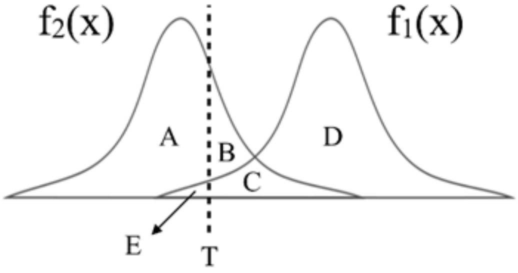
Computer Vision: from Recognition to Geometry
HW2

Name: 黃沛沅 Department: 機械系 Student ID: B04605067

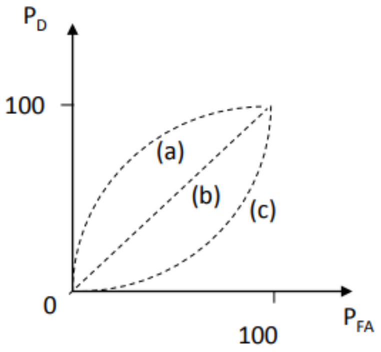
Problem 1

(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%)



False positive means that the testing result present positive (X>T), while it is negative in reality;
False negative means that the testing result present negative(X<T), while it is positive in reality.
B, C represent the case of false positive.
E represents the case of false negative.

(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%)



(a) curve is the reasonable one, and (c) is not. As the professor mentioned in class while we move the threshold T from negative infinity toward positive direction, the value of P_{FA} drops faster than the value of P_D. So, (a) is reasonable, and (c) is impossible.

The ROC will fall on curve (b) if the f_1 and f_2 have the same distribution. Therefore, the P_D and P_{FA} are always the same no matter where we put the threshold.

Problem 2

(a) Given a convolutional layer which contains:

n kernels with size $k * k * n_{in}$
padding size 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%)

$W_{out} = \frac{w-k+2p}{s} + 1$
$n_{out} = n$

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

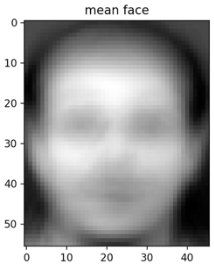
$w_{out} = \frac{64-5+2\times1}{2} + 1$
$total\ parameters = 31 \times 31 \times 256 = 246016$

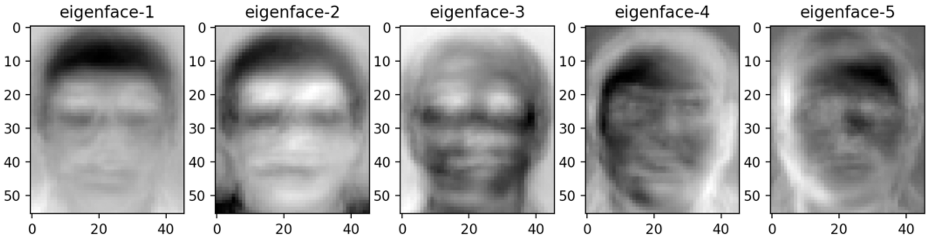
Problem 3 (report 35% + code 5%)

(a) PCA (15%)

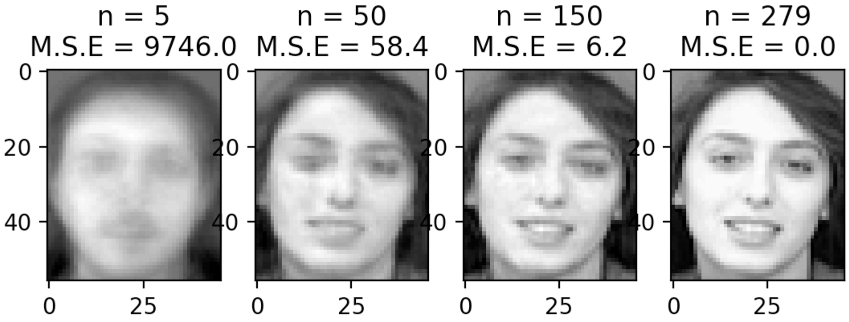
In this task, you need to implement PCA from scratch, which means you cannot call PCA function directly from existing packages.

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



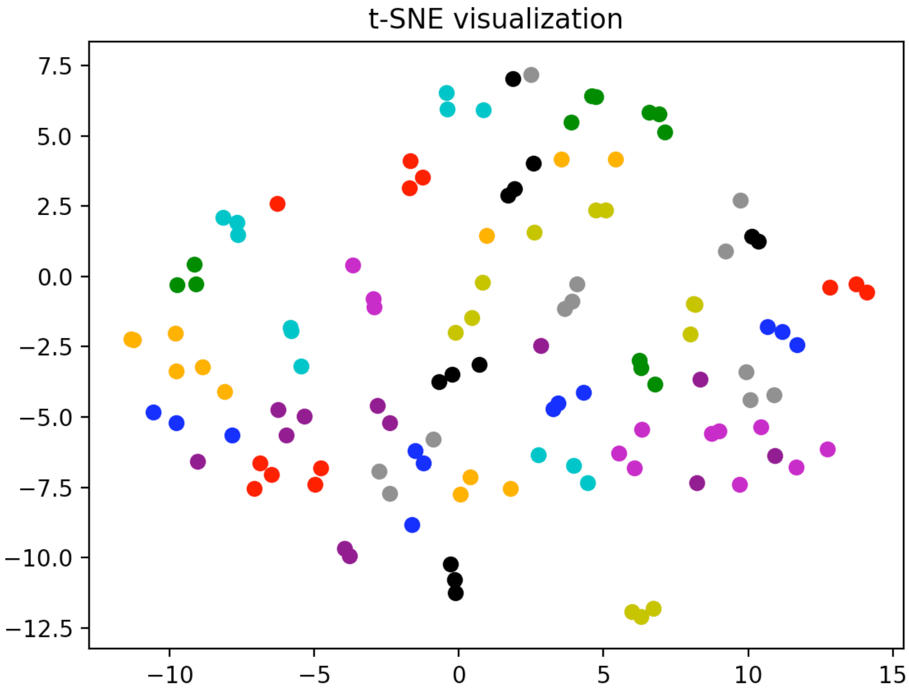


2. Take $\boldsymbol{person}_8\_\ \boldsymbol{image}_6$, and project it onto the above PCA eigenspace. Reconstruct this image using the first n = { 5, 50, 150, all } eigenfaces. For each n, compute the mean square error (MSE) between the reconstructed face image and the original $\boldsymbol{person}_8\_\ \boldsymbol{image}_6$. Plot these reconstructed images with the corresponding MSE values in the report. (5%)

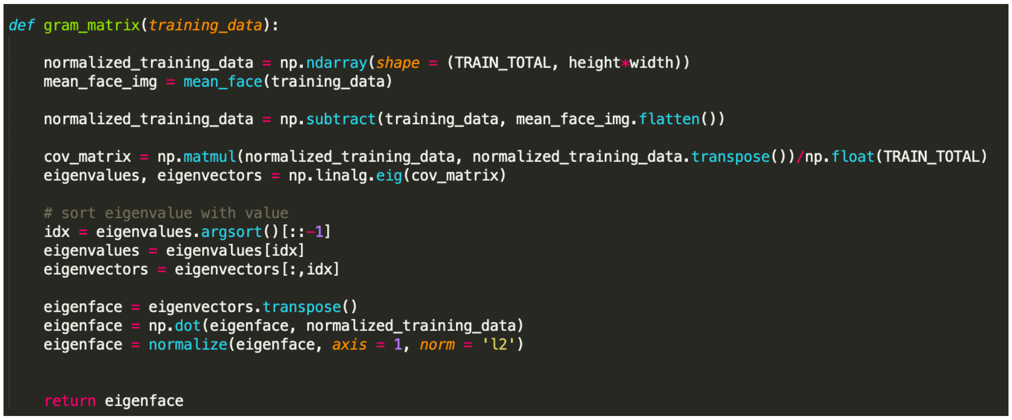


We can see that when we use all eigenfaces, the image M.S.E. is zero.

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



4. (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.



```
def gram_matrix(training_data):

    normalized_training_data = np.ndarray(shape = (TRAIN_TOTAL, height*width))
    mean_face_img = mean_face(training_data)

    normalized_training_data = np.subtract(training_data, mean_face_img.flatten())

    cov_matrix = np.matmul(normalized_training_data, normalized_training_data.transpose())/np.float(TRAIN_TOTAL)
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

    # sort eigenvalue with value
    idx = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:,idx]

    eigenface = eigenvectors.transpose()
    eigenface = np.dot(eigenface, normalized_training_data)
    eigenface = normalize(eigenface, axis = 1, norm = 'l2')


    return eigenface
```

Gram Matrix process takes less time to calculate, and the result is the same as PCA process. It’s because when counting eigenvectors the gram matrix has dimension of 280*280 while the PCA has 2576*2576. The result is shown when running the code.

(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} 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 table below shows k, n and their corresponding accuracy

k n	3	10	39
1	0.6833	0.8514	0.9236
3	0.5931	0.7458	0.8486
5	0.5153	0.6778	0.7764

The best k, n determined by cross-validation is 1, 39, which use the most eigenvectors and the least reference neighbor.
The accuracy on the testing with (k, n) = (1, 39) is 0.958 .

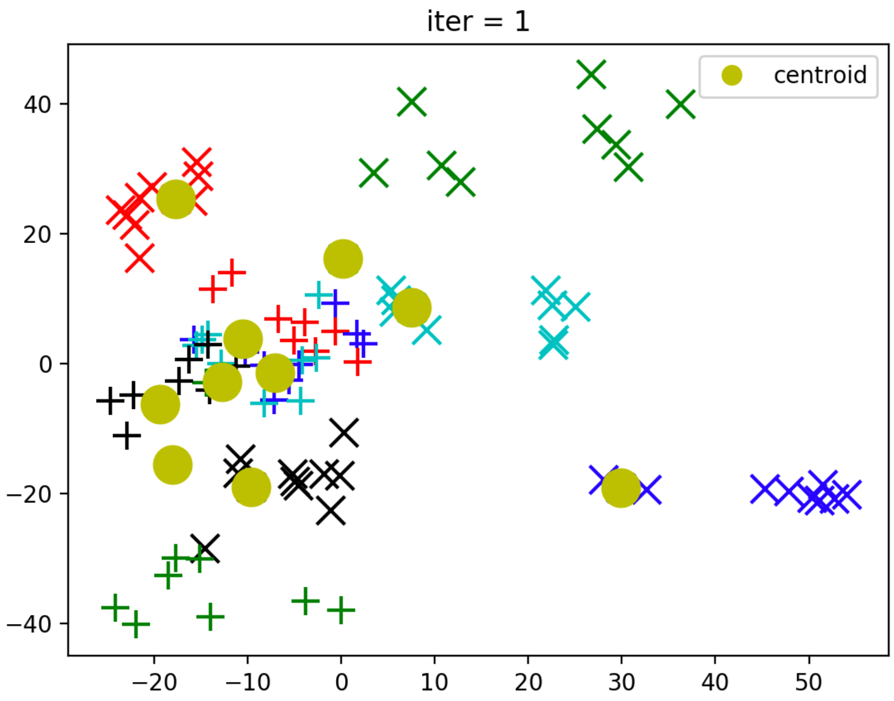
(c) K-means (10%)

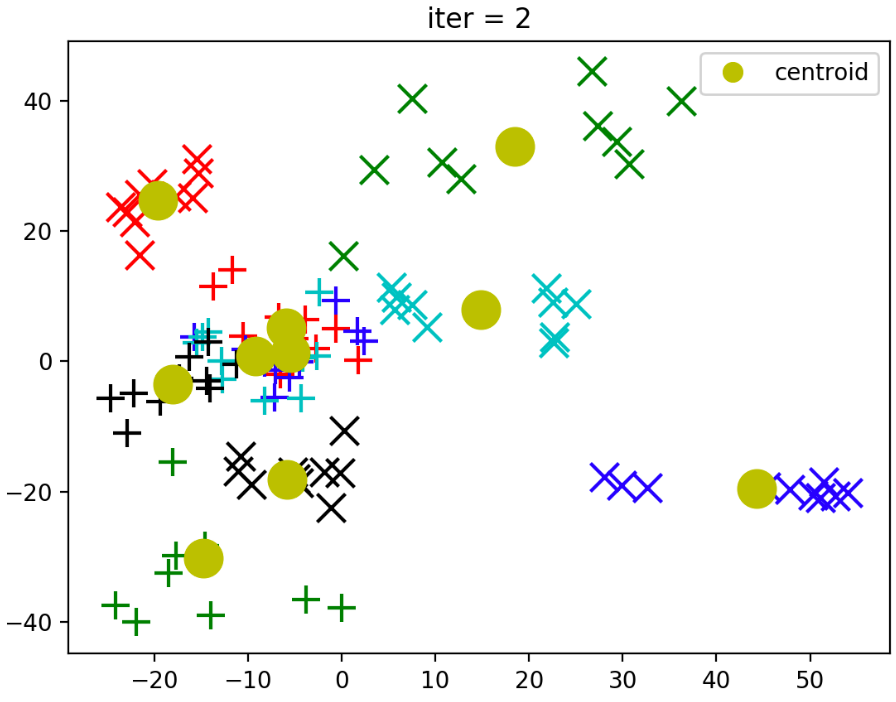
Reduce the dimension of the images in the first 10 class of training set to dim = 10 using PCA. Implement the k-means clustering method to classify the these images.

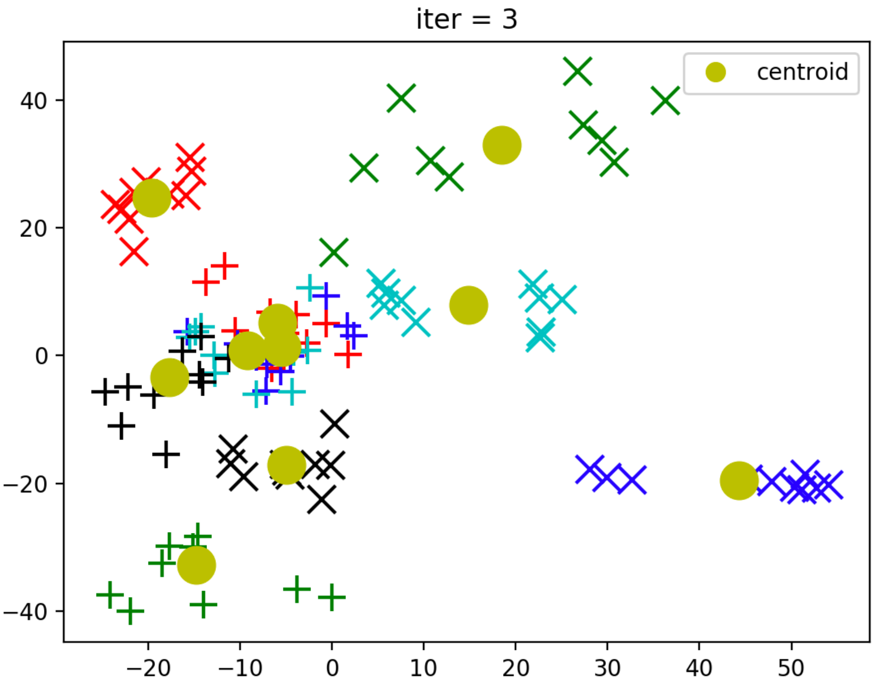
1. Please use weighted Euclidean distance to implement the k-means clustering. The weight of the best 10 eigenfaces is

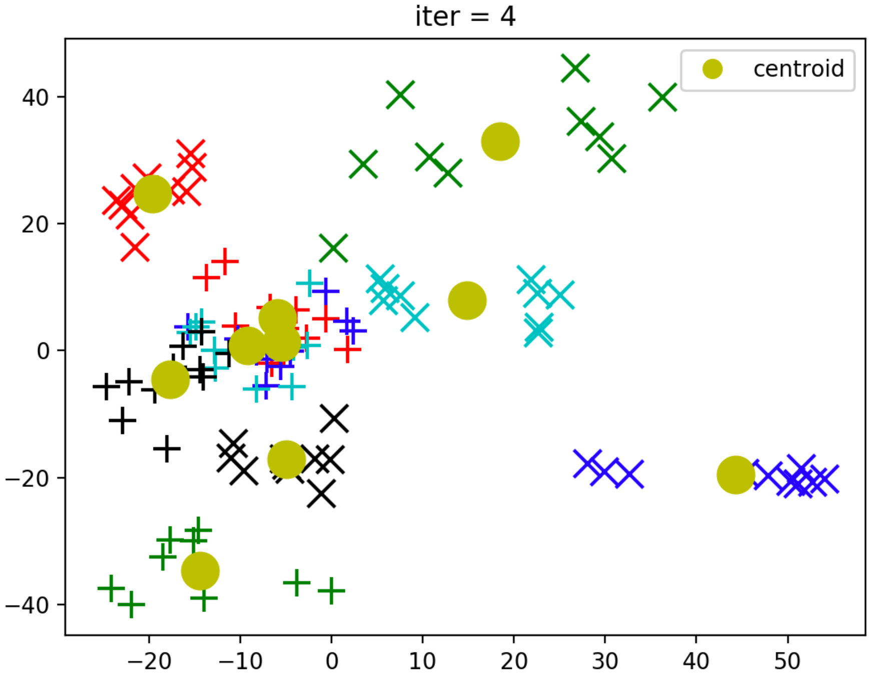
   [0.6, 0.4, 0.2, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]

2. 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%)


iter = 1
● centroid
40
20
0
−20
−40
−20
−10
0
10
20
30
40
50


iter = 2
● centroid
40
20
0
−20
−40
−20
−10
0
10
20
30
40
50





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

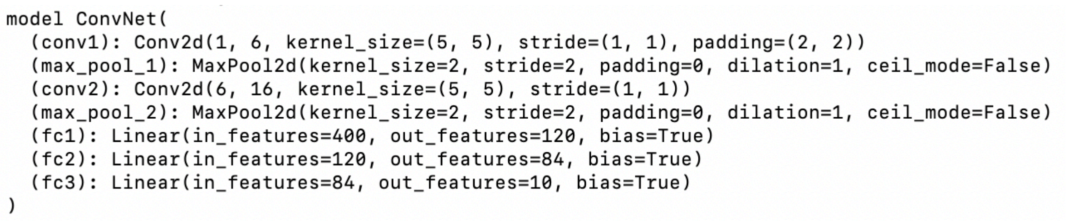
Accuracy = 0.985
Only the first data is incorrectly predicted as class_1, and others are truly predicted.

Problem 4 (report 30% + baseline 10%)

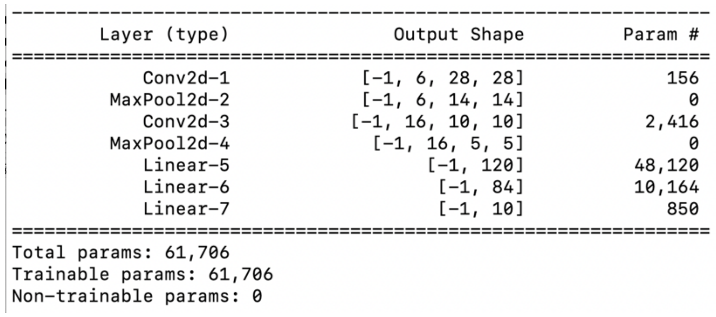
(a) MNIST Classification (20%)

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%)

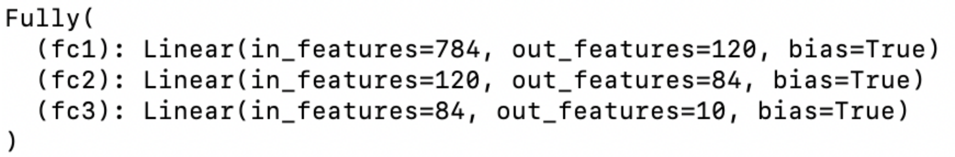
The convolution NN used totally 61706 parameters, while the fully connected network used 105214 parameters. We can see that CNN used much fewer parameters than fully connected network did.



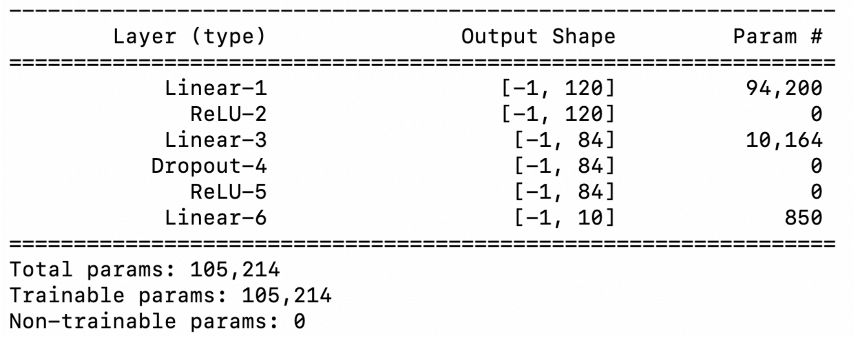
```
model ConvNet(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (max_pool_1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (max_pool_2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 28, 28]             156
         MaxPool2d-2            [-1, 6, 14, 14]               0
            Conv2d-3           [-1, 16, 10, 10]           2,416
         MaxPool2d-4             [-1, 16, 5, 5]               0
            Linear-5                  [-1, 120]          48,120
            Linear-6                   [-1, 84]          10,164
            Linear-7                   [-1, 10]             850
================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
```



```
Fully(
  (fc1): Linear(in_features=784, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```



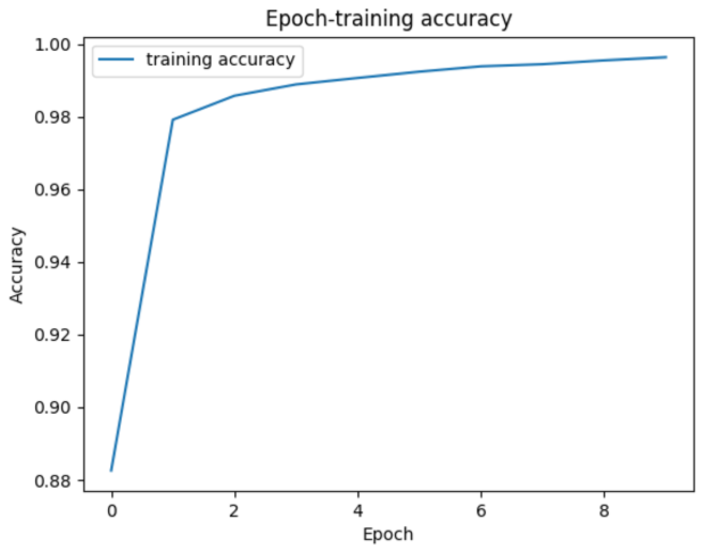
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                  [-1, 120]          94,200
              ReLU-2                  [-1, 120]               0
            Linear-3                   [-1, 84]          10,164
           Dropout-4                   [-1, 84]               0
              ReLU-5                   [-1, 84]               0
            Linear-6                   [-1, 10]             850
================================================================
Total params: 105,214
Trainable params: 105,214
Non-trainable params: 0
```

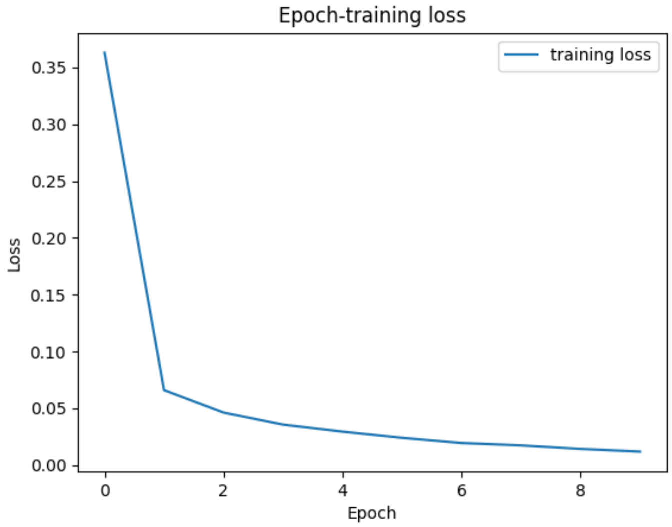
2. Report your training / validation accuracy, and plot the learning curve (loss, accuracy) of the training process. (figure 5%+ baseline 5%)

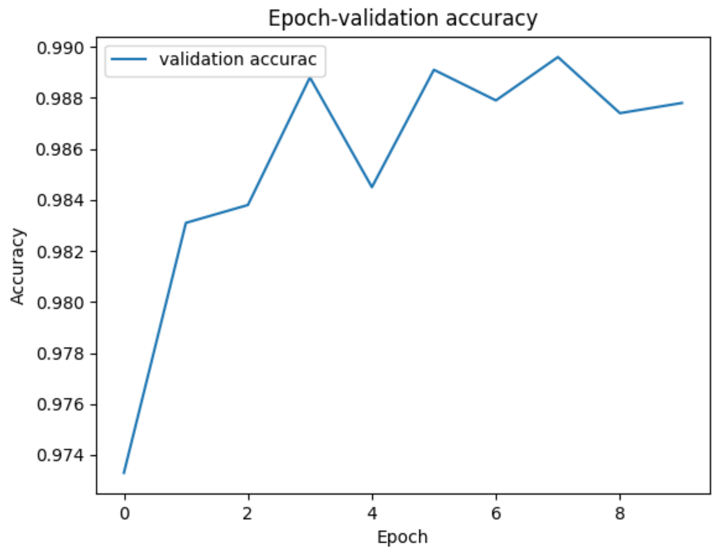
Lenet-5 model training process

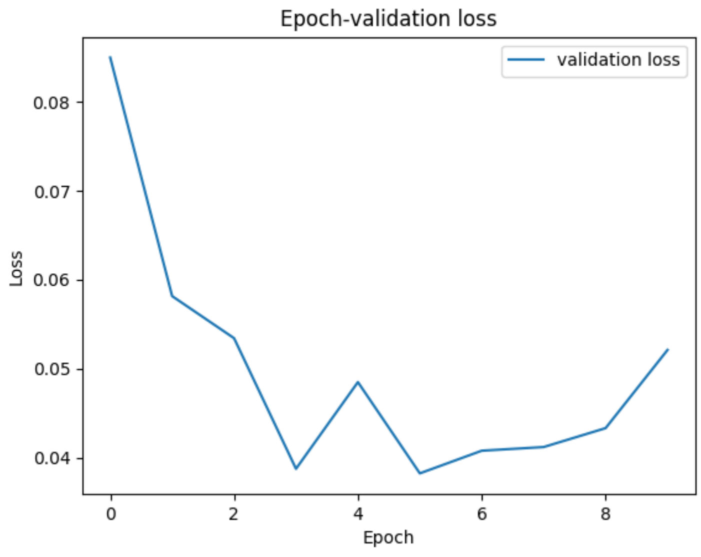
Training accuracy: 0.996

Validation accuracy: 0.99





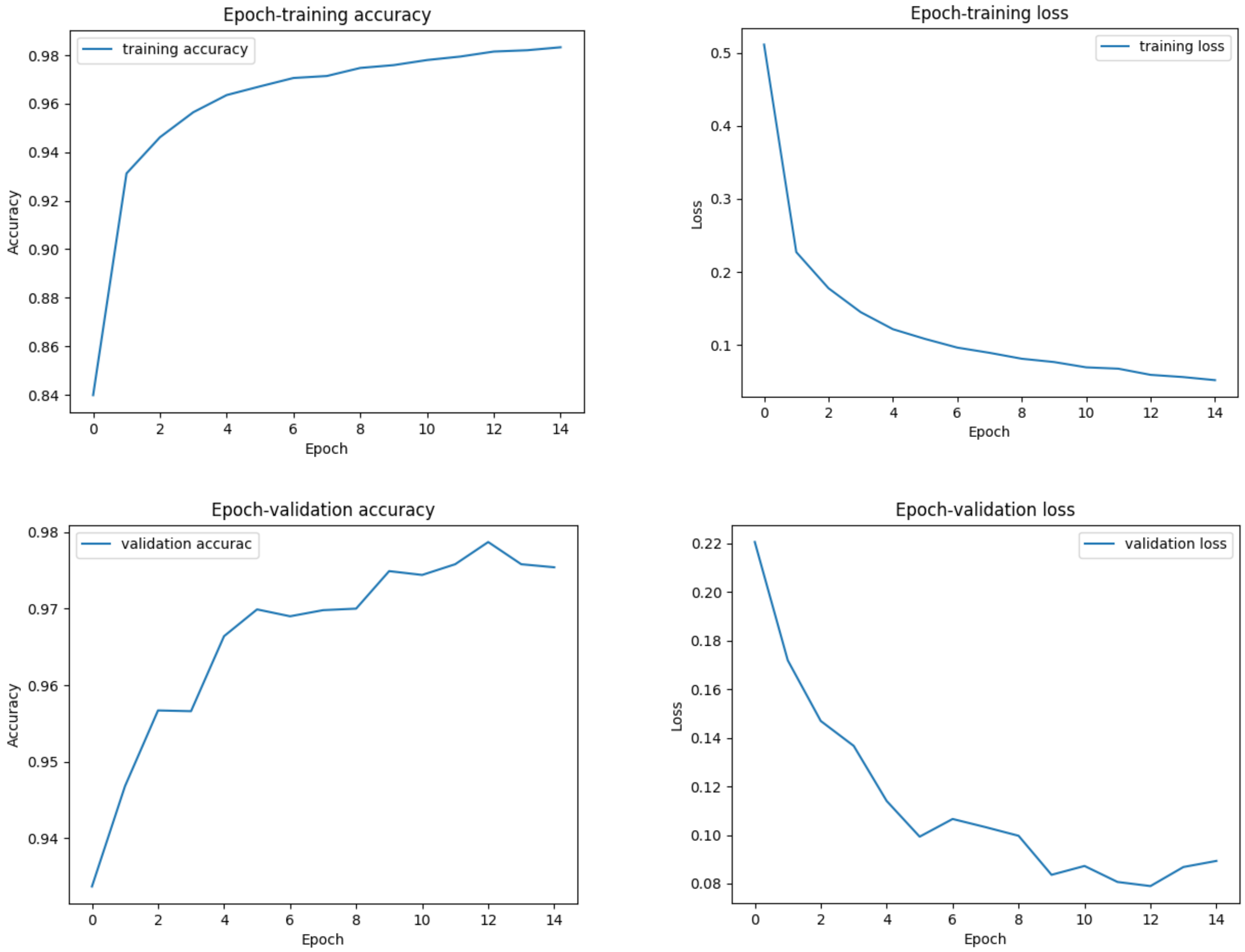




Fully connected network training process

Training accuracy: 0.988

Validation accuracy: 0.98



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

CNN used takes less computation cost but perform better than FCN. The reason probably is that CNN used more efficient way to learn. In convolutional layer in CNN, the network got image features first and used less dimension vector to present it. Then when it went to fully connected layer, it left fewer parameters but useful vectors as features. Therefore, its time costing is low, but performance is good.

(b) Face Recognition (20%)

1. 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%)

   Accuracy: 0.56
   Validation accuracy: 0.12
   Note: In the beginning, I achieve the accuracy of 35% easily, but the validation accuracy was pretty low (~6%), no matter the parameters I adjusted. Then I gave a look at the features extracted from alexnet. I noticed most them turned out to be zero because the usage of the mean in normalization is [0.485, 0.456, 0.406], which is closed to zero, and when they passed through ReLU activation function, the less-than zero features became zero, so some features vanished. To fix this issue, I changed the mean in normalization to be [3*0.485, 3*0.456, 3*0.406], escalating the validation accuracy (~0.12) as a result.

2. 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%)

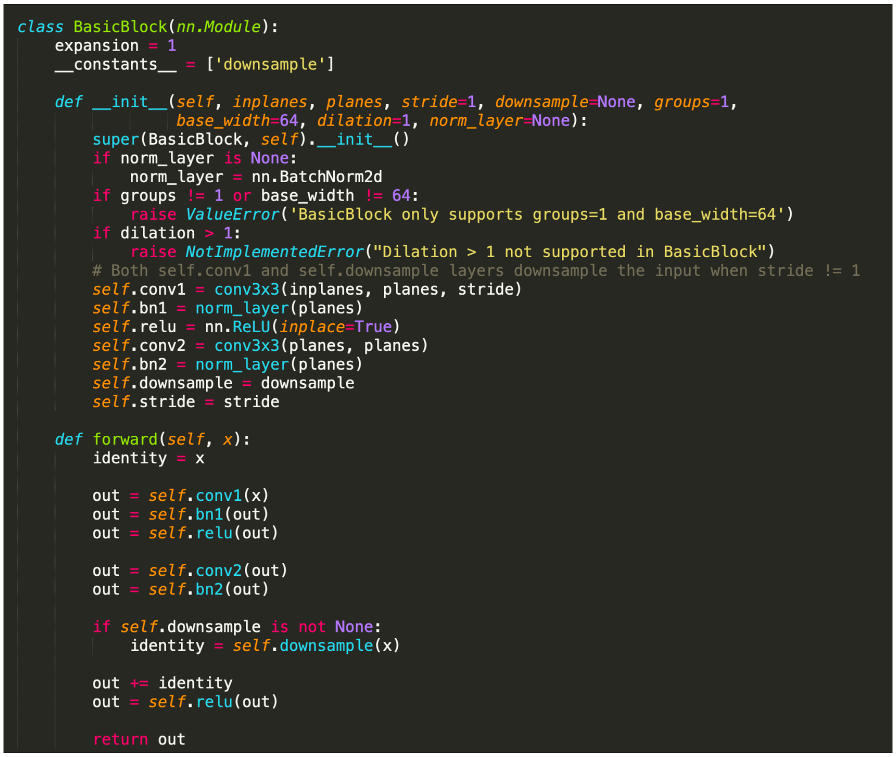
   Validation accuracy: 0.915
   I use resnet18 using the code similar to some website resources. Resnet18 includes convolution layer, batch normalization, maxpool, basic blocks, average pool and fully connected layers.

   The image initially goes into convolution layer, and then be normalized by batch normalization layer. The batch normalization normalizes the features without changing its dimension and it improve the efficiency. Before going Resnet layer, it is maxpooled.

Every layer of Resnet is composed of two blocks.
In Resnet18, each layer has two basic blocks, containing conv, bn1, relu, conv, bn2, downsampling.
The down sampling of the volume through the network is achieved by increasing the stride, making it fit the size we want.

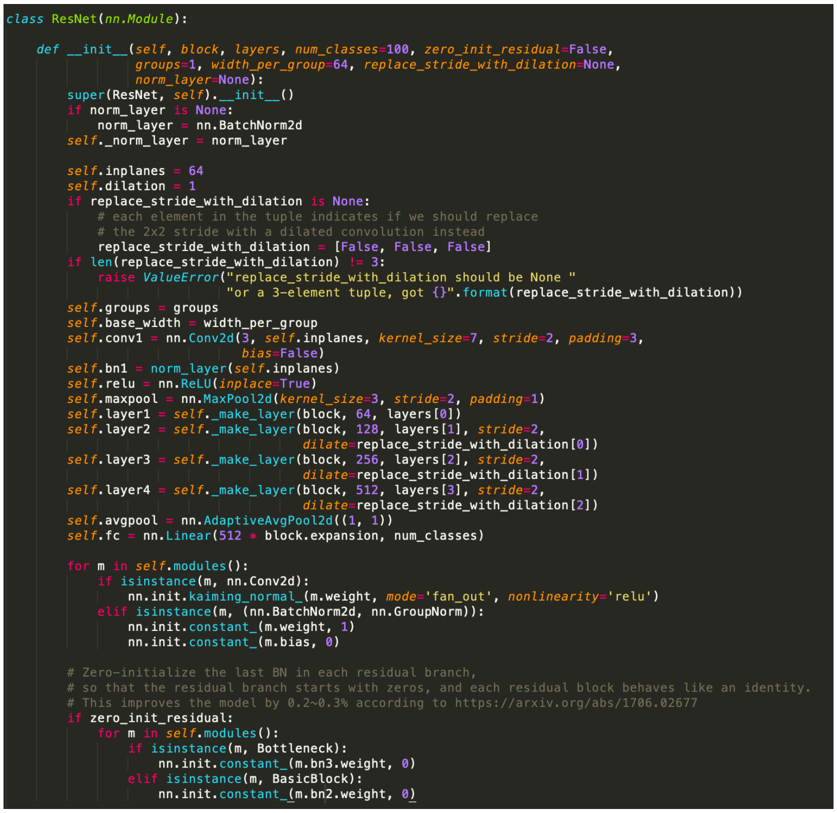
Finally, it turns out to be the prediction result by passing through average pooling and fully connected network.

Basic Block

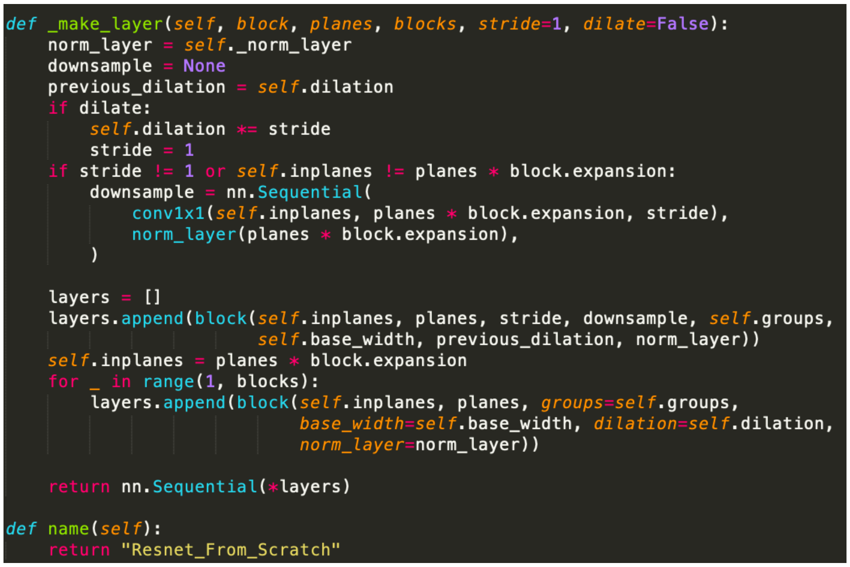


```
class BasicBlock(nn.Module):
    expansion = 1
    __constants__ = ['downsample']

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(BasicBlock, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError('BasicBlock only supports groups=1 and base_width=64')
        if dilation > 1:
            raise NotImplementedError("Dilation > 1 not supported in BasicBlock")
        # Both self.conv1 and self.downsample layers downsample the input when stride != 1
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```

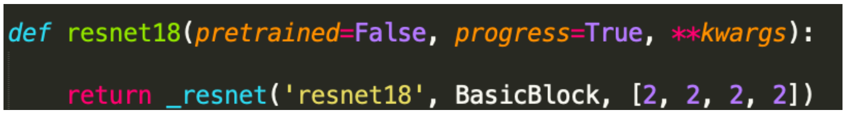
Resnet structure



```python
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=100, zero_init_residual=False,
                 groups=1, width_per_group=64, replace_stride_with_dilation=None,
                 norm_layer=None):
        super(ResNet, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        self._norm_layer = norm_layer

        self.inplanes = 64
        self.dilation = 1
        if replace_stride_with_dilation is None:
            # each element in the tuple indicates if we should replace
            # the 2x2 stride with a dilated convolution instead
            replace_stride_with_dilation = [False, False, False]
        if len(replace_stride_with_dilation) != 3:
            raise ValueError("replace_stride_with_dilation should be None "
                             "or a 3-element tuple, got {}".format(replace_stride_with_dilation))
        self.groups = groups
        self.base_width = width_per_group
        self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = norm_layer(self.inplanes)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2,
                                       dilate=replace_stride_with_dilation[0])
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2,
                                       dilate=replace_stride_with_dilation[1])
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2,
                                       dilate=replace_stride_with_dilation[2])
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

        # Zero-initialize the last BN in each residual branch,
        # so that the residual branch starts with zeros, and each residual block behaves like an identity.
        # This improves the model by 0.2~0.3% according to https://arxiv.org/abs/1706.02677
        if zero_init_residual:
            for m in self.modules():
                if isinstance(m, Bottleneck):
                    nn.init.constant_(m.bn3.weight, 0)
                elif isinstance(m, BasicBlock):
                    nn.init.constant_(m.bn2.weight, 0)
```

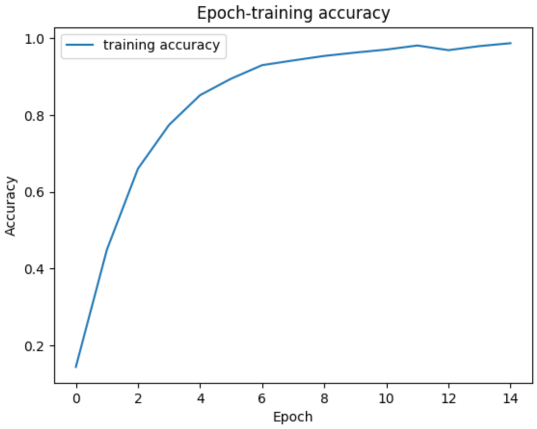


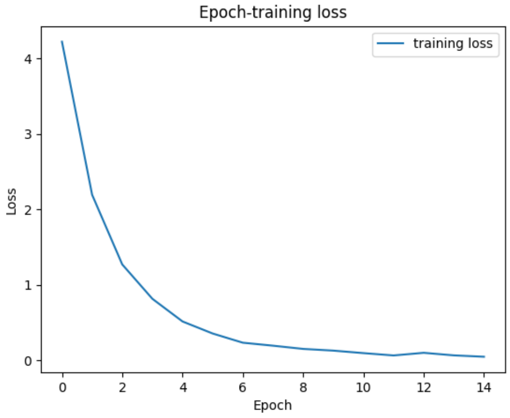
```python
def _make_layer(self, block, planes, blocks, stride=1, dilate=False):
    norm_layer = self._norm_layer
    downsample = None
    previous_dilation = self.dilation
    if dilate:
        self.dilation *= stride
        stride = 1
    if stride != 1 or self.inplanes != planes * block.expansion:
        downsample = nn.Sequential(
            conv1x1(self.inplanes, planes * block.expansion, stride),
            norm_layer(planes * block.expansion),
        )

    layers = []
    layers.append(block(self.inplanes, planes, stride, downsample, self.groups,
                        self.base_width, previous_dilation, norm_layer))
    self.inplanes = planes * block.expansion
    for _ in range(1, blocks):
        layers.append(block(self.inplanes, planes, groups=self.groups,
                            base_width=self.base_width, dilation=self.dilation,
                            norm_layer=norm_layer))

    return nn.Sequential(*layers)

def name(self):
    return "Resnet_From_Scratch"
```

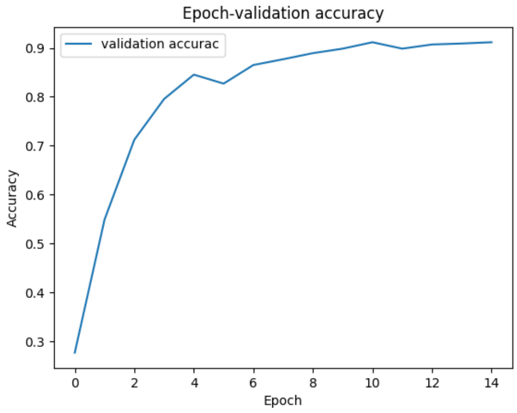


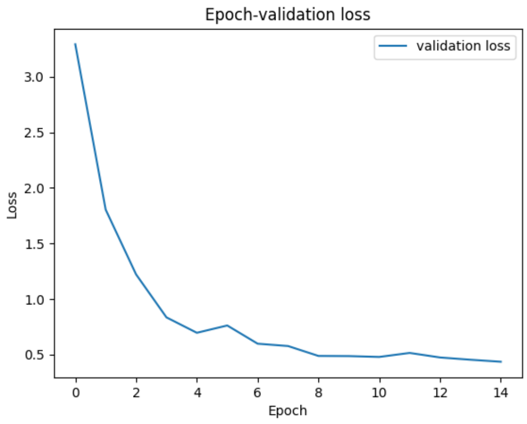
```python
def resnet18(pretrained=False, progress=True, **kwargs):

    return _resnet('resnet18', BasicBlock, [2, 2, 2, 2])
```

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





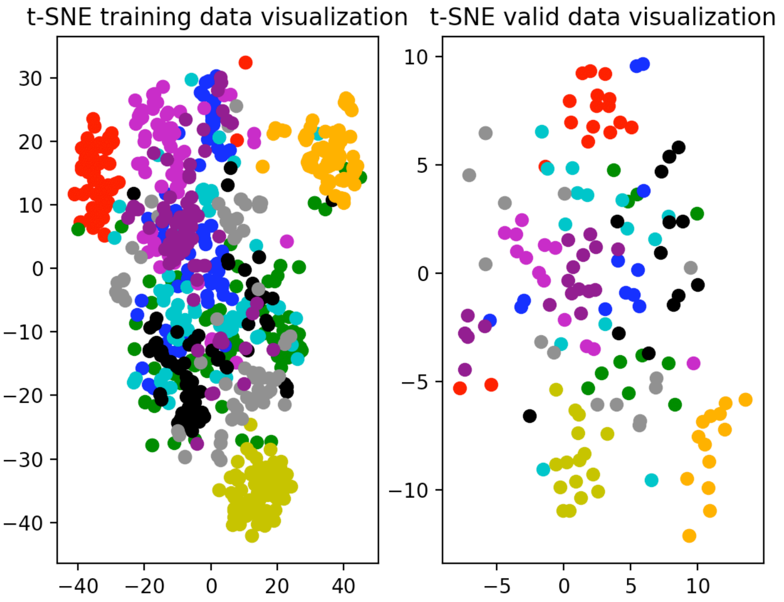




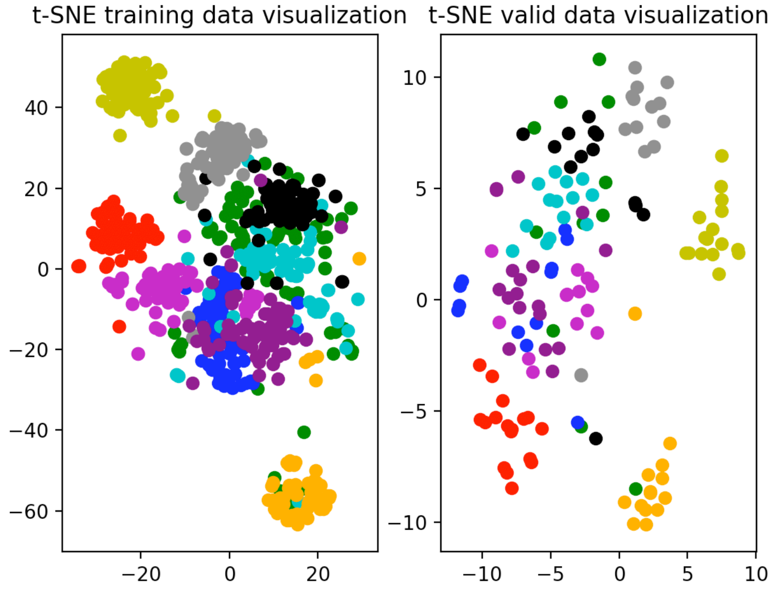
The training loss is approximately equal to validation loss, so I think it’s not underfitting nor overfitting.

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%)

ALEXNET:


t-SNE training data visualization t-SNE valid data visualization

MY MODEL:


t-SNE training data visualization t-SNE valid data visualization

The graphs above are visualization of feature extractions of Alexnet and my model perspectively. Alexnet extracts from “features” layer, and my model extracts from “layer3”.

We can see that the result of t-SNE of Resnet is better. I think it is mainly because the Resnet uses more convolution layer to get features of images, so when it projects data with t-SNE, the clustering are split more obviously.