Japanese Digit Classification Using Convolutional Neural Networks

Max Ardito
maxwell.ardito@mail.mcgill.ca

Edwin Meriaux
edwin.meriaux@mail.mcgill.ca

Ohood Sabr
ohood.sabr@mail.mcgill.ca

Abstract

In this paper, the performance of several Convolutional Neural Network (CNN) designs is evaluated using a variation on the MINIST dataset that includes both western Arabic numerals and Japanese numerals. Each image contains a single Japanese numeral and may also potentially have up to three western numerals. The goal of the classifier is to successfully classify the Japanese digit. This study examined the effectiveness and precision of both a base CNN designs and a CNN design that includes residual layers, as well as squeeze-and-excitation layers. The model that scored the highest accuracy is the later model, a variation on the famous ResNet-18 classifier. Preprocessing methods including digit rotation are also discussed.

Keywords: Convolutional neural network, CNN, Image classification, MNIST dataset, Residual layers.

1 Introduction

Computer vision (CV) deals with the issues and strategies of simulating human vision using computers. Its primary function is to analyze and comprehend images and videos in order to make conclusions about their content. Image recognition is a type of technology that employs machine learning models to analyze, evaluate, and classify various entities within images[1].

The CV field has greatly evolved during the last few decades. Convolution Neural Networks (CNNs) are a well-known CV model[2]. The standard CNN is a deep neural network made up of various layers of neurons that perform tasks such as filtering, pooling, and activation. Filtering and pooling layers in carry out feature extraction, while a fully connected layer perform feature mapping to produce the final output after being passed through activation functions[3]. What distinguishins a CNN from a "simple" multilayer perceptron (MLP) network is therefore this combined use of convolutional filtering layers, pooling, and non-linearities like tanh, sigmoid, and ReLU functions [4].

The focus of this work is to examine the classification performance of the CNN model in given a variation on the notable MNIST dataset that contains a Japanese numeral along with up to three western Arabic numerals in a single image. The class labels of these images refer to the digit value of the Japanese numeral.

This report is organized as follows: Section 2 discusses the data set and talks about the preprocessing steps that were taken. Section 3 presents the various models experimented with during the training phase. Section 4 presents the findings, and Section 5 concludes with some possible further research.

2 Datasets

2.1 Overview

The given dataset is a variation on the the MNIST data set which is a standard dataset given by the Modified National Institute of Standards and Technology [5]. The traditional dataset contains images, each of a handwritten digit (0-9) as well as their class label values. The variation of MNIST given for this project is a set of images, each of which contain a single handwritten Japanese digits and up to three western digits. The overall dataset consists of 60,000 training samples (images). The objective is to identify the Japanese digit present in the input image. Using CNNs, the multi-class image classification issue is resolved. After extracting images from the dataset, we have seen that each image has 28-pixel width and a 28-pixel height (28 X 28) and a grayscale (single channel) image. The intensity values for each pixel varies from 0 (black) to 255 (white). Figure 1 displays an example image from the dataset in question.

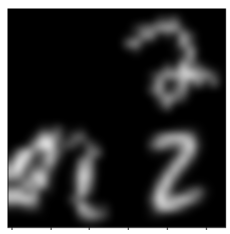

Figure 1: Traing image sample.

2.2 Preprocessing

Before the training data is fed into the different CNN designs, it goes through a number of preprocessing stages. Initially, the images are transformed from Numpy arrays into Pytorch tensors to enable high dimensional arithmetic operations on the GPU. In the next stage, the images are normalized so that their pixel values fall between -1 and 1. The advantage of normalizing the input data is that it prevents the training process from being difficult due to huge gradient values[5]. One other method that was tried was Sobel filtering to process images, however, this did not work out well mainly due to the fact that the images are only 28x28 pixels. The Sobel filter just created an output image that was effectively the same image due to the MNIST's small image size. The CNN models can also be fed directly with raw values using the greyscale pixels, however, doing so may result in longer training times and lower accuracy.

Aside from image quality improvement, augmentation was also used. This is used in order to expand the number of training instances without using new images, augmenting the original images and using both the original and augmented images for training. Another method is image rotation. As the name implies it means the images used in training set are rotated. This is done to augment the training dataset. This increases variation from the images in the training phase. Additionally, the handwritten digits in the MNIST dataset are not aligned at the same angle. This means that training the algorithm on images that are rotated should, in theory, also increase the accuracy. However, image rotation benefits were not seen in the output, as the accuracy in training and validation actually decreased. A possible way to improve this in the future would be to only rotate individual digits instead of the entire image. This can be done by separating the characters by recognizing the outline of each individual number before rotating and then merging the values into one image. A final preprocessing method used was image color augmentation. This works on modifying the pixel colors and intensity. This also did not improve accuracy, in part because the images are not in color.

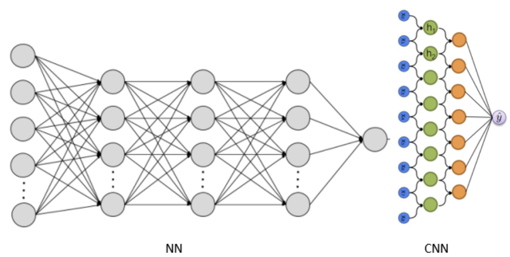

Figure 2: NN (Left) and CNN (Right) comparison.

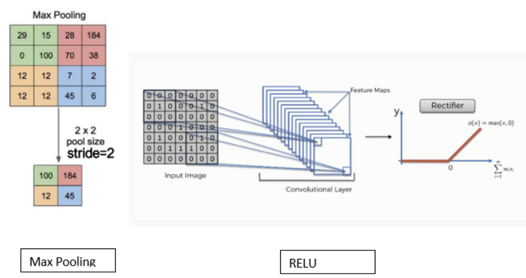

Figure 3: Visualizations of Max Pooling (Left) and ReLU activation (Right).

3 Proposed Approach

3.1 Base CNN Model

A Convolutional Neural Network (CNN) is a form of Neural Networks (NN) (Figure: 2) that works to try and reduce the computational complexity of a NN by attaching neurons on one layer only to a select number of neurons from the previous layer. This reduces the computation by reducing the number of features that each neurons has to take in. This makes the computation on each node lower and the back propagation faster as well. As the layers move from the broadest to the end the number of neurons generally decrease to an eventual single end neurons that takes input from all the previous neurons.

CNNs also contain additional layers for averaging and activation. For instance, Rectified Linear Activation Unit (RELU) layers are used to further help reduce computational complexity (Figure: 3). Effectively this is just a threshold value. In a one dimensional input case if the value is above a certain value then the output is a linear value greater than 0 but if it bellows a certain value then the output is 0. This threshold value is calculated as part of the training of the system to determine if the value is useful in the following layers. Turning the value to 0 makes it so that the next layer does not care for the output of this neutron.

Another layer used in CNNs to reduce the complexity is the pooling layers (Figure: 3). This filter reduces an image to a fraction of its current size by downsampling. It breaks up an image into specific rectangles and the output of the filter has each of those rectangles as just one pixel in the new image. This is done by averaging all the values the rectangle contained.

Various other reduction layers can be implemented in the base CNN model as well. The flatten function takes a PyTorch tensor and reshapes it into a one-dimensional tensor. This is similar to the squeeze function in NumPy which takes a NumPy matrix and turns it into a one dimension array. Adaptive Average Pool layer (Adaptiveavgpool) is a special pooling function that does not takes data from just one plane but multiple. The planes can be different inputs into the neutron in question. The linear activation layer is a different activation function. It is based off of linear regression and trained on the input values from the previous layer's neutrons.

3.2 Residual Layers

The aforementioned CNN layers are standard to all CNNs. These base layers can be configured in many different ways and their order and parameters resemble hyperparameters in the training process. Based on analyzing literature pertaining to models with high accuracy metrics on the original MNIST dataset, two additional layers were used in training a variation on the base CNN model.

The first type of additional CNN layer used is the residual layer, which allows for a sort of regularization in the network that solves the issue of vanishing or exploding gradient. Layers can be skipped and thus connected to the output of previous layers in the network in a non-sequential manner. The effect that this has on the network is not unlike that of LSTM blocks on recurrent neural networks (RNN). A semblance of memory is attained in the network and this regularization has been proven to increase accuracy on image recognition problems [6].

3.3 Squeeze and Excitation Layers

The concept of retaining both a balance of high and low-level features is perhaps the central conceit of CNN architectures. Learning adequate weights for high-level neurons that classify large objects and shapes is just as important as learning the weights for low-level neurons that distinguish edges and contours [7]. While residual layers attempt to establish this balance, squeeze and excitation (SE) layers take this once step further.

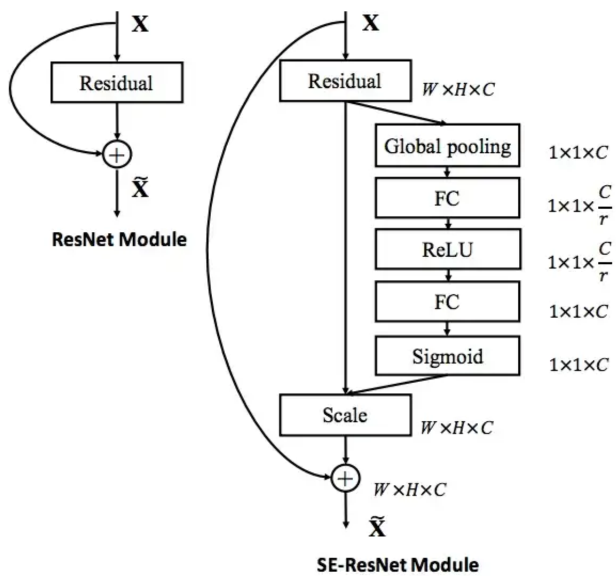

Figure 4: Squeeze-and-Excitation Layer (Right) compared to Residual Layer (Left) [7]

An SE layer is very similar to a residual layer, except in this case each channel is pooled or "squeezed" into a single value, passed to a ReLU activation function, reduced by a certain ratio r, and activated once more by the sigmoid function. Figure 4 shows a full diagram of the SE architecture compared to a residual layer.

4 Results

Based on techniques discussed in the previous sections, we constructed three different CNN architectures. The first architecture was a simple base CNN with no preprocessing and no residual layers. The second CNN was a model inspired roughly by the ResNet-18 model, with various changes including the addition of residual layers and SE layers. The third CNN used was simply the same as the second CNN model but with additional preprocessing steps. Each model was tested on a validation set consisting of 0.1% of the original training dataset (6,000 images). A batch size of 128 images was used on all models, and the number of epochs used to train each model varied between 2-15 depending on the accuracy metric of the first epoch.

The base model stood as a sort of benchmark attempt at designing a classifier that functioned properly. The CNN architecture used here consisted of two convolutional blocks, each of which consisted of a fully connected layer, a normalization layer, a ReLU layer, and a max pooling layer. Blocks were output to a linear layer which mapped the final number of output channels to 10 neurons, the maximum of which being the classified digit. Kernel sizes of 3 pixels were used on the image filters along with strides and paddings of 1 pixel. The base model scored 91% accuracy on the validation set but only 85% on the testing set.

Once this benchmark was established, a blueprint of ResNet-18 was used as a starting point to construct a network that yielded the highest accuracy. For this model, four convolutional blocks were constructed that consisted of a fully connected convolutional layer, a residual layer whose channels are sometimes downsampled, an SE layer with an r value that peaks in the middle of the network, ReLU activation, and average pooling. The final block contains no convolutional layers, but instead two residual layers that channels, average, flatten, and output to the 10 classifying neurons. This model scored 99% on validation data and 92% on the testing data.

Finally, rotational preprocessing was performed on the dataset, straightening out angles in the digits. While this improved accuracy on some training phases, the difference deemed the preprocessing step unnecessary.

5 Future Work (K-Means) and Conclusion

While the ResNet variation model performed quite well on the dataset, more refined preprocessing steps could indeed be taken given more time. For instance, each image could potentially be sliced up into four quadrants, and given the prior knowledge that each image contains up to four digits, one Japanese digit, and up to three western digits, a pre-classifier could be constructed using Bayesian probability and K-means clustering to classify which digit is the Japanese digit. It is estimated that this would greatly increase training, since passing a single Japanese digit into a CNN and estimating its value is a much simpler task than passing four images.

In order to do this, each image would be sliced up and a K-means clustering algorithm would be used to learn the differences between the images. This learned difference would distinguish blank spaces, western Digits, and Japanese digits. Then, using this data along with each type of digit's probability of occurring in a given quadrant, the model would identify the Japanese digit in each image and then pass it through the digit classifying CNN.

Additionally, future work could also include principal component analysis (PCA) as a pre-processing step to compress images through the use of an orthogonal transformation. PCA could help narrow down a complex data set's dimensions, and it is clearly stated in various pieces of literature that PCA improves the accuracy of CNNs [8]. Another possible preprocessing method that could also be done is cropping to reduce the amount of blank space in the images. This would reduce training on the blank spaces and only use the minimum amount of pixels of interest. Each image would be broken down into rectangles that act as bounding boxes for the characters, processing only the pixels in the boxes. A final possibility could be performing image segmentation. This breaks images down into multiple sections, effectively partitioning image content down to their boundaries. If this is done then by knowing the colors of each section a threshold can be used to know which paritions are characters. This can be used in the K-Means method or any method that requires determining the specific number and no extra pixels. This is also would by itself change the colors in the space by pushing the segments of interest to pure white colors and the rest to pure black. This would remove noise in the data associated with variations in the greyscale.

6 Statement of Contribution

1. Max Ardito: Report editing, Residual Layers, SE Layers, Training Architecture

2. Ohood Sabr: Report writing, Preprocessing (rotational method)

3. Edwin Meriaux: Report writing, Preprocessing (rotation and image augmentation), Basic RELU CNN algorithm, training and testing

References

[1] A. A. M. Al-Saffar, H. Tao, and M. A. Talab, "Review of deep convolution neural network in image classification," in *2017 International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET)*, pp. 26–31, 2017.

[2] W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural computation*, vol. 29, no. 9, pp. 2352–2449, 2017.

[3] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights into imaging*, vol. 9, no. 4, pp. 611–629, 2018.

[4] A. F. Agarap, "An architecture combining convolutional neural network (cnn) and support vector machine (svm) for image classification," *arXiv preprint arXiv:1712.03541*, 2017.

[5] R. Pramoditha, "Acquire, understand and prepare the mnist dataset," 2022–06–04.

[6] N. Armanfard, "Ecse 551 - machine learning for engineers lecture 21 — recurrent neural networks (rnns)," December 2022.

[7] P.-L. Pröve, "Squeeze-and-excitation networks," 2017-11-07.

[8] Y. K. Taehong Kwak, Ahram Song, "The effect of the principal component analysis in convolutional neural network for hyperspectral image classification," *The 40th Asian Conference on Remote Sensing (ACRS 2019)*, 2019.

7 Appendix A: Code

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

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1peOHRavaQ9INtuXjxKeM4kah2NzLRute
"""

# Code Citations:
# https://medium.com/@krishna.ramesh.tx/training-a-cnn-to-distinguish-between-
#mnist-digits-using-pytorch-620f06aa9ffa
# https://www.analyticsvidhya.com/blog/2019/10/building-image
#-classification-models-cnn-pytorch/

"""## Import Packages"""

# Commented out IPython magic to ensure Python compatibility.
import pickle
# importing the libraries
import pandas as pd
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
# for reading and displaying images
from skimage.io import imread
```

```
import matplotlib.pyplot as plt
# %matplotlib inline

# for creating validation set
from sklearn.model_selection import train_test_split

# for evaluating the model
from sklearn.metrics import accuracy_score
from tqdm import tqdm

# PyTorch libraries and modules
import torch
from torch.autograd import Variable
from torch.nn import Linear, ReLU, CrossEntropyLoss, Sequential, Conv2d, MaxPool2d, Module, Softmax, BatchNorm2d, Dropout
from torch.optim import Adam, SGD

# Commented out IPython magic to ensure Python compatibility.
import pickle
# importing the libraries
import pandas as pd
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
# for reading and displaying images
from skimage.io import imread
import matplotlib.pyplot as plt
# %matplotlib inline

# for creating validation set
from sklearn.model_selection import train_test_split

# for evaluating the model
from sklearn.metrics import accuracy_score
from tqdm import tqdm

# PyTorch libraries and modules
import torch
from torch.autograd import Variable
from torch.nn import Linear, ReLU, CrossEntropyLoss, Sequential, Conv2d, MaxPool2d, Module, Softmax, BatchNorm2d, Dropout
from torch.optim import Adam, SGD
#pytorch utility imports
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, TensorDataset
from torchvision.utils import make_grid
import pickle
#neural net imports
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
#import external libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import os
import math
# %matplotlib inline

"""## Mount Google Drive
```

```

Make sure you have all of the datasets in your google drive.
"""

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

"""_NOTE: In the code block below, make sure you change the paths accordingly to your drive :)_"""

# Training data and labels
data_x = pickle.load( open('drive/MyDrive/Train.pkl', 'rb'))
data_y = np.genfromtxt('drive/MyDrive/Train_labels.csv', delimiter=',')

# Test data for Kaggle submission
test_x = pickle.load( open( 'drive/MyDrive/Test.pkl', 'rb' ))
test_y = np.genfromtxt('drive/MyDrive/ExampleSubmissionRandom.csv', delimiter=',')

# Sample a random image from the dataset and print its class label
test_sample = 155
plt.imshow(data_x[test_sample][0], cmap='gray', interpolation="bicubic")
plt.show()
print("Label: ", data_y[test_sample][1])
print(data_x.shape[0])

# Image Rotation - NOT WORKING
# from keras_preprocessing.image import ImageDataGenerator, array_to_img, img_to_array, load_img
# #img = load_img('Screenshot_47.png')

# from numpy import expand_dims
# from keras_preprocessing.image import load_img
# from keras_preprocessing.image import img_to_array
# from keras_preprocessing.image import ImageDataGenerator
# import matplotlib.pyplot  as plt

# rotated_train_images = []
# rotated_train_labels = []

# for j in range(train_images.shape[0]):
# #for j in range(10):
#   # load the image
#   plt.figure(figsize=(45,30))

#   # convert to numpy array
#   data = img_to_array(train_images[j])
#   rotated_train_images.append(train_images[j])
#   rotated_train_labels.append(train_labels[j])
#   #print(data.shape)
#   # expand dimension to one sample
#   samples = expand_dims(data, 0)

#   datagen = ImageDataGenerator(featurewise_center=True,
# rotation_range=(0-10),brightness_range=[0,1])
#   # prepare iterator
#   it = datagen.flow(samples, batch_size=1)

#   # generate samples and plot
#   for i in range(6):
#     # define subplot
#     #plt.subplot(330 + 1 + i)
#     # generate batch of images
#     batch = it.next()
```

```
#     # convert to unsigned integers for viewing
#     image = batch[0].astype('uint8')
#     rotated_train_images.append(image)
#     # plot raw pixel data
#     rotated_train_labels.append(train_labels[j])
# print(len(rotated_train_images))
# print(len(rotated_train_labels))

"""## Split and Convert Data To Pytorch Tensors"""

data_x_tmp = []
test_x_tmp = []
for i in data_x:
  data_x_tmp.append(i[0])

for i in test_x:
  test_x_tmp.append(i[0])

data_x = np.asarray(data_x_tmp)
test_x = np.asarray(test_x_tmp)
print("Training Data Shape: ", data_x.shape)
print("Test Data Shape: ", test_x.shape)

# Remove invalid first element and index elements from class labels
data_y = data_y[1:]
data_y = data_y[:,1]
data_y = np.int_(data_y)

test_y = test_y[1:]
test_y = test_y[:,1]
test_y = np.int_(test_y)

train_x, val_x, train_y, val_y = train_test_split(data_x, data_y, test_size = 0.1)
(train_x.shape, train_y.shape), (val_x.shape, val_y.shape)

# Convert training images into torch format
train_x = train_x.reshape(54000, 1, 28, 28)
train_x  = torch.from_numpy(train_x)

test_x = test_x.reshape(10000, 1, 28, 28)
test_x = torch.from_numpy(test_x)

# converting the target into torch format
train_y = train_y.astype(int);
train_y = torch.from_numpy(train_y)

test_y = test_y.astype(int);
test_y = torch.from_numpy(test_y)

# shape of training data
print("Shape of training data and labels")
train_x.shape, train_y.shape

# shape of training data
print("Shape of test data")
test_x.shape, test_y.shape

# converting validation images into torch format
val_x = val_x.reshape(6000, 1, 28, 28)
val_x  = torch.from_numpy(val_x)

# converting the target into torch format
val_y = val_y.astype(int);
```

```
val_y = torch.from_numpy(val_y)

# shape of validation data
val_x.shape, val_y.shape

# converting original dataset to final training set
final_x = data_x.reshape(60000, 1, 28, 28)
final_x = torch.from_numpy(final_x)

final_y = data_y.astype(int);
fianl_y = torch.from_numpy(final_y)

batch_size = 128

# converting training images into torch format
train_tensor_x = torch.Tensor(train_x)
val_tensor_x = torch.Tensor(val_x)
test_tensor_x = torch.Tensor(test_x)
final_tensor_x = torch.Tensor(final_x)

# converting the target into torch format (adding indices)
train_tensor_y = torch.Tensor(train_y)
val_tensor_y = torch.Tensor(val_y)
test_tensor_y = torch.Tensor(test_y)
final_tensor_y = torch.Tensor(final_y)

train_dataset = TensorDataset(train_tensor_x, train_tensor_y) # create your datset
val_dataset = TensorDataset(val_tensor_x,val_tensor_y) # create your datset
test_dataset = TensorDataset(test_tensor_x,test_tensor_y) # create your datset
final_dataset = TensorDataset(final_tensor_x,final_tensor_y) # create your datset

# Final datasets
train_dl = DataLoader(train_dataset, batch_size=batch_size)
val_dl = DataLoader(val_dataset, batch_size=batch_size)
test_dl = DataLoader(test_dataset, batch_size=batch_size)
final_dl = DataLoader(final_dataset, batch_size=batch_size)

# Display the first N batches of data
N = 4
for i, (images, labels) in enumerate(train_dl):
    print('Batch index: ', i)
    print('Batch size: ', images.size())
    print('Batch label: ', labels)
    if(i > N):
      break

"""## Define The Model"""

class ResBlock(nn.Module):
    def __init__(self, in_channels, out_channels, downsample):
        super().__init__()
        if downsample:
            self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=2, padding=1)
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=2),
                nn.BatchNorm2d(out_channels)
            )
        else:
            self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1)
            self.shortcut = nn.Sequential()
```

```python

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, input):
        shortcut = self.shortcut(input)
        input = nn.ReLU()(self.bn1(self.conv1(input)))
        input = nn.ReLU()(self.bn2(self.conv2(input)))
        input = input + shortcut
        return nn.ReLU()(input)

# Squeeze and Excite Layer
# Code citation: https://towardsdatascience.com/introduction-to-squeeze-excitation-
# networks-f22ce3a43348
class SELayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SELayer, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )
    def forward(self, x):
            b, c, _, _ = x.size()
            y = self.avg_pool(x).view(b, c)
            y = self.fc(y).view(b, c, 1, 1)
            return x * y.expand_as(x)

"""
CURRENT BEST MODEL
"""
class Net(nn.Module):
    def __init__(self, resblock, outputs=10):
        super().__init__()
        self.network = nn.Sequential(
        nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=2),
        resblock(32, 32, downsample=False),
        SELayer(32, 2),
        nn.ReLU(),
        nn.AvgPool2d(1, stride=1),

        nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=2),
        resblock(64, 64, downsample=False),
        SELayer(64, 16),
        nn.ReLU(),
        nn.AvgPool2d(1, stride=1),

        nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=0),
        resblock(128, 128, downsample=False),
        SELayer(128, 64),
        nn.ReLU(),
        nn.AvgPool2d(2, stride=2),

        nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=0),
        resblock(256, 256, downsample=False),
        resblock(256, 512, downsample=True),
        SELayer(512, 2),
        nn.ReLU(),
        nn.AvgPool2d(2, stride=2),
```

```python

        resblock(512, 512, downsample=False),
        resblock(512, 1024, downsample=True),
        nn.AdaptiveAvgPool2d(1),
        nn.Flatten(),
        nn.Linear(1024, outputs)
      )

    def forward(self, input):
        y = self.network(input)
        return y

"""# Define Training Functions"""

# Validation Function
# Code Citation: https://medium.com/@krishna.ramesh.tx/training-a-cnn-to-distinguish-between
# -mnist-digits-using-pytorch-620f06aa9ffa
def validate(model, data):
    total = 0
    correct = 0
    for i, (images, labels) in enumerate(data):
        images = images.cuda()
        x = model(images)
        value, pred = torch.max(x,1)
        pred = pred.data.cpu()
        total += x.size(0)
        correct += torch.sum(pred == labels)
    return correct*100./total

# Training Function
# Code Citation: https://medium.com/@krishna.ramesh.tx/training-a-cnn-to-distinguish-between-
# mnist-digits-using-pytorch-620f06aa9ffa
import copy
from torchsummary import summary

def train(numb_epoch=3, dataset=train_dl, lr=1e-3, device="cpu"):
    resnet18 = Net(ResBlock, outputs=10)
    cnn = resnet18.to(torch.device("cuda:0" if torch.cuda.is_available() else "cpu"))
    accuracies = []
    # cnn = create_model().to(device)
    cec = nn.CrossEntropyLoss()
    optimizer = optim.Adam(cnn.parameters(), lr=lr)
    max_accuracy = 0
    for epoch in range(numb_epoch):
        for i, (images, labels) in enumerate(dataset):
            images = images.to(device)
            # labels = labels.to(device)
            labels = labels.type(torch.cuda.LongTensor)
            optimizer.zero_grad()
            pred = cnn(images)
            loss = cec(pred, labels)
            loss.backward()
            optimizer.step()
        accuracy = float(validate(cnn, val_dl))
        accuracies.append(accuracy)
        if accuracy > max_accuracy:
            best_model = copy.deepcopy(cnn)
            max_accuracy = accuracy
            print("Saving Best Model with Accuracy: ", accuracy)
        print('Epoch:', epoch+1, "Accuracy :", accuracy, '%')
    plt.plot(accuracies)
```

```
    return best_model

"""## Train on Training Data"""

if torch.cuda.is_available():
    device = torch.device("cuda:0")
else:
    device = torch.device("cpu")
    print("No Cuda Available")

device

cnn = train(5, dataset=train_dl, device=device)

"""## Train on Entire Dataset"""

# Train the final model
cnn = train(2, dataset=final_dl, device=device)

"""## Predict Testing Set"""

def predict_final(model, data):
    y_pred = []
    for i, (images, labels) in enumerate(data):
        images = images.cuda()
        x = model(images)
        value, pred = torch.max(x, 1)
        pred = pred.data.cpu()
        y_pred.extend(list(pred.numpy()))
    return np.array(y_pred)

y_pred = predict_final(cnn, test_dl)

"""## Save Testing Set Predictions (CSV)"""

y_final = np.zeros((len(y_pred), 2))
for i in range(len(y_pred)):
  y_final[i, :] = np.array([i, int(y_pred[i])])

# save array into csv file
np.savetxt("final.csv",
           y_final,
           '%i,%i',
           delimiter = ",")

```