 16 CNN model
"""

'''
class ModifiedVGG16(VGG):
    def __init__(self, num_classes=10):
        features = vgg16(pretrained=False).features

        # Remove the first MaxPool2d layer and the last three Conv2d layers
        features = nn.Sequential(*list(features.children())[:-10])

        super(ModifiedVGG16, self).__init__(features)
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )
        self.features[0] = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1)


vgg_model = ModifiedVGG16().cuda()

# Create separate optimizers and criteria for ModifiedVGG16
```

```
# Create separate optimizers and criteria for ModifiedVGG16
optimizer_vgg = optim.Adam(vgg_model.parameters(), lr=0.001)
criterion_vgg = nn.CrossEntropyLoss()
'''

"""VGG 16 optimizer & loss criterion"""

'''
optimizer_vgg_adam = optim.Adam(vgg_model.parameters(), lr=0.001)

loss_criterion_vgg = nn.CrossEntropyLoss()
'''

"""Training of VGG 16 parallel to ResNet 18"""

'''
lst_unique = list(range(0, 10000, 1))
batch_size = 1000
lst_splits = [lst_unique[i:i + batch_size] for i in range(0, len(lst_unique), batch_size)]

test_counts = [i*len(training_data.dataset) for i in range(3)]
best_accuracy = 0
average_accuracy = 0

num_epochs = 5

progress_bar_resnet18 = tqdm_notebook(iterable=range(num_epochs), position=0, leave=True, desc="ResNet 18 Progress")
progress_bar_vgg16 = tqdm_notebook(iterable=range(num_epochs), position=0, leave=True, desc="VGG 16 Progress")

start_time = time.time()

lst_training_losses_resnet18 = [] # Create an empty list to store training losses after each epoch for ResNet 18
lst_training_losses_vgg16 = [] # Create an empty list to store training losses after each epoch for vgg16

for epoch in range(num_epochs):

  model_resnet.train()
  correct1 = 0
  total1 = 0
  correct2 = 0
  total2 = 0
  epoch_loss_1 = 0
```

```
for batch_idx, (data, target) in enumerate(training_data):

  data, target = data.cuda(), target.cuda()
  adam_optimizer.zero_grad()
  output1 = model_resnet(data).cuda()
  _, predicted = torch.max(output1, 1)
  correct1 += (predicted == target).sum().item()
  loss1 = loss_criterion(output1, target)
  total1 += target.size(0)
  acc1 = (correct1 / total1) * 100.0
  progress_bar_resnet18.set_description(f"Loss Model 1: {loss1.item():.3f}, Main Data Accuracy Model 1: {acc1:.3f}")
  progress_bar_resnet18.update()
  loss1.backward()
  adam_optimizer.step()

  lst_training_losses_resnet18.append(epoch_loss_1/len(training_data))    # Save training accuracy after each epoch


# Train VGG16
vgg_model.train()
correct_2 = 0
total_2 = 0
epoch_loss_2 = 0
for batch_idx, (data, target) in enumerate(training_data):
    data, target = data.cuda(), target.cuda()
    optimizer_vgg_adam.zero_grad()
    output_2 = vgg_model(data).cuda()
    _, predicted_2 = torch.max(output_2, 1)
    correct_2 += (predicted_2 == target).sum().item()
    loss_2 = loss_criterion(output_2, target)
    total_2 += target.size(0)
    acc_2 = (correct_2 / total_2) * 100.0
    progress_bar_vgg16.set_description(f"Loss Model 2: {loss_2.item():.3f}, Main Data Accuracy Model 2: {acc_2:.3f}")
    progress_bar_vgg16.update()
    loss_2.backward()
    optimizer_vgg_adam.step()

    lst_training_losses_vgg16.append(epoch_loss_2/len(training_data))    # Save training accuracy after each epoch


torch.save(model_resnet.state_dict(), '/trained_model_1.pth')
torch.save(adam_optimizer.state_dict(), '/trained_optimizer_1.pth')
```

```
  torch.save(model_resnet.state_dict(), '/trained_model_1.pth')
  torch.save(adam_optimizer.state_dict(), '/trained_optimizer_1.pth')
  torch.save(vgg_model.state_dict(), '/trained_model_vgg16.pth')
  torch.save(optimizer_vgg.state_dict(), '/trained_optimizer_vgg16.pth')

# Close progress bars after the loop
progress_bar_resnet18.close()
progress_bar_vgg16.close()
'''

"""VGG 16 Test Prediction (without the store_csv function)"""

'''
test_loader_final = test_data # This is the test data loader
y_pred_vgg = []

def pred_vgg16(testdata=test_data):
    # evaluation mode
    vgg_model.eval()

    test_loss = 0
    correct_count = 0
    y_pred_vgg = []

    with torch.no_grad():
        for single_data in testdata:
          # transfer data to cuda if available
          data = data.to(device)

          model_output = vgg_model(single_data)
          prediction = model_output.data.max(1, keepdim=True)[1]
          y_pred_vgg.append(prediction.tolist())

    flattened_predictions = [item for sublist in y_pred_vgg for item in sublist] # Flatten the list, since the predictions are in a list of lists
    final_predictions = [item for sublist in flattened_predictions for item in sublist] # Flatten the list, since the predictions are in a list of lists
    return final_predictions

startall = time.time()
y_pred_vgg16 = pred_vgg16(testdata=test_data)
'''
```