ECSE-551 Mini-project 3 Report of Group 23: **Developing Convolutional Neural Network for Modeling Image Data Sets**

Ali Raoofian[a], Asal Zabetian Hosseini[b], Alaa S. Abdelgawad[c] and Peyman Moeini[d]

[a]*Department of Mechanical Engineering and Centre for Intelligent Machines, McGill University, Montréal, Canada*
[b]*Department of Electrical and Computer Engineering, McGill University, Montréal, QC, Canada*
[c]*Department of Biomedical Engineering, McGill University, Montréal, QC, Canada*
[d]*Department of Electrical and Computer Engineering, McGill University, Montréal, QC, Canada*

ARTICLE INFO

Keywords:
Neural Network
Deep learning
image processing
Computer vision

ABSTRACT

In this project, two neural network-based approaches are deployed to predict the output labels of a modified Fashion-MNIST data set. In more details, the convolutional networks, pooling, and nonlinear functions (i.e. Softmax, Tanh) functions were deployed to predict the output labels. Furthermore, the predefined neural network layers in the Pytorch library were utilized to create a model using the given data sets under the condition that the pretrained weighting values were False. Finally, the results of both approaches are demonstrated and evaluated. The approach with the maximum achieved accuracy was submitted on the Kaggle competition.

1. Introduction

A Neural Network consists of algorithm layers that learn to perform specific or general tasks through analyzing a training data set. In other words, the Neural Network method learns the relationship between ab input data and the output labels [2]. The relationships can be nonlinear and complex which makes them great tools for various applications such as predictions in the fields of control systems, medical diagnosis, natural language processing, and computer vision. A famous lass of Neural Network is the Convolutional Neural Network (CNN) which can be separated into input layers, output layers, and the hidden layers. The hidden layers usually consist of convolutional layers including ReLU layers, pooling layers, and fully connected layers. The convolutional layers apply a convolution operation to the input which results in the passage of the results to the next layer [6]. The pooling combines the outputs of clusters of neurons into a single neuron in the next layer. Furthermore, fully connected layers connect every neuron in one layer to every neuron in the next layer.

In this report, the Data Sets Description Section describes the given data sets. In the Proposed Approach Section, the deployed layers to create CNN are explained and three CNN with different layer lengths are discussed. In the Result Section, the performance of the proposed CNNs are analyzed based on variation of different hyper-parameters. The Additional Remark Section describes the performance of the predefined NN in Pythorch library. Finally, in the Conclusion section, the summary of the result are presented.

2. Data Sets Description

In this section, the data set utilized in the Mini-Project 3 is described in details. To develop the NN model, the modified version of the Fashion-MNIST dataset is constructed. The dataset consists of 60000

ali.raoofian@mail.mcgill.ca (A. Raoofian); asal.zabetian-hosseini@mail.mcgill.ca (A. Zabetian Hosseini); alaa.abdelgawad@mail.mcgill.ca (A. S. Abdelgawad); Peyman.moeini@mail.mcgill.ca (P. Moeini)
STUDENT ID(s): 260856711 (A. Raoofian); 260859392 (A. Zabetian Hosseini); 260912527 (A. S. Abdelgawad); 260950246 (P. Moeini)

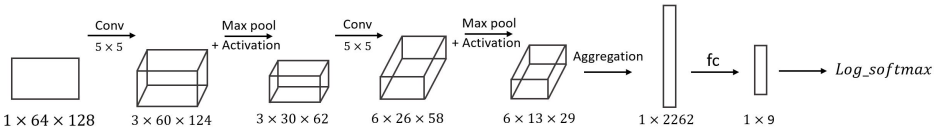

Figure 1: Model 1 layer scheme

training image data. Each image contains three articles and the goal is to predict the total price of all articles shown in the image. The articles are categorized from the least to the most expensive: T-shirt/top ($1), Trouser ($ 2), Pullover($ 3), Dress ($ 4), and Coat($ 5). The total price of each image is changing from the least expensive ($ 5) to the most expensive ($ 13). Consequently, the images are labeled in nine class classifications. The size of each input image is 64,124,1 pixels. Furthermore, the input channel of the NN is one and the final output is 9. Based on the training data set, the NN models are designed. Consequently, the developed models are used to predict the output labels for the test data.

3. Proposed Approach

In this section, the proposed approach to develop the multi-layers NN is explained in details. Three basic NN models are designed for this study which are Model 1 and Model 2. All these models contains convolution neural network (CNN), pooling max, activation function, and Fully Connected (FC) neural network. The CNN is used to filter the image and makes it deeper [9]. The pooling max is deployed to reduce the size [10]. The activation function models non-linearity of the data in the NN [11] . Different activation functions can be utilized; some of the most common activation functions are the rectified linear unit (Relu) and the tangent hyperbolic (Tanh) functions [12] . The Relu function changes between zero to infinity and both the function and its derivative are monotonic. The other function presented is the Tanh. The advantage of the Tanh is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph and it is differentiable. In addition, the tanh function is monotonic while its derivative is not monotonic [14]. In this report both of these functions are studied. The FC is deployed in the final layers for further modeling. Finally, to calculate the loss function and minimize losses, two loss functions were incorporated which were (1) Cross Entropy Loss and (2) Negative Log-Likelihood Loss (NLLLoss). Both of these loss methods achieved similar outcomes as per the experiments conducted. Additionally, the developed model could be processed using two different processing architectures, (1) CPU and (2) GPU. The main difference between CPU and GPU architecture is that a CPU where as a GPU is designed to quickly parallel-process high-resolution images and videos concurrently. In more details, a CPU is designed to handle a wide-range of tasks quickly (as measured by CPU clock speed), but are limited in the concurrency of tasks that can be running unlike GPUs that perform parallel operations. Interestingly, the GPUs are commonly used for non-graphical tasks such as machine learning and scientific computation. In the following, the studied models are discussed:

- Model 1 is a very simple short model and consists of 6 basic layers as shown in figure 1. Layer 1 is the CNN $5 * 5$ with 3 filters. Layer 2 is the max pooling with the activation function. An additional 3 filter based CNN is added in Layer 3 and max pooling and activation function in Layer 4 followed by aggregation. In addition, a fully connected layer is added with activation to predict the output category of the input data in Layer 6.

- Model 2 is more complicated and deeper than the Model 1 as shown in figure 2. This model has 9 main layers consists of CNN, max pooling, and the activation and then three fully connected NN layers

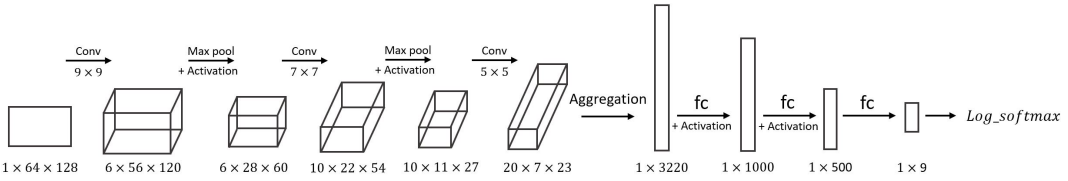

Figure 2: Model 2 layer scheme

with activation functions are added. Finally, similar to Model 1, the log Softmax function is deployed to decide on the output function.

- Model 3 is the modified version of the LeNet-5 layers with 7 main layers. In this model, Relu activation function substituted all the Tanh activation functions and additional fully connected layer is added to the final layer to have with three fully connected layers.

4. Results

In this section, the developed code to implemented the proposed approaches is discussed and the simulation results and the proposed model performances is evaluated. To study the performance of the proposed models the following steps were taken: 1) The provided 60000 images were divided to 50000 training data and 10000 validation data, 2) both data sets were analyzed with respects to their input and output sizes, and 3) the images were transformed and normalized using image transform, Dataset, and Dataloader classes in Python. After the pre-processing was completed, the loaded data was ready for training. In the train class, the data was selected based on the batching size and was transferred to the processing device with architecture options of GPU or CPU. The transferred data was trained using the network class in which the NN layers and the optimizer were defined. After this step, the output of the data was predicted and the actual label was compared with the predicted value to calculate the loss for the training data. Then the test class was deployed to calculate the accuracy of the validation data set based on the developed model in the training class in the each epochs.

4.1. Proposed Models Performance in Different Epochs

To study the effects of the hyper-parameters on both proposed models, Models 1 and 2, the accuracy of the validation data set for different epochs up to 250 and Tanh and Relu activation functions were evaluated. Table 1 shows mean time and accuracy of Model 1 as a short NN and Model 2 as a long model with respect to the number of epochs. The SGD optimizer with the learning rate of 0.001 and the momentum of 0.9 is deployed for all the models. As expected, the more layers the model has, the better accuracy it provides. Additionally, in most cases increasing the number of epochs does not provide any improvement which could be caused by overfiting problem.

Figure 3 shows the loss and accuracy of the validation set when Model 3 is deployed and as expected, by increasing the number of epochs the loss decreases and the accuracy improves.

4.2. Model 2 Performance with Different Learning Rates and Momentum

In this study, We use model 2 with Relu function 20 epochs to study the impacts of the learning rate and the momentum on the model performance. Table 2 shows the performance of the Model 2 when the learning rate changed between the 0.1 to 0.00001 and the momentum changed from 0.9 to 0.3. As it can be seen, the accuracy improves when the momentum is decreased. Additionally, the larger learning rates that

Table 1
The accuracy and mean time results of the models based on the number of epochs

		Number of Epochs						
Model	No. Layers	5	20	50	100	200	250	Mean Time (min)
Model 1 (Short+Relu)	6	25%	27%	28%	28%	29%	29%	10.73
Model 1 (Short+Tanh)	6	19%	22%	25%	N/A	N/A	N/A	12.35
Model 2 (Long+Relu)	9	35%	**38%**	33%	N/A	N/A	N/A	16.12
Model 2 (Long+Tanh)	9	29%	33 %	32 %	28%	N/A	N/A	17.57
Model 3 (Modified LeNet-5)	7	59 %	**68** %	N/A	N/A	N/A	N/A	15.3 %

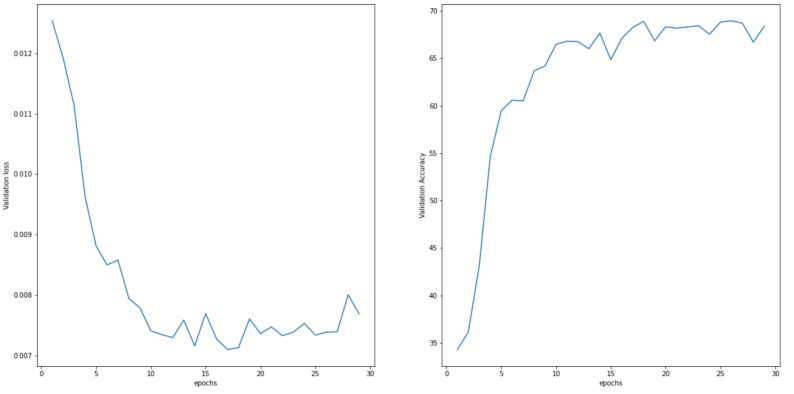

Figure 3: Model 3 Validation Loss and Accuracy

Table 2
The mean time and accuracy results for the Model 2 with (Long+Relu+NIIloss) when the learning rate and momentum change

	Momentum= 0.3		Momentum= 0.6		Momentum= 0.9	
Learning Rate	Accuracy (%)	Time (min)	Accuracy (%)	Time (min)	Accuracy (%)	time (min)
0.01	**55**	**17.38**	52	17.33	49	18.3
0.0001	53	17.28	52	16.78	54	17.02
0.00001	46	17.41	47	16.82	47	17.51

was experimented provided better accuracies. Therefore, in this model, the best accuracy is achieved when the momentum is 0.3 and the learning rate is 0.01.

4.3. GPU vs CPU

To study the impact of the GPU or CPU, the GPU and the CPU running times were compared for identical models (Model 2 with Relu, 0.1 learning rate, and momentum of 0.3). Table 3 demonstrates the GPU and CPU time for one epochs. It can be observed that employing GPU as the device can decrease the running time significantly from 253.8 minutes for CPU to 17.1 minutes for GPU. The results is expected since the GPU is designed to quickly process high-resolution images. Additionally, it can perform parallel operation on multiple sets of data.

5. Additional remarks

Additional to the proposed NN models for this project, the accuracy of famous predefined models without using their pretrained weights was evaluated. These models consisted of the LeNet-5, ResNet18, ResNet156,

Table 3
A comparison between the running time for the Model 2 when the device is connected to GPU and CPU

Model= Model 3	GPU Time	CPU Time
One Epoch time	17.1	253.8

Table 4
Predefined Neural Network Models

Model	Epoch	Learning Rate	Momentum	Optimizer	Loss	Batch-size	Accuracy (%)
LeNet−5	20	0.001	0.6	Adam	cross entropy loss	128	71
ResNet18	50	0.001	0.9	SGD	Cross entropy loss	92	
ResNet152	30	N/A	N/A	Adam	NLLLoss	64	88.4
	60	N/A	N/A	Adam	NLLLoss	64	90.4
	90	N/A	N/A	Adam	NLLLoss	64	91.6
DenseNet	10	0.001	0.9	SGD	Cross entropy loss	64	90.9
	20	0.001	0.9	SGD	Cross entropy loss	64	96
	40	**0.001**	**0.9**	**SGD**	**Cross entropy loss**	**64**	**98**
	40	N/A	NA	**Adam**	**Cross entropy loss**	**64**	97.5

and denseNet [18]. The simplest model was LeNet-5 with 7 layers followed by ResNet18 with 18 layers, DenseNet121 with 121 layers, and ResNet 152 with 152 layers [19] [20][21]. The DenseNet consisted of two blocks, (1) a block with batch normalization, Relu, CNN and (2) Batch, CNN, and Averaging pooling. These models were defined for the RBG images with 3 channels and the minimum size of the images were 224*224 pixels. However, the training dataset in this project only had 1 channel and the size of the images were 64*124. As a result, to use these models, the images in the dataset were converted to RBG image and were resized to 224*224. Table 4 shows the accuracy of the models with respect to the type of optimizer (Adam or SGD with learning rates and momentum), and the number of epochs. The results indicate that by increasing the number of epochs the accuracy is increased in both DenseNet and ResNet. However depending on the model, after some epochs, the model is prone to over-fitting. Additionally, the results showed that the DenseNet121 provided the best accuracy for our test data set with 98% after 40 epochs. The developed DenseNet121 model was submitted to the Kaggle competition in which it provided similar accuracy results.

6. Discussion and Conclusion

To recapitulate, in Mini Project 3, several NN layers were developed to model the modified version of Fashion-MNIST dataset. The effects of the hyper-parameters (epochs, learning rate, momentum, batching size), the activation function (Relu verses Tanh), and the device were studied on the proposed models. The experimented results showed a NN with more layers provided a better accuracy; In addition, the Relu function provided a better accuracy compared to Tanh for multiclass classification. Furthermore, by increasing the number of epochs the prediction accuracy was improved; however, depending on the dataset, the models were subject to over-fitting with higher number of epochs. The other hyper-parameters such as the learning rates and momentum had different impacts depending on the layers. According to our experiments, the accuracy was improved by decreasing the momentum and increasing the learning rate. Finally, the predefined neural networks in the Pytorch library were studied to achieve the best accuracy. Among all the considered neural networks, the DenseNet121 provided the best accuracy which was submitted in the Kaggle Competition with accuracy of 98%.

7. Statement of Contributions

All the group members were involved equally in developing the code and writing the report.

A. Appendix: Codes

```
# -- coding: utf-8 --
"""Mini-project 3.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1191rzl3JcSHDCUGzWqxWY31GlEOPpatj
"""

# Commented out IPython magic to ensure Python compatibility.
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import transforms
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from PIL import Image
import torchvision.models as models

import numpy as np
from sklearn.preprocessing import OneHotEncoder
import matplotlib.pyplot as plt
import pickle
import time
import pandas as pd
from google.colab import files

from google.colab import drive
drive.mount('/content/gdrive')
# %cd '/content/gdrive/MyDrive/mini-proj3_data/'

# ////////////////////////////////////////////////////////////////////////////
# /////////////////////////// Functions and Classes //////////////////////////
# ////////////////////////////////////////////////////////////////////////////

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

    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



class Net1(nn.Module):
    # This part defines the layers
    def __init__(self):
        super(Net1, self).__init__()

        self.conv1 = nn.Conv2d(1, 3, kernel_size=5)
        self.conv2 = nn.Conv2d(3, 6, kernel_size=5)

        self.fc1 = nn.Linear(2262, 9)

    def forward(self, x):

        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))

        # This layer is an imaginary one. It simply
        #states that we should see each member of x
        # as a vector of 320 elements, instead of a tensor of
        # 20x4x4 (Notice that 2044=320)
        x = x.view(-1, 2262)

        # Feedforeward layers. Remember that fc1 is a layer
        #that goes from 320 to 50 neurons
        x = (self.fc1(x))

        # We should put an appropriate activation for the output layer.
        return F.log_softmax(x)
```

```
class Net2(nn.Module):
    # This part defines the layers
    def __init__(self):
        super(Net2, self).__init__()

        self.conv1 = nn.Conv2d(1, 3, kernel_size=5)
        self.conv2 = nn.Conv2d(3, 6, kernel_size=5)

        self.fc1 = nn.Linear(2262, 9)

    def forward(self, x):

        x = F.tanh(F.max_pool2d(self.conv1(x), 2))
        x = F.tanh(F.max_pool2d(self.conv2(x), 2))

        # This layer is an imaginary one. It simply states
        #that we should see each member of x
        # as a vector of 320 elements, instead of a
        # tensor of 20x4x4 (Notice that 2044=320)
        x = x.view(-1, 2262)

        # Feedforeward layers. Remember that fc1 is
        # a layer that goes from 320 to 50 neurons
        x = (self.fc1(x))

        # We should put an appropriate activation for the output layer.
        return F.log_softmax(x)


class Net3(nn.Module):
    # This part defines the layers
    def __init__(self):
        super(Net3, self).__init__()

        self.conv1 = nn.Conv2d(1, 6, kernel_size=9)
        self.conv2 = nn.Conv2d(6, 10, kernel_size=7)
        self.conv3 = nn.Conv2d(10, 20, kernel_size=5)

        self.fc1 = nn.Linear(3220, 1000)
        self.fc2 = nn.Linear(1000, 300)
        self.fc3 = nn.Linear(300, 9)

    def forward(self, x):

        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = self.conv3(x)
```

```
        # This layer is an imaginary one. It simply states that we should
        # see each member of x
        # as a vector of 320 elements, instead of a tensor of
        #20x4x4 (Notice that 2044=320)
        x = x.view(-1, 3220)

        # Feedforeward layers. Remember that fc1 is a layer
        #that goes from 320 to 50 neurons
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))

        # Output layer
        x = self.fc3(x)

        # We should put an appropriate activation for the output layer.
        return F.log_softmax(x)



class Net4(nn.Module):
    # This part defines the layers
    def __init__(self):
        super(Net4, self).__init__()

        self.conv1 = nn.Conv2d(1, 6, kernel_size=9)
        self.conv2 = nn.Conv2d(6, 10, kernel_size=7)
        self.conv3 = nn.Conv2d(10, 20, kernel_size=5)

        self.fc1 = nn.Linear(3220, 1000)
        self.fc2 = nn.Linear(1000, 300)
        self.fc3 = nn.Linear(300, 9)

    def forward(self, x):

        x = F.tanh(F.max_pool2d(self.conv1(x), 2))
        x = F.tanh(F.max_pool2d(self.conv2(x), 2))
        x = self.conv3(x)

        # This layer is an imaginary one. It simply states
        #that we should see each member of x
        # as a vector of 320 elements, instead of a
        #tensor of 20x4x4 (Notice that 2044=320)
        x = x.view(-1, 3220)

        # Feedforeward layers. Remember that fc1 is a
        #layer that goes from 320 to 50 neurons
```

```
        x = F.tanh(self.fc1(x))
        x = F.tanh(self.fc2(x))

        # Output layer
        x = self.fc3(x)

        # We should put an appropriate activation for the output layer.
        return F.log_softmax(x)



def train(epoch):
  network.train()
  for batch_idx, (data, target) in enumerate(train_loader):
    data = data.to(device)
    target = target.to(device)
    optimizer.zero_grad()
    output = F.log_softmax(network(data))
    loss = F.nll_loss(output, target) #negative log liklhood loss
    loss.backward()
    optimizer.step()
    if batch_idx % 20 == 0:
      print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
        epoch, batch_idx * len(data), len(train_loader.dataset),
        100. * batch_idx / len(train_loader), loss.item()))
      train_losses.append(loss.item())
      train_counter.append(
        (batch_idx*batchsize) + ((epoch-1)*len(train_loader.dataset)))
      torch.save({
                  'Lepoch': epoch,
                  'network_state_dict':network.state_dict(),
                  'optimizer_state_dict': optimizer.state_dict(),
                  'Ltotal_time': total_time,
                  'Ltrain_losses': train_losses,
                  'Ltrain_counter':train_counter,
                  'Ltest_losses':test_losses,
                  'Lloss': loss}, './trainsaves/model.pth')
      # torch.save(optimizer.state_dict(), './trainsaves/optimizer.pth')


def test():
  network.eval()
  test_loss = 0
  correct = 0
  with torch.no_grad():
    test_loader = valid_loader
    for data, target in test_loader:
```

```
        data = data.to(device)
        target = target.to(device)
        output = F.log_softmax(network(data))
        test_loss += F.nll_loss(output, target, size_average=False).item()
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).sum()
    test_loss /= len(test_loader.dataset)
    test_losses.append(test_loss)
    print('\nValid_set: Avg. loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
      test_loss, correct, len(test_loader.dataset),
      100. * correct / len(test_loader.dataset)))

# ////////////////////////////////////////////////////////////////////////////
# ////////////////////// Inputs and Initialization ///////////////////////////
# ////////////////////////////////////////////////////////////////////////////

batchsize = 64        # Specify the batch size
if_validation = True  # Specify if you want to
#consider a part of training data as validation set
if_shuffle = True     # Specify if you want to
#shuffle the training data before training process

learningrate = 0.01
moment = 0.9
num_of_epochs = 50
random_seed = 0

if if_validation:
    train_idx = range(0,50000)
    valid_idx = range(50000,60000)
else:
    train_idx = None
# ////////////////////////////////////////////////////////////////////////////

torch.manual_seed(random_seed)
torch.cuda.manual_seed(random_seed)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

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


train_dataset = MyDataset('./Train.pkl', './TrainLabels.csv',
                          transform=img_transform, idx=train_idx)
```

```
train_loader = DataLoader(train_dataset,
                          batch_size=batchsize, shuffle=if_shuffle)

if if_validation:
  valid_dataset = MyDataset('./Train.pkl', './TrainLabels.csv',
                            transform=img_transform, idx=valid_idx)
  valid_loader = DataLoader(valid_dataset,
                            batch_size=batchsize, shuffle=if_shuffle)
else:
  valid_loader = train_loader

# ////////////////////////////////////////////////////////////////////////
# ////////////////////////////// Learning Stage ///////////////////////////
# ////////////////////////////////////////////////////////////////////////

network = Net1().to(device)

# optimizer = optim.SGD(network.parameters(), lr=learningrate, momentum=momen
optimizer = optim.Adam(network.parameters())

total_time = []
train_losses = []
train_counter = []
test_losses = []

for epoch in range(1, num_of_epochs+1):
  start_time = time.clock()
  train(epoch)
  run_time = (time.clock() - start_time)
  total_time.append(run_time)

  test()

mean_time = sum(total_time) / len(total_time)
print('The_mean_run_time_for_each_epoch_was:', mean_time, 'seconds')

# ////////////////////////////////////////////////////////////////////////
# ////////////////////////// Continue saved work //////////////////////////
# ////////////////////////////////////////////////////////////////////////


network = Net1().to(device)

# optimizer = optim.SGD(network.parameters(), lr=learningrate, momentum=momen
optimizer = optim.Adam(network.parameters())

# ////////////////////////////////////////////////////////////////////////
```

```
checkpoint = torch.load('./trainsaves/model.pth')

network.load_state_dict(checkpoint['network_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
lastepoch = checkpoint['Lepoch']
loss = checkpoint['Lloss']
total_time = checkpoint['Ltotal_time']
train_losses = checkpoint['Ltrain_losses']
train_counter = checkpoint['Ltrain_counter']
test_losses = checkpoint['Ltest_losses']

# ////////////////////////////////////////////////////////////////////////////

for epoch in range(lastepoch+1, lastepoch+1+2):
  start_time = time.clock()
  train(epoch)
  run_time = (time.clock() - start_time)
  total_time.append(run_time)

  test()

mean_time = sum(total_time) / len(total_time)
print('The mean run time for each epoch was:', mean_time, 'seconds')

# ////////////////////////////////////////////////////////////////////////////
# /////// Prediction of test data and download the labels in CSV format //////
# ////////////////////////////////////////////////////////////////////////////

# Here, the values of "TestdummyLabels.csv" is not used. Just put a random
#file with 10000 labels (it is important that the size be 10000)
testdataset = MyDataset('./Test.pkl', './TestdummyLabels.csv',
                        transform=img_transform, idx=None)
testloader = DataLoader(testdataset, shuffle=False)

j=1
network.eval()
y_guess = []
with torch.no_grad():
  for data, target in testloader:
        data = data.to(device)
        outputarray = F.log_softmax(network(data))
        outputlabel = outputarray.data.max(1, keepdim=True)[1]
        if j % 500 == 0:
          # print(type(outputlabel.item()), type(outputlabel))
          print('The prediction is done up to label index:', j)
        j = j+1
```

```
        y_guess.append(outputlabel.item())

y_guess2 = [x+5 for x in y_guess] # This line adds the
#previously subtracted 5 to the labels
y_guess_df = pd.DataFrame(y_guess2, columns=['class'])
y_guess_df.to_csv('KaggleLabels.csv', columns=['class'])
files.download("KaggleLabels.csv")
```

References

[1] M. Volpi and D. Tuia, "Dense semantic labeling of subdecimeter resolution images with convolutional neural networks," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 55, no. 2, pp. 881–893, 2016.

[2] L. Liu, C. Shen, and A. van den Hengel, "The treasure beneath convolutional layers: Cross-convolutional-layer pooling for image classification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 4749–4757.

[3] H. Lee and H. Kwon, "Going deeper with contextual cnn for hyperspectral image classification," *IEEE Transactions on Image Processing*, vol. 26, no. 10, pp. 4843–4855, 2017.

[4] Yuchi Huang, Xiuyu Sun, Ming Lu, and M. Xu, "Channel-max, channel-drop and stochastic max-pooling," in *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2015, pp. 9–17.

[5] S. Y. Yadhav, T. Senthilkumar, S. Jayanthy, and J. J. A. Kovilpillai, "Plant disease detection and classification using cnn model with optimized activation function," in *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)*, 2020, pp. 564–569.

[6] M. Khalid, J. Baber, M. K. Kasi, M. Bakhtyar, V. Devi, and N. Sheikh, "Empirical evaluation of activation functions in deep convolution neural network for facial expression recognition," in *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*, 2020, pp. 204–207.

[7] D. Misra, "Mish: A self regularized non-monotonic neural activation function," *arXiv preprint arXiv:1908.08681*, 2019.

[8] F. Sultana, A. Sufian, and P. Dutta, "Advancements in image classification using convolutional neural network," in *2018 Fourth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*. IEEE, 2018, pp. 122–129.

[9] G. Wang and J. Gong, "Facial expression recognition based on improved lenet-5 cnn," in *2019 Chinese Control And Decision Conference (CCDC)*, 2019, pp. 5655–5660.

[10] D. C. Liyanage, R. Hudjakov, and M. Tamre, "Hyperspectral imaging methods improve rgb image semantic segmentation of unstructured terrains," in *2020 International Conference Mechatronic Systems and Materials (MSM)*, 2020, pp. 1–5.

[11] S. Mo and M. Cai, "Deep learning based multi-label chest x-ray classification with entropy weighting loss," in *2019 12th International Symposium on Computational Intelligence and Design (ISCID)*, vol. 2, 2019, pp. 124–127.