IF-5-OT2

Machine Learning

Haneul Kim, 04027151

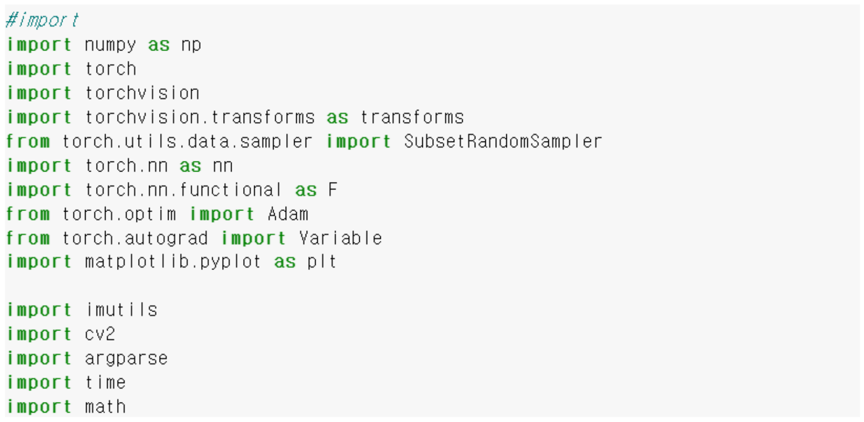
Hayeong Lee, 04027155

1. Introduction

This project aims to implement a face recognition system that can detect face/non-face in images. This system is based on deep learning, Convolutional Neural Networks (CNN). To build this system, we went through several stages. First, we preprocessed our dataset. Then, we implemented a Convolutional Neural Network which data can pass. After that, we made a detector that can apply CNN. With this detector, we can explore our image in a small size using Sliding Window, Non-Max suppression, and Scaling.

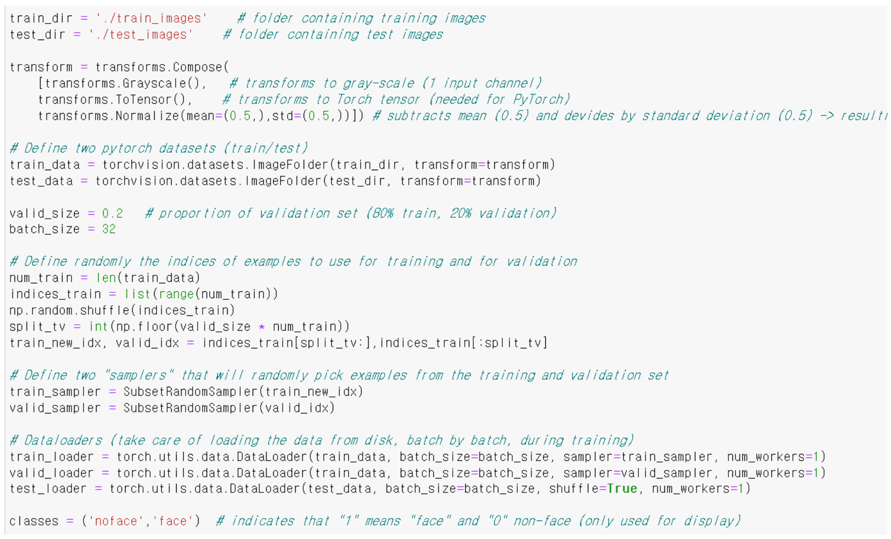
2. Importing and Loading data

Before implementing, we import some libraries such as ‘torch’, ‘numpy’, ‘matplotlib’, and so on. And for the dataset, we preprocessed and loaded the data. For preprocessing, we use the ‘torchvision’ library to transform the image into a tensor, make it in grayscale, and normalize it. And we set the validation set size as 0.2 and batch size as 32, then load data in ‘train_loader’, ‘valid_loader’, and ‘test_loader’. The labels for data are set to classes, ‘noface’ and ‘face’.



```
#import
import numpy as np
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import Adam
from torch.autograd import Variable
import matplotlib.pyplot as plt

import imutils
import cv2
import argparse
import time
import math
```

Figure 1: Importing libraries



```python
train_dir = './train_images'    # folder containing training images
test_dir = './test_images'    # folder containing test images

transform = transforms.Compose(
    [transforms.Grayscale(),   # transforms to gray-scale (1 input channel)
    transforms.ToTensor(),    # transforms to Torch tensor (needed for PyTorch)
    transforms.Normalize(mean=(0.5,),std=(0.5,))]) # subtracts mean (0.5) and devides by standard deviation (0.5) -> resulti

# Define two pytorch datasets (train/test)
train_data = torchvision.datasets.ImageFolder(train_dir, transform=transform)
test_data = torchvision.datasets.ImageFolder(test_dir, transform=transform)

valid_size = 0.2   # proportion of validation set (80% train, 20% validation)
batch_size = 32

# Define randomly the indices of examples to use for training and for validation
num_train = len(train_data)
indices_train = list(range(num_train))
np.random.shuffle(indices_train)
split_tv = int(np.floor(valid_size * num_train))
train_new_idx, valid_idx = indices_train[split_tv:],indices_train[:split_tv]

# Define two "samplers" that will randomly pick examples from the training and validation set
train_sampler = SubsetRandomSampler(train_new_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# Dataloaders (take care of loading the data from disk, batch by batch, during training)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, sampler=train_sampler, num_workers=1)
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler, num_workers=1)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True, num_workers=1)

classes = ('noface','face')  # indicates that "1" means "face" and "0" non-face (only used for display)
```

Figure 2: Preprocessing the data

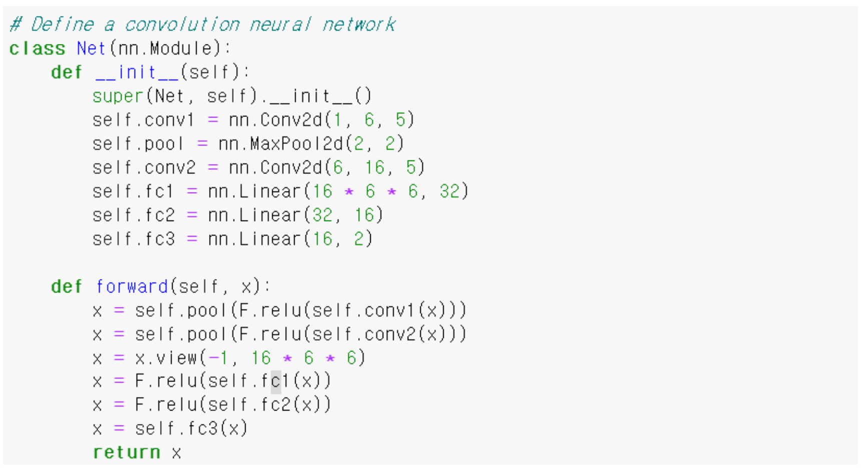
3. Creation of a face/non-face classifier

3.1. Implementing CNN

After preparing the dataset, we made a neural network that can recognize the image. For implementing our CNN, we define a class, Net(nn.Module) using 'torch.nn.Module', and two methods inside the class.

The first method is '__init__(self)'. In this method, we make our network layers. We use two convolutional layers, one pooling layer, and three fully connected layers.

The second method is 'forward(self, x)' which can pass data through the neural network. In the 'forward' function, we use two pooling layers after activating the ReLU function with each convolutional layer. Then, we reshape the tensor using 'view' and apply two ReLU activation functions with each fully connected layer.



```python
# Define a convolution neural network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 6 * 6, 32)
        self.fc2 = nn.Linear(32, 16)
        self.fc3 = nn.Linear(16, 2)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 6 * 6)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Figure 3: CNN

3.2 Increasing Accuracy

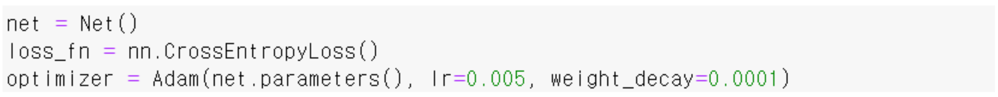
To increase the accuracy of our model, we tried to do several different methods such as changing the loss function, optimizer, and learning rate.

First, we changed the loss function to the ‘MSE’. Contrary to our expectation, its accuracy result was not much better than the ‘CrossEntropyLoss’ function. So, we decided to use the ‘CrossEntropyLoss’ function again which measures the difference between distributions of labels and model output. ‘CrossEntropyLoss’ function can avoid the problem of slowing down the learning rate, whereas ‘MSE’ function cannot avoid that problem. It was the reason why our loss function is better than ‘MSE’.

Secondly, we set our optimizer as ‘Adam’, which combines ‘RMSprop’ and ‘Momentum’ instead of using ‘SGD’ optimizer. ‘Adam’ can improve both the direction and size of learning.

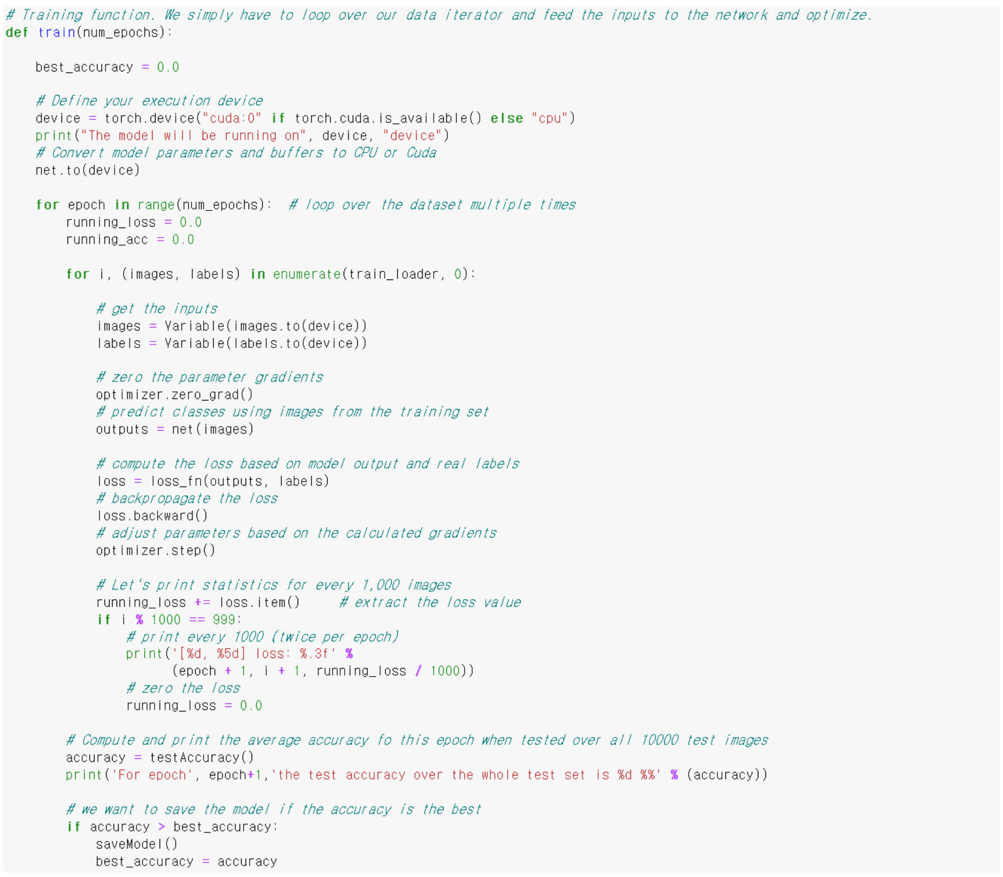
Lastly, we changed the learning rate values from 0.001 to the 0.005, and set a weight decay to the 0.0001 for avoiding overfitting.

We use this loss function and optimizer in our ‘train’ function.



```python
net = Net()
loss_fn = nn.CrossEntropyLoss()
optimizer = Adam(net.parameters(), lr=0.005, weight_decay=0.0001)
```

Figure 4: Loss function and Optimizer

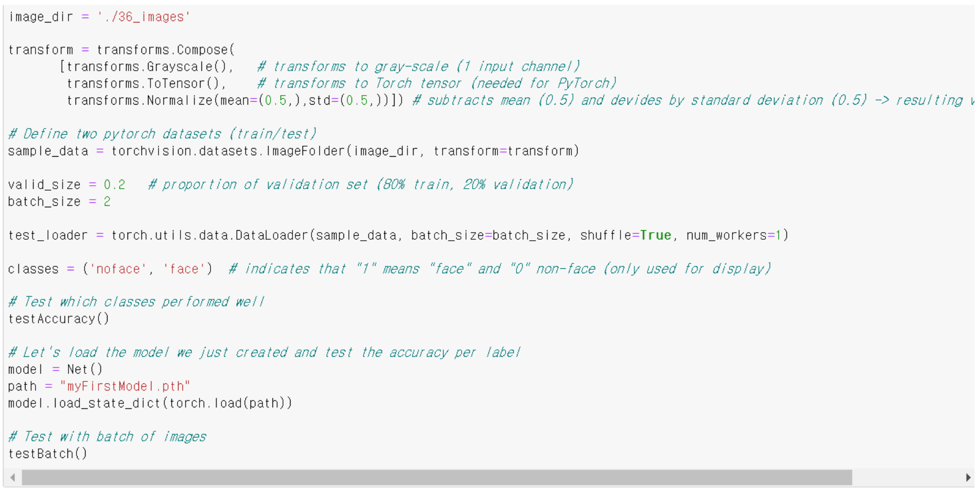


```python
# Training function. We simply have to loop over our data iterator and feed the inputs to the network and optimize.
def train(num_epochs):

    best_accuracy = 0.0

    # Define your execution device
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print("The model will be running on", device, "device")
    # Convert model parameters and buffers to CPU or Cuda
    net.to(device)

    for epoch in range(num_epochs):  # loop over the dataset multiple times
        running_loss = 0.0
        running_acc = 0.0

        for i, (images, labels) in enumerate(train_loader, 0):

            # get the inputs
            images = Variable(images.to(device))
            labels = Variable(labels.to(device))

            # zero the parameter gradients
            optimizer.zero_grad()
            # predict classes using images from the training set
            outputs = net(images)

            # compute the loss based on model output and real labels
            loss = loss_fn(outputs, labels)
            # backpropagate the loss
            loss.backward()
            # adjust parameters based on the calculated gradients
            optimizer.step()

            # Let's print statistics for every 1,000 images
            running_loss += loss.item()     # extract the loss value
            if i % 1000 == 999:
                # print every 1000 (twice per epoch)
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 1000))
                # zero the loss
                running_loss = 0.0

        # Compute and print the average accuracy fo this epoch when tested over all 10000 test images
        accuracy = testAccuracy()
        print('For epoch', epoch+1,'the test accuracy over the whole test set is %d %%' % (accuracy))

        # we want to save the model if the accuracy is the best
        if accuracy > best_accuracy:
            saveModel()
            best_accuracy = accuracy
```

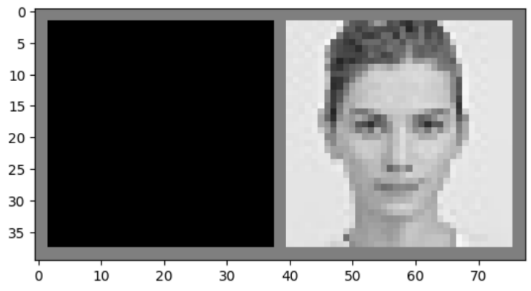
Figure 5: train method

3.3. Testing with sample image

For testing our model, we use our own image, which sizes are 36*36. We made another folder, '36_images', and inside the folder, there are two folders, '0' for a non-face image and '1' for a face image. Then, after the process of preprocessing the data, we test the accuracy and batches. The result is shown below.



```python
image_dir = './36_images'

transform = transforms.Compose(
       [transforms.Grayscale(),   # transforms to gray-scale (1 input channel)
        transforms.ToTensor(),    # transforms to Torch tensor (needed for PyTorch)
        transforms.Normalize(mean=(0.5,),std=(0.5,))]) # subtracts mean (0.5) and devides by standard deviation (0.5) -> resulting v

# Define two pytorch datasets (train/test)
sample_data = torchvision.datasets.ImageFolder(image_dir, transform=transform)

valid_size = 0.2   # proportion of validation set (80% train, 20% validation)
batch_size = 2

test_loader = torch.utils.data.DataLoader(sample_data, batch_size=batch_size, shuffle=True, num_workers=1)

classes = ('noface', 'face')  # indicates that "1" means "face" and "0" non-face (only used for display)

# Test which classes performed well
testAccuracy()

# Let's load the model we just created and test the accuracy per label
model = Net()
path = "myFirstModel.pth"
model.load_state_dict(torch.load(path))

# Test with batch of images
testBatch()
```



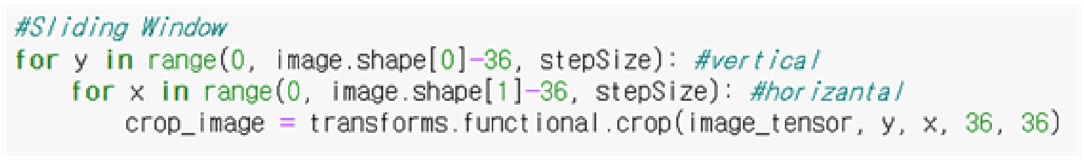
```
Real labels:  noface  face
Predicted:  noface noface
```

Figure 6: Test with 36*36 images

4. Implementing a Detector

4.1. Sliding window

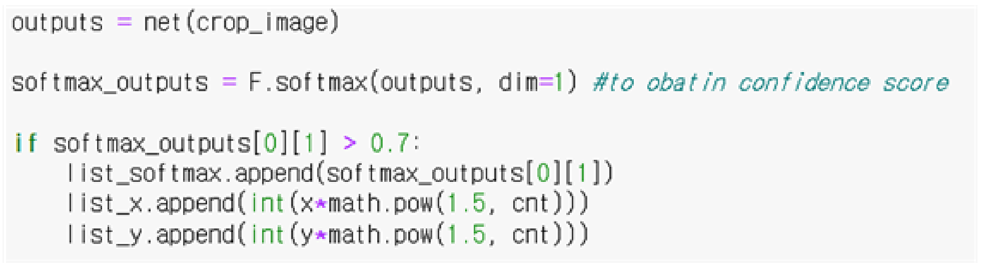
We implemented a sliding window with the simplest method which uses a nested loop. First, we selected a 736*736 size image and resized it to a 150*150 size using Photoshop to ensure that the face size fits 36*36. Then, we tried to determine whether or not a face exists in an image of 150*150 size by using a 36*36 size sliding window.



```python
#Sliding Window
for y in range(0, image.shape[0]-36, stepSize): #vertical
    for x in range(0, image.shape[1]-36, stepSize): #horizantal
        crop_image = transforms.functional.crop(image_tensor, y, x, 36, 36)
```

Figure 7: Sliding Window

First of all, the image was loaded through ‘cv2.imread’ function, and then, using nested for loop, the vertical and horizontal starting points, where the image to be tested was located, were sequentially received as input. To crop an image to a 36*36 window size, we used the existing ‘transforms.functional.crop’ function. As a result, we could get a cropped image by passing a tensor data type image, vertical starting point, horizontal starting point, height, and width to the corresponding function as parameters. At this time, when the interval between cropped images was initially set to 8 pixels, the number of windows detecting ‘face’ was only 5. Later, when ‘stepSize’ interval variable was adjusted to 4 pixels, it was confirmed that the number of windows detecting ‘face’ increased.

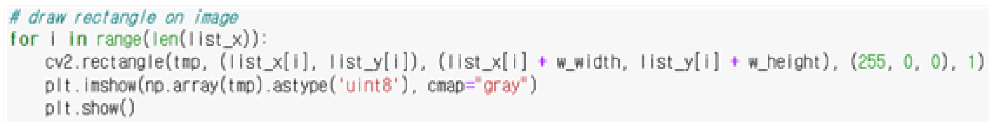


```
outputs = net(crop_image)

softmax_outputs = F.softmax(outputs, dim=1) #to obatin confidence score

if softmax_outputs[0][1] > 0.7:
    list_softmax.append(softmax_outputs[0][1])
    list_x.append(int(x*math.pow(1.5, cnt)))
    list_y.append(int(y*math.pow(1.5, cnt)))
```

Figure 8: Cropping images

Next, we put the cropped image into our classifier to detect faces and put the result in the ‘outputs’ variable. Then, the ‘softmax’ function was used to obtain the reliability probability of the test result. We are using the ‘cross-entropy loss’ in the network as the loss function, but using that function, a probability value could not be obtained as a result. So the ‘softmax’ function was used separately during the test. By using the ‘softmax’ function, it is possible to obtain the probability of a face and the probability of not being a face with a total sum of 1. After then, only values with a probability of being a face higher than 0.7 were selected, because when set to higher than 0.5, the detection performance was shown with low accuracy. So, only when it has a high probability value, the probability, vertical and horizontal starting points were stored separately using a list. The reason why the starting points are multiplied with 1.5 power in the list will be described in detail later in Scaling part(4.4).

4.2. Visualization



```
# draw rectangle on image
for i in range(len(list_x)):
    cv2.rectangle(tmp, (list_x[i], list_y[i]), (list_x[i] + w_width, list_y[i] + w_height), (255, 0, 0), 1)
    plt.imshow(np.array(tmp).astype('uint8'), cmap="gray")
    plt.show()
```

Figure 9: Visualization

From now on, we describe the visualization of face detection on the existing image. Visualization was displayed on the 150*150 size image using the previously saved list of starting points with a probability of more than 70% being a face. At first, we made rectangles using the 'cv2.rectangle' function while looping as many as the number of horizontal starting points. Then, we tried to display the results on the screen using the 'cv2.imshow' function, but in our jupyter environment, this function did not work properly, so we proceeded with visualization using 'matplotlib.pyplot' as 'plt'. After simple pre-processing with the 'plt.imshow' function, the result using the 'plt.show' function to display the image is as follows.

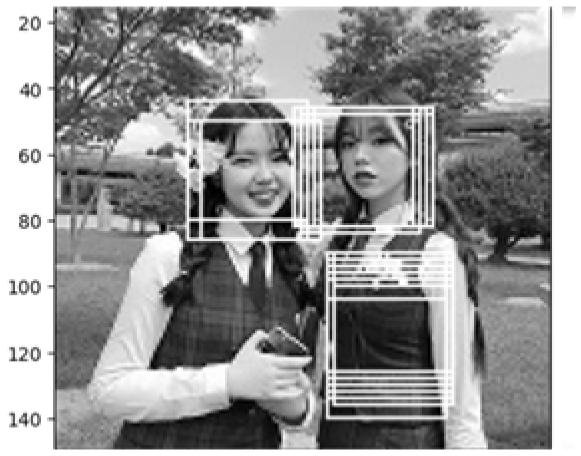
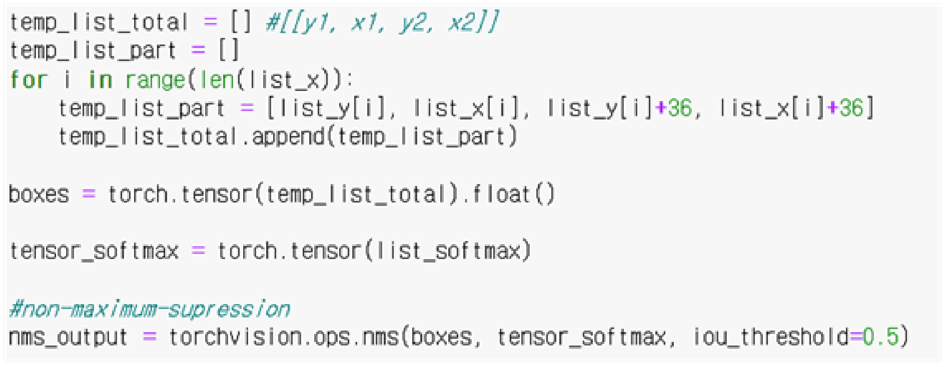


Figure 10: Result of visualization

4.3. Implementing Non-Max Suppression

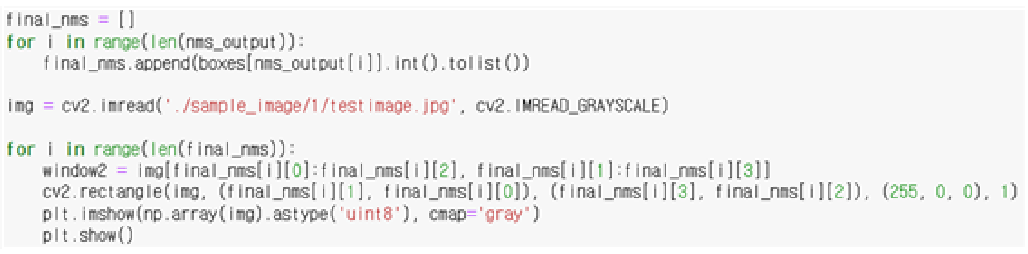
As a result of performing face detection using a sliding window, it was confirmed that faces were detected several times due to the characteristics of the window even though they were the same face. To eliminate this duplication for optimization, we decided to use a method called Non-Maximum-Suppression.



```
temp_list_total = [] #[[y1, x1, y2, x2]]
temp_list_part = []
for i in range(len(list_x)):
    temp_list_part = [list_y[i], list_x[i], list_y[i]+36, list_x[i]+36]
    temp_list_total.append(temp_list_part)

boxes = torch.tensor(temp_list_total).float()

tensor_softmax = torch.tensor(list_softmax)

#non-maximum-supression
nms_output = torchvision.ops.nms(boxes, tensor_softmax, iou_threshold=0.5)
```

Figure 11: Non-Max Suppression

In order to use the 'NMS' function, the location of an image with a tensor data type, the confidence score of the image, and the iou threshold value are required as parameters. To create 'boxes' variables as arguments, which include locations of the cropped images, a list is created separately and the data type is changed to tensor. For the remaining arguments, a softmax output value was changed to tensor, and the threshold value was initially set to 0.7. In consequence, duplicates were excessively removed, which led to the removal of correctly detected objects. To solve this problem, we lowered the threshold value to 0.5, we got the following result we were looking for.



```
final_nms = []
for i in range(len(nms_output)):
    final_nms.append(boxes[nms_output[i]].int().tolist())

img = cv2.imread('./sample_image/1/testimage.jpg', cv2.IMREAD_GRAYSCALE)

for i in range(len(final_nms)):
    window2 = img[final_nms[i][0]:final_nms[i][2], final_nms[i][1]:final_nms[i][3]]
    cv2.rectangle(img, (final_nms[i][1], final_nms[i][0]), (final_nms[i][3], final_nms[i][2]), (255, 0, 0), 1)
    plt.imshow(np.array(img).astype('uint8'), cmap='gray')
    plt.show()
```

Figure 12: Visualization of NMS

To check the appearance after applying ‘NMS’, the previous visualization technique was implemented in the same way. The reason for making a separate ‘final_nms’ list and converting the tensor data type to a list is to access the elements more easily using indexing. Resulting of the usage of the ‘plt.show’ function to display the image, is as follows.

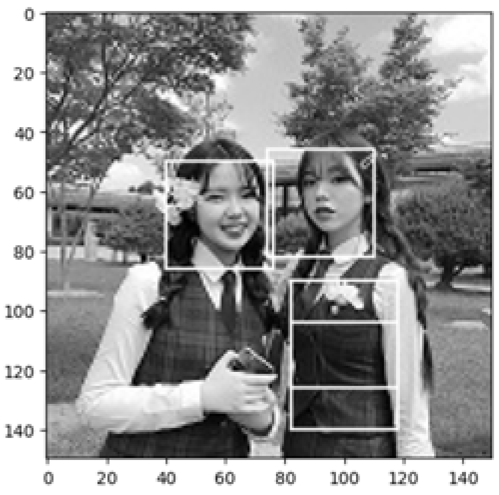
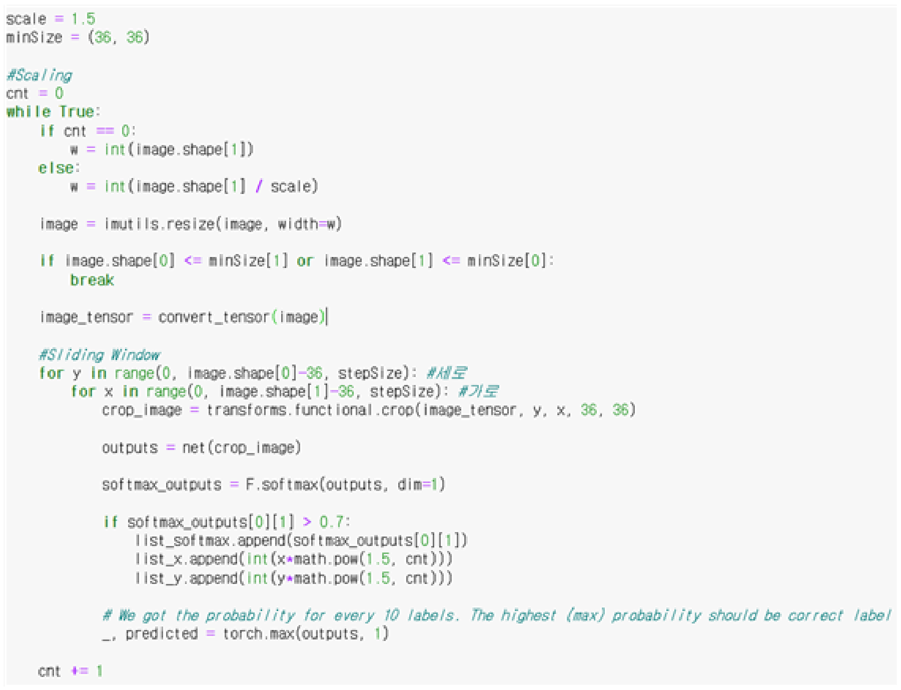


Figure 13: Result of NMS

4.4. Implementing Scaling

After building the classifier, we tested with a 36*36 image, tested with a 150*150 image with a sliding window, and finally implemented scaling. This was because we thought that scaling to create a pyramid of images is meaningful only when face detection is properly performed in the previous test.



```
scale = 1.5
minSize = (36, 36)

#Scaling
cnt = 0
while True:
    if cnt == 0:
        w = int(image.shape[1])
    else:
        w = int(image.shape[1] / scale)

    image = imutils.resize(image, width=w)

    if image.shape[0] <= minSize[1] or image.shape[1] <= minSize[0]:
        break

    image_tensor = convert_tensor(image)

    #Sliding Window
    for y in range(0, image.shape[0]-36, stepSize): #세로
        for x in range(0, image.shape[1]-36, stepSize): #가로
            crop_image = transforms.functional.crop(image_tensor, y, x, 36, 36)

            outputs = net(crop_image)

            softmax_outputs = F.softmax(outputs, dim=1)

            if softmax_outputs[0][1] > 0.7:
                list_softmax.append(softmax_outputs[0][1])
                list_x.append(int(x*math.pow(1.5, cnt)))
                list_y.append(int(y*math.pow(1.5, cnt)))

            # We got the probability for every 10 labels. The highest (max) probability should be correct label
            _, predicted = torch.max(outputs, 1)

    cnt += 1
```

Figure 14: Scaling

We implemented the code to reduce the size of the image by 1.5 scales and proceed with a sliding window for each resized image. In a while loop, the condition for the stopping scaling is when the scaled image becomes equal to or smaller than the minimum image size of 36*36. At this time, the reason why we use a ‘cnt’ variable is to count the number of Scaling. When a resized image enters the sliding window, the image size is smaller than the original image, so the detected position cannot be matched with the actual image. That is, as much as scaling, we draw a rectangle in each starting point by multiplying the power of scale and ‘cnt’ using a ‘pow’ function. Also, we considered when the ‘cnt’ value is 0 because in this case, we do not need to change our image size.

At first, we did not consider these parts, so when the result was shown, we thought it failed in face detection. But we could solve this problem by changing the location of the starting point appropriately. In other words, we could see the detections bring back to the original scale and be merged well.

5. Conclusion

We made a face recognition system based on the Convolutional Neural Networks. This system is trained with a training dataset and predicts whether the given image is face or not. To make this system, we went through several steps.

First, we imported necessary libraries and preprocessed the dataset. Second, we created a face/non-face classifier based on the Convolutional Neural Network using PyTorch. With this network, we tried to increase the accuracy of detection results. Third, we implemented a detector. To recognize the bigger size image, we made a sliding window, which can divide the whole size image into the small size. Then, using the Non-Max Suppression method, the redundant detection boxes were removed, and we visualized the final result boxes by using the 'matplotlib' library. Also, to increase the probabilities of finding faces of the image, the image is resized for a few steps. The images which are resized and cropped using the sliding window are put into the network.

During the process of making this system, we took some trial and error. After making the classifier, to improve the accuracy of it, we tried various methods such as adjusting learning rate and optimizer. And then, when implementing a detector, the number of detections is adjusted properly by revising the interval between cropped images. To remove duplicate detected boxes, we used the NMS existing library, and we tried to adjust iou threshold values.

Through this step, we could learn the architecture of the neural network and how CNN is trained from data. And we could know how each step is influenced by a plenty of variables, for example, the step of increasing accuracy is influenced by the type of loss function, optimizer, learning rate and so on . However, we had a limitation. It is that we could not improve our detector performance. In the result of detecting faces in the image, there are some incorrect boxes drawing on non-face objects. If we improve our network and detector performance, the boxes will be more accurate.