

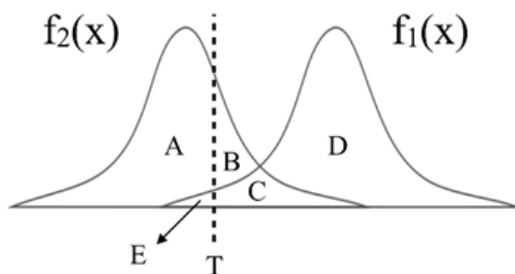
Up Computer Vision: from Recognition to Geometry

HW2

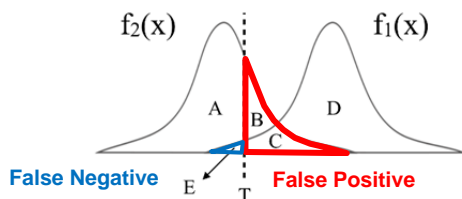
Name: 楊仲萱 Department: 電子所 ICS 組 博一 Student ID: F07943023

Problem 1

- (a) Assume X is a continuous random variable that denotes the estimated probability of a binary classifier. The instance is classified as positive if $X > T$ and negative otherwise. When the instance is positive, X follows a PDF $f_1(x)$. When the instance is negative, X follows a PDF $f_2(x)$. Please specify which regions (A ~ E) represent the cases of *False Positive* and *False Negative*, respectively. Clearly explain why. (6%)



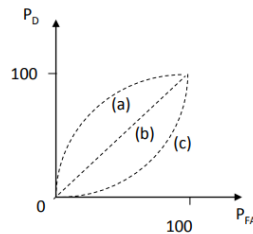
Answer:



The region B and region C represent the case of *False Positive* since they are under the PDF $f_2(x)$ which are negative in reality but are classified as positive. Similarly, the region E follows the PDF $f_1(x)$ meaning it's positive in reality but is classified as negative, therefore the region E represent the case of *False Negative*.

It makes sense that when we lower the threshold T from the middle, there are more instances being classified as positive. As the result, the area of *False Positive* will increase and the area of *False Negative* will decrease.

- (b) There are three ROC curves in the plot below. Please specify which ROC curves are considered to have reasonable discriminating ability, and which are not. Also, please answer that under what circumstances will the ROC curve fall on curve (b)? (6%)



Answer:

The ROC curve (a) are considered to have reasonable discriminating ability, since when we move the threshold value away from negative infinity, the value of P_{FA} drops faster than P_D which is reasonable. On the other hand, ROC curves (b) and (c) are not considered to have reasonable discriminating ability since curve (b) is just a random guess classifier and curve (c) is a classifier that is worse than random classifier.

When a predictor makes its decision by random guesses, the ROC curve will fall on curve (b) which is the line of no-discrimination. Because of the randomness, the fraction of cases in true positive is the same as it in false positive, meaning that the true positive rate and the false positive rate are the same (diagonal line).

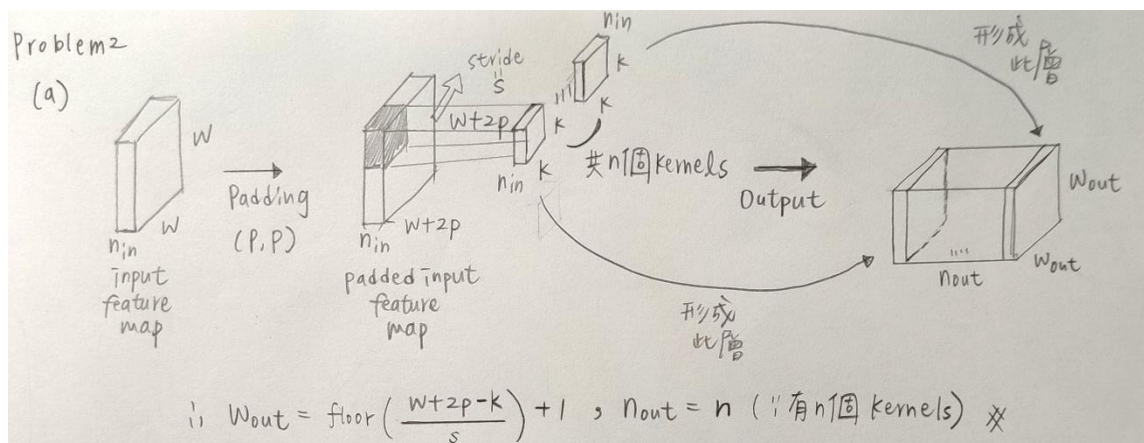
Problem 2

- (a) Given a convolutional layer which contains:

n kernels with size $k * k * n_{in}$
padding size (p, p)
stride size (s, s) .

With input feature size $W * W * n_{in}$, calculate the size of output feature $W_{out} * W_{out} * n_{out}$.

- (i) $W_{out} = ?$ (2%)
(ii) $n_{out} = ?$ (2%)



- (b) $n_{in} = 3, k = 5, s = 2, p = 1, n = 256, W = 64$, calculate the number of parameters in the convolutional layer (4%)

承上是頁之圖，中間的 kernels 代表著 total number of parameters

若有考慮 bias term:
$$\underbrace{[(k \times k \times n_{in}) + 1]}_{\text{filter 大小} \times \text{Bias}} \times \underbrace{n}_{\text{filter 個數}} = ((5 \times 5 \times 3) + 1) \times 256 = 76 \times 256 = 19456$$

若沒有 bias term:
$$(k \times k \times n_{in}) \times n = (5 \times 5 \times 3) \times 256 = 19200$$
 ✖

Problem 3 (report 35% + code 5%)

(a) PCA (15%)

In this task, you need to implement PCA from scratch, which means you cannot call PCA function directly from existing packages.

1. Perform PCA on the training data. Plot the mean face and the first five eigenfaces and show them in the report. (5%)

The mean face of all 280 training data is shown below.



The first five eigenfaces are shown in the following table.

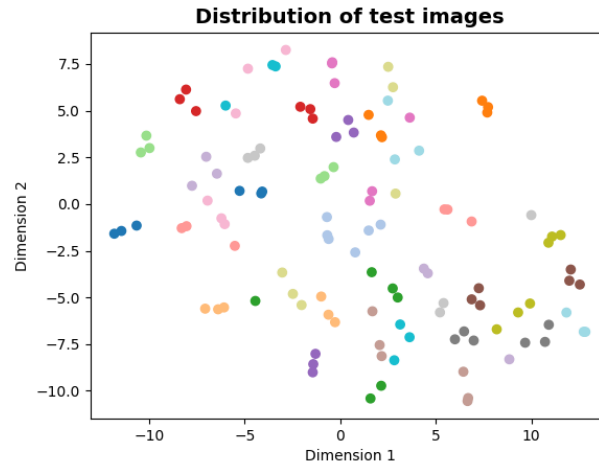
	Eigenface 1	Eigenface 2	Eigenface 3	Eigenface 4	Eigenface 5
Result Image					

2. Take **person₈_image₆**, and project it onto the above PCA eigenspace. Reconstruct this image using the first $n = \{ 5, 50, 150, \text{all} \}$ eigenfaces. For each n , compute the mean square error (MSE) between the reconstructed face image and the original **person₈_image₆**. Plot these reconstructed images with the corresponding MSE values in the report. (5%)

	Original_8_6	n = 5	n = 50	n = 150	n = all (280)
MSE		694 (693.702)	119 (119.200)	40 (40.397)	0 (9.5×10^{-25})
Result Image					











- Reduce the dimension of the image in testing set to dim = 100. Use t-SNE to visualize the distribution of test images. (5%)

After reducing the dimension of test images and visualizing with t-SNE, the distribution is shown below.



- (bonus 5%) Implement the Gram Matrix trick for PCA. Compare the two reconstruction images from standard PCA/Gram Matrix process. If the results are different, please explain the reason. Paste the main code fragment(screenshot) on the report with discussion.

The results from standard PCA and Gram Matrix are the same. The reconstructed output images are listed in the table.

	Original	n = 5	n = 50	n = 150	n = all (280)
Reconstruct images from standard PCA					
MSE	0	694 (693.702)	119 (119.200)	40 (40.397)	0 (9.5×10^{-25})
Result Image					
Reconstruct images from Gram Matrix					
MSE		694 (693.702)	119 (119.200)	40 (40.397)	0 (0.283)
					

The main code fragment for Gram Matrix is in the following picture.

```
# (280,280)
covariance_matrix_G = np.matmul(np.transpose(x_minus_mean),x_minus_mean)

print("Calculating eigen vector for Gram Matrix...")
eigen_value_G, eigen_vector_G = np.linalg.eig(covariance_matrix_G)
eigen_value_G = eigen_value_G.real
eigen_vector_G = eigen_vector_G.real    #(280,280)

index_array_G = np.argsort(eigen_value_G)    #(280,)
sorted_eigen_vector_G = np.zeros((num_of_training_data,num_of_training_data))    #(280,280)
for k in range(num_of_training_data):
    sorted_eigen_vector_G[:,k] = eigen_vector_G[:,index_array_G[num_of_training_data-1-k]]

# Recover the eigen vector
recovered_eigen_vector_G = np.matmul(x_minus_mean, sorted_eigen_vector_G)    #(2576,280)
recovered_eigen_vector_G = recovered_eigen_vector_G / np.linalg.norm(recovered_eigen_vector_G, axis = 0)
recovered_eigen_vector_G = np.transpose(recovered_eigen_vector_G)    #(280,2576)

# Project to Eigen vector
P_G = np.matmul(recovered_eigen_vector_G,(image_8_6-mean_face))    #(280,)
P_G = P_G.reshape((1,280))
```

The new covariance matrix is now only in the size of (280,280) which is much smaller than the original one (2576,2576). As the result, the process of finding eigen vectors and eigen values can be faster. In order to find the real eigen vector, it can be achieved with multiplying the input (X-mean) matrix and then normalizing the result. After these steps, the true eigen vector can be recovered, therefore the results from two methods are the same.

(b) k-NN (10%)

To apply the k-nearest neighbors (k-NN) classifier to recognize the testing set images, please determine the best k and n values by 3-fold cross-validation.

For simplicity, the choices for such hyper-parameters are:

$$k = \{1, 3, 5\} \text{ and } n = \{3, 10, 39\}.$$

Please show the cross-validation results and explain your choice for (k, n). Also, show the recognition rate on the testing set using your hyper-parameter choice.

The accuracy after using 3-fold cross-validation			
	k = 1	k = 3	k = 5
n = 3	0.68333 (0.7 + 0.6625 + 0.6875) / 3	0.59306 (0.5417 + 0.625 + 0.6125) / 3	0.51528 (0.4333 + 0.5125 + 0.6) / 3
n = 10	0.85139 (0.8667 + 0.8625 + 0.825) / 3	0.74583 (0.725 + 0.7875 + 0.725) / 3	0.67778 (0.6833 + 0.6875 + 0.6625) / 3
n = 39	0.92361 (0.9333 + 0.9375 + 0.9) / 3	0.84861 (0.8083 + 0.8875 + 0.85) / 3	0.77639 (0.7417 + 0.8 + 0.7875) / 3

With the help of cross_val_score function from sklearn.model_selection, I take the average of accuracy from 3-fold cross-validation as its final accuracy. Since the accuracy when using k as 1 and n as 39 is the highest, I chose (k, n) = (1, 39) as the hyper-parameter.

The recognition rate on the testing set (120 images) using (k, n) = (1, 39) is **0.95833**

(c) K-means (10%)

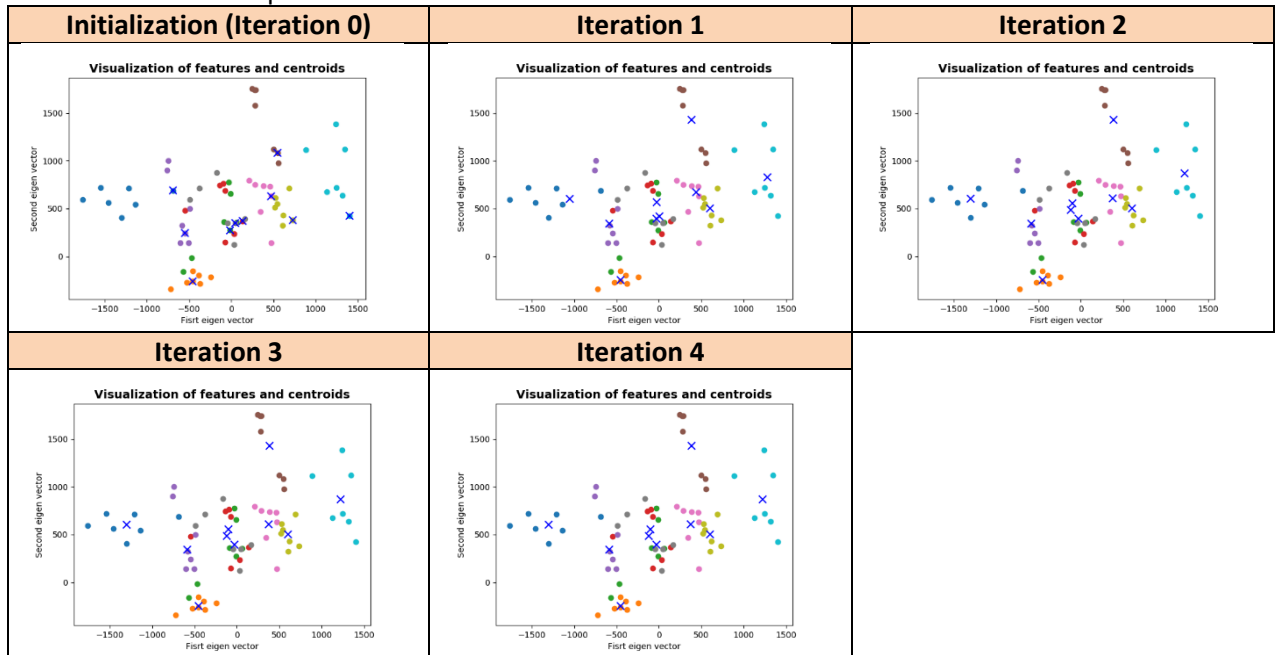
Reduce the dimension of the images in the first 10 class of training set to $\text{dim} = 10$ using PCA.

Implement the k-means clustering method to classify these images.

Please use weighted Euclidean distance to implement the k-means clustering. The weight of the best 10 eigenfaces is $[0.6, 0.4, 0.2, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]$

- Please visualize the features and the centroids to 2D space with the first two eigenfaces for each iteration in k-means clustering (up to 5 images). (5%)

The blue "X" represents the centroid of each cluster.



- Compare the results of K-means and ground truth. (5%)

The results of K-means clustering is down below.

[0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 4 4 3 3 3 2 3 3 2 3 3 4 4 4 4 4 4 5 5 5 5 5 5 6 6 6 6 6 6 6 7 7 7 7 7 7 7 8 8 8 8 8 8 8 9 9 9 9 9 9 9]

The ground truth in green and red part is 2, and the ground truth in blue part is 3.

After input images being projected to $\text{dim} = 10$, they are more similar to resulted means.

Ground Truth	Results of K-means	The projected input image	
Projected mean_2	Projected mean_4	2_3	2_4
Projected mean_2	Projected mean_3	2_5	2_6
Projected mean_3	Projected mean_2	3_1	3_4

Problem 4 (report 30% + baseline 10%)

(a) MNIST Classification (20%)

1. Build a CNN model and a Fully-Connected Network and train them on the MNIST dataset. Show the architecture of your models and the correspond parameters amount in the report. (5%)

For CNN model, the architecture is shown below.

The amount of parameters is 44,426.

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 6, 24, 24]	156
ReLU-2	[-1, 6, 24, 24]	0
MaxPool2d-3	[-1, 6, 12, 12]	0
Conv2d-4	[-1, 16, 8, 8]	2,416
ReLU-5	[-1, 16, 8, 8]	0
MaxPool2d-6	[-1, 16, 4, 4]	0
Conv2d-7	[-1, 120, 1, 1]	30,840
ReLU-8	[-1, 120, 1, 1]	0
Linear-9	[-1, 84]	10,164
ReLU-10	[-1, 84]	0
Linear-11	[-1, 10]	850
LogSoftmax-12	[-1, 10]	0
Total params: 44,426		
Trainable params: 44,426		
Non-trainable params: 0		

For Fully-Connected Network model, the architecture is shown below.

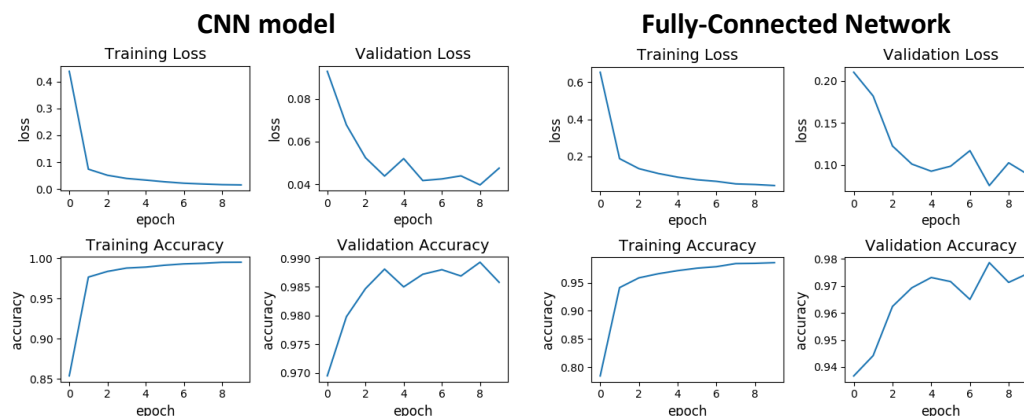
The amount of parameters is 563,662.

Layer (type)	Output Shape	Param #
Linear-1	[-1, 576]	452,160
ReLU-2	[-1, 576]	0
Linear-3	[-1, 144]	83,088
ReLU-4	[-1, 144]	0
Linear-5	[-1, 120]	17,400
ReLU-6	[-1, 120]	0
Linear-7	[-1, 84]	10,164
ReLU-8	[-1, 84]	0
Linear-9	[-1, 10]	850
LogSoftmax-10	[-1, 10]	0
Total params: 563,662		
Trainable params: 563,662		
Non-trainable params: 0		

2. Report your training / validation accuracy, and plot the learning curve (loss, accuracy) of the training process. (figure 5%+ baseline 5%)

(2-1) CNN model: Training accuracy: **99.5%** ; Validation accuracy: **98.6%**.

(2-2) Fully-Connected Network: Training accuracy: **98.6%** ; Validation accuracy: **97.5%**.

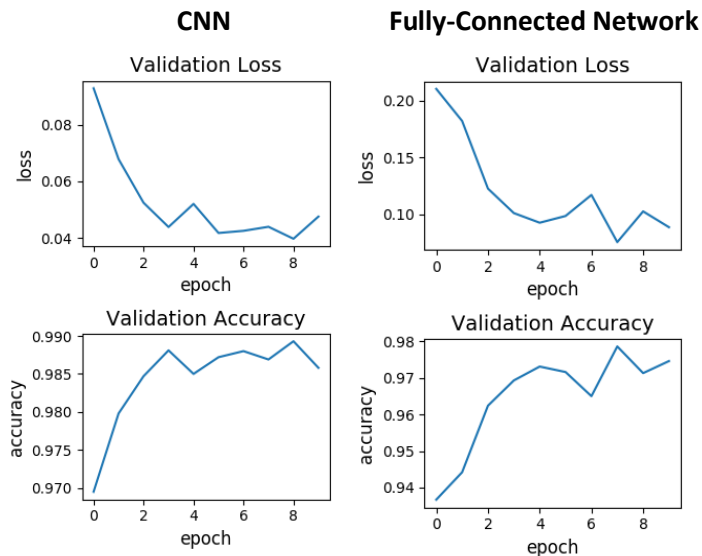


- Compare the results of both models and explain the difference. (5%)

CNN Model			Fully-Connected Network		
Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 6, 24, 24]	156	Linear-1	[-1, 576]	452,160
ReLU-2	[-1, 6, 24, 24]	0	ReLU-2	[-1, 576]	0
MaxPool2d-3	[-1, 6, 12, 12]	0	Linear-3	[-1, 144]	83,088
Conv2d-4	[-1, 16, 8, 8]	2,416	ReLU-4	[-1, 144]	0
ReLU-5	[-1, 16, 8, 8]	0	Linear-5	[-1, 120]	17,400
MaxPool2d-6	[-1, 16, 4, 4]	0	ReLU-6	[-1, 120]	0
Conv2d-7	[-1, 120, 1, 1]	30,840	Linear-7	[-1, 84]	10,164
ReLU-8	[-1, 120, 1, 1]	0	ReLU-8	[-1, 84]	0
Linear-9	[-1, 84]	10,164	Linear-9	[-1, 10]	850
ReLU-10	[-1, 84]	0	LogSoftmax-10	[-1, 10]	0
Linear-11	[-1, 10]	850			
LogSoftmax-12	[-1, 10]	0			
Total params: 44,426			Total params: 563,662		
Trainable params: 44,426			Trainable params: 563,662		
Non-trainable params: 0			Non-trainable params: 0		

The number of layers in CNN model is more than which in Fully-Connected network, but the total amount of parameters in CNN model is less than in Fully-Connected Network. Therefore, it shows that using CNN model is more suitable and more efficient for MNIST classification.

Furthermore, the final validation accuracy when using CNN model is higher than using Fully-Connected network which can be seen in the following figure.



(b) Face Recognition (20%)

- Extract image feature using pytorch pretrained alexnet and train a KNN classifier to perform human face recognition on the given dataset. Report the validation accuracy. (5%)

The validation accuracy is
43.578%

```
==>>> total training batch number: 219
==>>> total testing batch number: 48
Recognition rate: 0.43577981651376146
```


- Build your own model and train it on the given dataset to surpass the accuracy of previous stage. Show and explain the architecture of your model and report the validation accuracy. Paste the main code fragment(screenshot). (5%)

The architecture of my model is shown in the right figure. Since the task is face recognition, I decided to use VGG architecture as my reference to build my model and add some dropout layers to make its validation accuracy higher.

The final validation accuracy is shown in the below figure, which is 87.8%

```
Epoch: 98
Training batch index: 110, train loss: 0.051986, acc: 0.986
Validation batch index: 24, val loss: 0.806815, acc: 0.878
```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 94, 94]	1,792
LeakyReLU-2	[-1, 64, 94, 94]	0
Conv2d-3	[-1, 128, 92, 92]	73,856
LeakyReLU-4	[-1, 128, 92, 92]	0
MaxPool2d-5	[-1, 128, 46, 46]	0
Dropout-6	[-1, 128, 46, 46]	0
Conv2d-7	[-1, 256, 44, 44]	295,168
LeakyReLU-8	[-1, 256, 44, 44]	0
Conv2d-9	[-1, 256, 42, 42]	590,080
LeakyReLU-10	[-1, 256, 42, 42]	0
MaxPool2d-11	[-1, 256, 21, 21]	0
Dropout-12	[-1, 256, 21, 21]	0
Conv2d-13	[-1, 512, 19, 19]	1,180,160
LeakyReLU-14	[-1, 512, 19, 19]	0
Conv2d-15	[-1, 512, 17, 17]	2,359,808
LeakyReLU-16	[-1, 512, 17, 17]	0
MaxPool2d-17	[-1, 512, 8, 8]	0
Dropout-18	[-1, 512, 8, 8]	0
Conv2d-19	[-1, 512, 6, 6]	2,359,808
LeakyReLU-20	[-1, 512, 6, 6]	0
Conv2d-21	[-1, 512, 4, 4]	2,359,808
LeakyReLU-22	[-1, 512, 4, 4]	0
MaxPool2d-23	[-1, 512, 2, 2]	0
Dropout-24	[-1, 512, 2, 2]	0
Linear-25	[-1, 1024]	2,098,176
LeakyReLU-26	[-1, 1024]	0
Dropout-27	[-1, 1024]	0
Linear-28	[-1, 500]	512,500
LeakyReLU-29	[-1, 500]	0
Dropout-30	[-1, 500]	0
Linear-31	[-1, 100]	50,100
LogSoftmax-32	[-1, 100]	0
Total params: 11,881,256		
Trainable params: 11,881,256		
Non-trainable params: 0		

The main code fragment (screenshot) is in the following figures including the training part and the model building part.

Training Part

```
ep = 100
for epoch in range(ep):
    print('Epoch:', epoch)
    #####
    ## Training ##
    #####

    # Record the information of correct prediction and loss
    correct_cnt, total_loss, total_cnt = 0, 0, 0

    # Load batch data from dataloader
    for batch, (x, label) in enumerate(train_loader, 1):
        # Set the gradients to zero (left by previous iteration)
        optimizer.zero_grad()
        # Put input tensor to GPU if it's available
        if use_cuda:
            x, label = x.cuda(), label.cuda()
        # Forward input tensor through your model
        out = model(x)
        # Calculate loss
        loss = criterion(out, label)
        # Compute gradient of each model parameters base on calculated loss
        loss.backward()
        # Update model parameters using optimizer and gradients
        optimizer.step()

        # Calculate the training loss and accuracy of each iteration
        total_loss += loss.item()
        _, pred_label = torch.max(out, 1)
        correct_cnt += (pred_label == label).sum().item()

    # Calculate the training loss and accuracy of each iteration
    total_loss += loss.item()
    _, pred_label = torch.max(out, 1)
    total_cnt += x.size(0)
    correct_cnt += (pred_label == label).sum().item()

    # Show the training information
    if batch % 500 == 0 or batch == len(train_loader):
        acc = correct_cnt / total_cnt
        ave_loss = total_loss / batch
        print('Training batch index: {}, train loss: {:.6f}, acc: {:.3f}'.format(
            batch, ave_loss, acc))

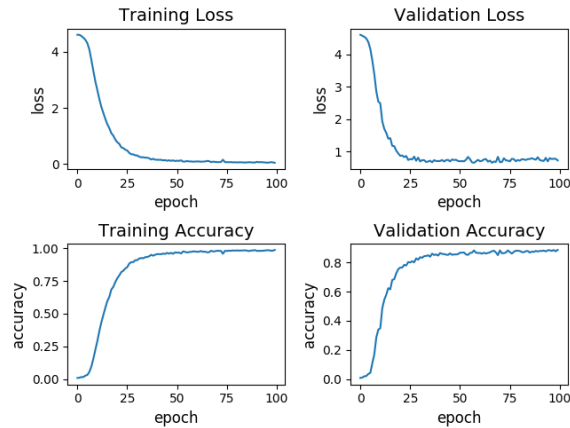
    train_loss.append(ave_loss)
    train_acc.append(acc)
```

Model Building Part

```
def __init__(self):
    super(VGG, self).__init__()
    # TODO
    self.convnet = nn.Sequential(OrderedDict([
        ('c1', nn.Conv2d(3, 64, kernel_size=(3, 3))),
        ('relu1', nn.LeakyReLU()),
        ('c2', nn.Conv2d(64, 128, kernel_size=(3, 3))),
        ('relu2', nn.LeakyReLU()),
        ('s3', nn.MaxPool2d(kernel_size=(2, 2), stride=2)),
        ('d4', nn.Dropout(p=0.2)),
        ('c5', nn.Conv2d(128, 256, kernel_size=(3, 3))),
        ('relu5', nn.LeakyReLU()),
        ('c6', nn.Conv2d(256, 256, kernel_size=(3, 3))),
        ('relu6', nn.LeakyReLU()),
        ('s7', nn.MaxPool2d(kernel_size=(2, 2), stride=2)),
        ('d8', nn.Dropout(p=0.2)),
        ('c9', nn.Conv2d(256, 512, kernel_size=(3, 3))),
        ('relu9', nn.LeakyReLU()),
        ('c10', nn.Conv2d(512, 512, kernel_size=(3, 3))),
        ('relu10', nn.LeakyReLU()),
        ('s9', nn.MaxPool2d(kernel_size=(2, 2), stride=2)),
        ('d9', nn.Dropout(p=0.2)),
        ('c13', nn.Conv2d(512, 512, kernel_size=(3, 3))),
        ('relu13', nn.LeakyReLU()),
        ('c14', nn.Conv2d(512, 512, kernel_size=(3, 3))),
        ('relu14', nn.LeakyReLU()),
        ('s15', nn.MaxPool2d(kernel_size=(2, 2), stride=2)),
        ('d15', nn.Dropout(p=0.2))
    ]))

    self.fc = nn.Sequential(OrderedDict([
        ('f16', nn.Linear(2048, 1024)),
        ('relu16', nn.LeakyReLU()),
        ('d16', nn.Dropout(p=0.5)),
        ('f17', nn.Linear(1024, 500)),
        ('relu17', nn.LeakyReLU()),
        ('d17', nn.Dropout(p=0.5)),
        ('f18', nn.Linear(500, 100)),
        ('sig18', nn.LogSoftmax(dim=-1))
    ]))
```

3. Plot the learning curve of your training process(training/validation loss/accuracy) in stage 2, explain the result you observed. (5%)

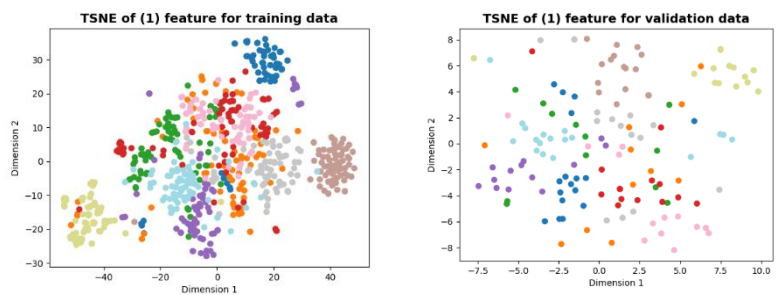


The validation accuracy is higher compared to the baseline accuracy. Moreover, it needs about 25 epochs to achieve the point that the following accuracy won't increase dramatically, but the baseline model only took about 4 epochs.

4. Pick the first 10 identities from dataset. Visualize the features of training/validation data you extract from pretrained alexnet and your own model to 2D space with t-distributed stochastic neighbor embedding (t-SNE), compare the results and explain the difference. (5%)

It's obvious that the features of training and validation data extracted from my model are more separate meaning that the accuracy for classification will be higher which is the same as the final result. The reason is that the VGG architecture is more suitable and the data processed by more convolutional layers will get high-level features.

Pretrained Alexnet



My own model

