SphereFace and CosFace

Zhuonan Lin, A53218121

May 30, 2019

A SphereFace for Face Verification

A.1 Evaluation SphereFace on LFW dataset

As the homework instructs, we download the LFW [1] dataset and the PyTorch code for ShpereFace [2], provided on the course website. Unzipping both the dataset and pretrained model, we can evaluate the SphereFace model on the LFW dataset by running:

```
python ./lfw_eval.py --model model/sphere20a_20171020.pth
```

The evaluation process and final result will be printed on the screen. The final accuracy for pretrained SphereFace on LFW dataset is 0.9918, with standard derivation and threshold of different folds are 0.0051 and 0.3095, respectively.

A.2 Evaluation Details on LFW dataset

A.2.1 Distance Metric

In *lfw_eval.py*: Line 135, the distance metric to measure the distance between two faces is calculated by:

```
1    cosdistance = f1.dot(f2)/(f1.norm()*f2.norm()+1e-5)
```

Here, $f1$ and $f2$ are the vectors for the two face images to be measured. This line of the code indicates that, as the variable name in the left hand side suggests, the distance between the two faces is measured by the cosine similarity, i.e. the cosine value of the angle between two face vectors. The small $\epsilon = 1e-5$ in the denominator is set to prevent dividing by zero. This metric is corresponding to [2],as it introduces in the **Testing** part in Section 4.1 in the paper:

> "For all experiments, the final representation of a testing face is obtained by concatenating its original face features and its horizontally flipped features. The score (metric) is computed by the ***cosine distance*** of two features."

A.2.2 Threshold and Accuracy Computation

The threshold for verification determination and the test accuraty computaion are done in *lfw_eval.py*: Line 141-148

```
folds = KFold(n=pairNum, n_folds=10, shuffle=False)
thresholds = np.arange(-1.0, 1.0, 0.005)
predicts = np.array([*map(lambda line:line.strip('\n').split(),
    ... predicts) ] )
for idx, (train, test) in enumerate(folds):
    best_thresh = find_best_threshold(thresholds, predicts[train]
        ...  )
    accuracy.append(eval_acc(best_thresh, predicts[test]))
    thd.append(best_thresh)
print('LFWACC={:.4f} std={:.4f} thd={:.4f}'.format(np.mean(
    ... accuracy), np.std(accuracy), np.mean(thd)))
```

From the code snippet, we know that the evaluation is carried out with a K-Fold style. There is a threshold candidate pool ranges from -1.0 to 1.0, with a step 0.005. For each fold, the *find_best_threshold()* function will be called, where each threshold candidate will be enumerated and the corresponding accuracy on the training set will be evaluated. The threshold that yields the best accuracy will be returned and subsequently used as threshold on this fold.

Accuracy evaluation need to be performed both in threshold determination and final test set evaluation. Here, the accuracy for a given data threshold is calculated by the function *eval_acc()*, where the predicted label with distance larger then the threshold will be regarded as 1 and otherwise 0. Finally, the accuracy will be calculated between the converted label and the ground truth and returned as the final accuracy result.

A.3 Face Alignment and Crop

A.3.1 Default Alignment and Crop

The face alignment in our code, is done by the function *alignment()* in *lfw_eval.py*:Line 11-26

```
def alignment(src_img, src_pts):
    ref_pts = [ [30.2946, 51.6963],[65.5318, 51.5014],
        [48.0252, 71.7366],[33.5493, 92.3655],[62.7299, 92.2041]
            ... ]
    crop_size = (96, 112)
    src_pts = np.array(src_pts).reshape(5,2)

    s = np.array(src_pts).astype(np.float32)
    r = np.array(ref_pts).astype(np.float32)

    tfm = get_similarity_transform_for_PIL(s, r)
```

```
11
12      face_img = src_img.transform(crop_size, Image.AFFINE,
13              tfm.reshape(6), resample=Image.BILINEAR)
14      face_img = np.asarray(face_img )[:, :, ::-1]
15
16      return face_img
```

Here, the *ref_pts* are the target positions of landmarks after alignment, which have been determined during the SphereFace training process. Then, the transform will be calculated using source points (*src_pts*) and target points by the function *get_similarity_transform_for_PIL()*. Given this transform (*tmf*), the original image *src_img* will be transformed, together with the bilinear interpolation and cropped into desired size (*crop_size*). The final resultant image (*face_img*) will be warped and returned.

A.3.2 Crop Without Alignment

In this part, we will try to evaluate the SphereFace withour face alignment. Instead, we will crop a image patch, with same size used in face alignment, at the center of each face image. In our code, we will first fill in the *cropping()* function defined in *lfw_eval.py*: Line 28 as following, and two cropping examples for the alphabetically first two face images in LFW dataset are shown in Figure 1.

```
1  def cropping(src_img ):
2      # IMPLEMENT cropping the center of image
3       # Define crop_size as required
4      crop_size = (112, 96)
5      # convert original input to array
6      s = np.asarray(src_img)
7      # dicide image size (H: height and W: width)
8      H, W, _ = np.asarray(s).shape
9      # get the four boundaries index, center at image center
10     bottom, top = int(H/2.+crop_size[0]/2.), int(H/2.-crop_size
           ... [0]/2.)
11     right, left = int(W/2.+crop_size[1]/2.), int(W/2.-crop_size
           ... [1]/2.)
12     # crop with boudaries above, and convert to array for
           ... returning
13     face_img = np.asarray(s[top:bottom, left:right, :])
14
15     return face_img
```

With this cropping function, we can then evaluate the ShpereFace on LFW dataset again without face alignment but center cropping by running:

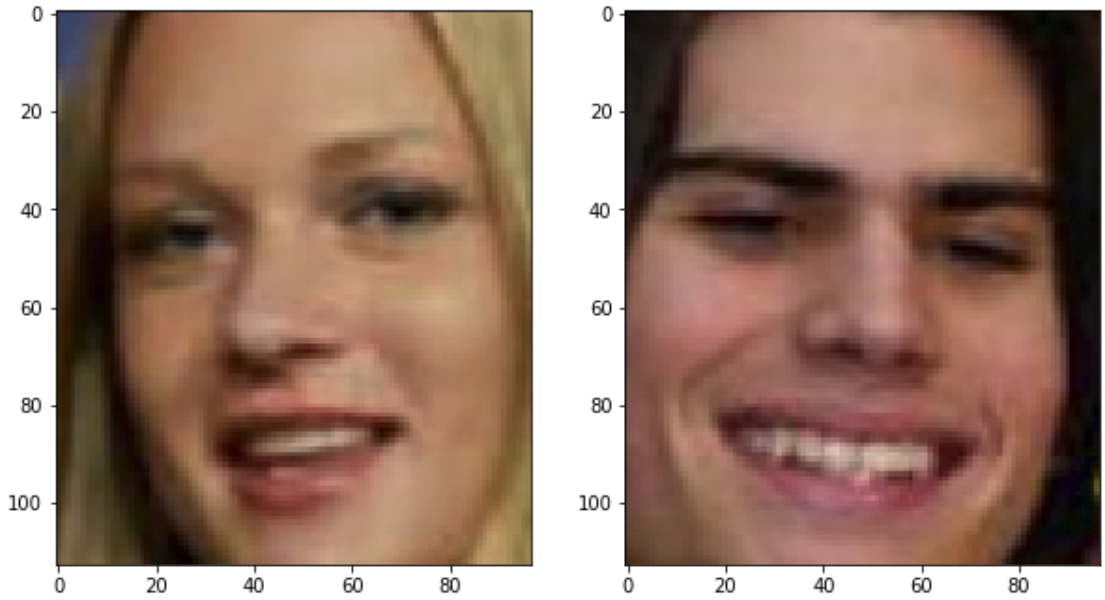

Figure 1: Cropping examples with center at image center and a height and width of 112 and 96 pixels, respectively.

python ./lfw_eval.py --model model/sphere20a_20171020.pth --alignmentMode 0

The evaluation process and final result will be printed on the screen. With this center cropping, the final accuracy for pretrained SphereFace on LFW dataset is 0.8297, with standard derivation and mean threshold of different folds are 0.0252 and 0.2440, respectively. The results are worse than those with face alignment as we can expect.

B MTCNN Face Landmark Detection

B.1 MTCNN Face Landmark Detection Examples

In this part, we use the MTCNN method [3] for pre-processing the face detection and alignment. After downloading the modified codes from course website, we run *lfw_landmark.py* to perform MTCNN on LFW dataset. The results will be written in *../sphereFace/data/lfw_landmarkMTCNN.txt*, in the form for further SphereFace inputting. Two example output lines are as following:

AJ_Cook/AJ_Cook_0001.jpg 103.672 114.958 146.303 109.461 118.223 135.837 110.155 158.646 149.281 154.148
AJ_Lamas/AJ_Lamas_0001.jpg 104.424 111.958 149.001 118.325 130.224 134.433 97.063 152.340 147.845 158.792

where the numbers following the image name are the coordinates of landmarks found by MTCNN. The detection results of the above two face images are shown below (this can be

done separately by a simple piece of codes using the *detect_faces* and *show_bboxes* methods from MTCNN).

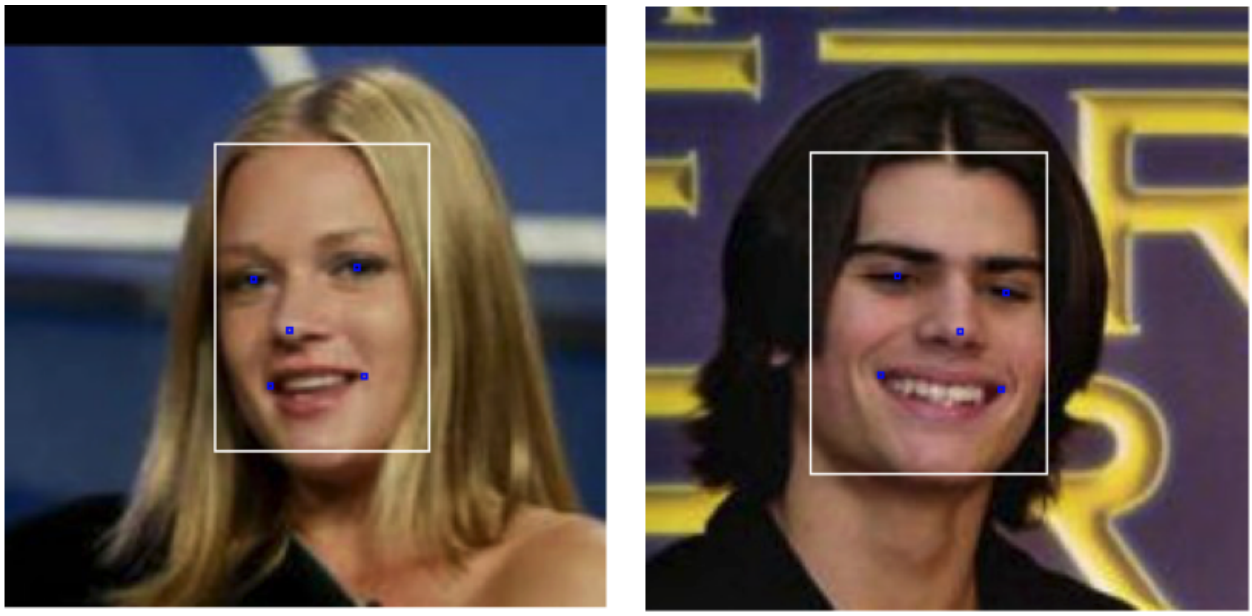

Figure 2: Face landmarks detection result examples using MTCNN. Landmarks are shown together with the bounding box also from MTCNN dectector.

B.2 SphereFace with MTCNN Landmarks

With landmarks detected from MTCNN in the previous section, we can re-run the evaluation of SphereFace on LFW dataset, with the following line:

```
python ./lfw_eval.py --model model/sphere20a_20171020.pth --alignmentMode 2
```

The evaluation process and final result will be again printed on the screen. This time, the final accuracy with MTCNN pre-detected landmarks is 0.9845, with standard derivation and threshold of different folds are 0.0057 and 0.2995, respectively.

B.3 Comparison to Original Results

Compare to the evaluation results to the one with default given face landmarks, our result with our MTCNN dectected landmarks is not better, and even worse by 0.73%. One possible reason is that, although [2] claims that they also used MTCNN detected landmarks in the whole network, the details about the application is unknown to us. Their MTCNN model may be trained on different dataset so that the detected results are possibly slightly different from and more accurately than the pre-trained model we used, so that it out-performs a bit in A than B. Also, the crop procedure in the given pre-trained SphereFace is directly related

to the training process (e.g. the pre-determined *ref_pts* in *alignment()* function). Therefore, we think there are some possible ways to improve out MTCNN + SphereFace results. First, we cloud retrain the MTCNN on larger dataset for better performance. Second, we can try to relate MTCNN with ShpereFace in training stage by ourselves, which can give better reference points for cropping and alignment.

B.4 Real-time Speed in MTCNN

One advantage of MTCNN is that the network can achieve the state-of-art accuracy in real-time. The real-time speed is achieved due to three main reasons: **carefully designed cascaded CNNs architecture, online hard sample mining strategy, and joint face alignment learning**.

First, The MTCNN completes its joint detection and alignment with a multistage cascaded CNN. There are 3 stages, where each has a individual CNN to achieve the goal step by step, namely, proposal network (P-Net, a fully convolutional network), refine network (R-Net) and output network (O-Net). P-Net will obtain all the candidate facial windows and their bounding box regression vectors. R-Net will refine the candidated and reject a large number of false ones. O-Net will identify face regions and five facial landmarks' positions. Note that the input for the whole structure is the image pyramid, which consists of different scales of the resized original image. In these three CNNs, 3×3 **filters** are used instead of previous 5×5 filters, which can reduce computing for less run time, while increase the depth to get better performance. Consequently, given this cascade structure with lightweight CNNs, MTCNN method can achieve real time speed in joint face detection and alignment task.

Second, in the MTCNN method, the authors came up with a online hard sample mining strategy in face/nonface classification task. To do this, in each minibatch, the losses computed in the forward propagation from all samples are sorted and the top 70% of them will be selected as hard samples. Only the gradients from these hard samples are calculated in the backward propagation. This online hard mining strategy ignores the easy samples that are less helpful to strengthen the detector during training and also help the runtime performance.

Last, the face alignment is learning jointly with detection. In training process, the three loss functions from individual CNN (face classification loss, bounding box regression loss, and facial landmark localization loss) will compose the final loss together as a multisource training. The final loss is tuned by some task-importance coefficients at different stages. This joint training strategy will simultaneously learn the detection and alignment tasks, and therefore improve the performance for less time.

B.5 Non-maximal suppression (NMS) in MTCNN

Non-maximal suppression (NMS) is used both in P-Net and R-Net. In P-Net, NMS is employed non-maximum to merge highly overlapped face window candidates. In R-Net, NMS is used to help further reject a large number of false candidates. In the MTCNN code, the NMS precedures are carried out in *src/box_utils.py*: Line 5-68.

```python
def nms(boxes, overlap_threshold=0.5, mode='union'):
    """Non-maximum suppression.

    Arguments:
        boxes: a float numpy array of shape [n, 5],
            where each row is (xmin, ymin, xmax, ymax, score).
        overlap_threshold: a float number.
        mode: 'union' or 'min'.

    Returns:
        list with indices of the selected boxes
    """

    # if there are no boxes, return the empty list
    if len(boxes) == 0:
        return []

    # list of picked indices
    pick = []

    # grab the coordinates of the bounding boxes
    x1, y1, x2, y2, score = [boxes[:, i] for i in range(5)]

    area = (x2 - x1 + 1.0)*(y2 - y1 + 1.0)
    ids = np.argsort(score)  # in increasing order

    while len(ids) > 0:

        # grab index of the largest value
        last = len(ids) - 1
        i = ids[last]
        pick.append(i)

        # compute intersections
        # of the box with the largest score
        # with the rest of boxes

        # left top corner of intersection boxes
        ix1 = np.maximum(x1[i], x1[ids[:last]])
        iy1 = np.maximum(y1[i], y1[ids[:last]])

        # right bottom corner of intersection boxes
        ix2 = np.minimum(x2[i], x2[ids[:last]])
```

```
44          iy2 = np.minimum(y2[i], y2[ids[:last]])
45
46          # width and height of intersection boxes
47          w = np.maximum(0.0, ix2 - ix1 + 1.0)
48          h = np.maximum(0.0, iy2 - iy1 + 1.0)
49
50          # intersections' areas
51          inter = w * h
52          if mode == 'min':
53              overlap = inter/np.minimum(area[i], area[ids[:last]])
54          elif mode == 'union':
55              # intersection over union (IoU)
56              overlap = inter/(area[i] + area[ids[:last]] - inter)
57
58          # delete all boxes where overlap is too big
59          ids = np.delete(
60              ids,
61              np.concatenate([[last], np.where(overlap >
                    ... overlap_threshold)[0]])
62          )
63
64      return pick
```

The basic idea for NMS used here is to **iteratively pick the current maximum score box (*boxes*$[i,:]$) and eliminate the boxes whose overlap with *boxes*$[i,:]$ is larger than the threshold (*over_lapthreshold*), until all candidates (in *boxes*) are either picked or eliminated.**

To be specific, in the code:

1. First we get the index list that has the order sorted by the score of each box (*ids*). While we still have box that neigther picked nor eliminated ($len(ids) > 0$), we picked the current largest score box ($ids[last]$) to be one of the remianing box to return (i.e. *pick.append(ids[last])*.

2. Then we carry on the NMS filter based on the current largest score box. Given the current max score box, we find the index of left top and right bottom corner (*ix1, iy1, ix2, iy2*)of the intersection area by calculating the maximum and minimum of x and y indexes among the highest score box and others. (Line: 38-48 above). Given *ix1, iy1, ix2, iy2*, **the ratio of intersection area to the box area** can be calculated. In the actual code, there are two types of intersection area, namely *min* and *union*, which calculates minimum overlap or the total union overlap, as the names suggest, respectively. Using the overlap area, each other box will be filter by whether its overlap with highest score box is larger than the threshold (*overlap_threshold*).

3. Repeat 1 and 2 until all boxes are either picked or eliminated. The picked boxes' indexes in *pick* will be returned, others are filtered by NMS method. This can reduced

the number of candidate facial window effectively, and thus help accelerate the overall speen of MTCNN.

C Training SphereFace on CASIA Dataset

In this part, we will train the SphereFace using the angular softmax loss on CASIA dataset [4], and evaluate the performance on LFW dataset.

C.1 A-Softmax Loss and ψ Function

To train the nerual network, we will follow the A-softmax loss function defined by Eq.(7) in [2]:

$$L_{ang} = \frac{1}{N}\sum_{i} -log(\frac{e^{||\mathbf{x}_i||\psi(\theta_{y_i,i})}}{e^{||\mathbf{x}_i||\psi(\theta_{y_i,i})} + \sum_{j\neq y_i} e^{||\mathbf{x}_i||cos(\theta_{j,i})}}) \tag{1}$$

where

$$\psi(\theta_{y_i,i}) = (-1)^k cos(m\theta_{y_i,i}) - 2k \tag{2}$$

with $\theta_{y_i,i} \in [\frac{k\pi}{m}, \frac{(k+1)\pi}{m}], k \in [0, m-1]$ and $m \geq 1$.

In *sphereface/faceNet.py*, we implement Eq.(2) under class *CustomLinear* and Eq.(1) under class *CustomLoss* respectively. The following snippets showing the the specific implementation with PyTorch:

Implement *psi_theta*

```
    # def forward(self, input):
        # ...

        # IMPLEMENT phi_theta
        # Use multi-angle formulae to calculte cos(m*theta)
        # these fornulae given in self.mlambda
        cos_m_theta = self.mlambda[self.m](cos_theta)
        # Calculate the hyper-parameter k
        # k should be in the [0, m-1] and
        # theta should be in [k*pi/m, (k+1)*pi/m]
        # here in the forward process, we choose k that makes
            ... theta = k*pi/m
        theta = Variable(cos_theta.data.acos())
        k = (self.m * theta / np.pi).floor()
        # Finally, calculate phi using the equation in SphereFace
            ...  paper
        phi_theta = (-1) ** (k) * cos_m_theta - 2 * k

        # ...
```

***Implement** Loss*

```
    #def forward(self, input, target):
        # ...

        # IMPLEMENT loss
        # Construct a tensor with index-i equals 1 ohterwise 0
        #    Initialize with all 0 in index tensor, then fill in ... 1 in corresponding places
        index = cos_theta * 0.0
        index.scatter_(1, target.data.view(-1, 1), 1)
        #    Convert index to int for further use
        index = Variable(index.to(torch.uint8))


        # Notice the difference between the angular softmax and ... regular softmax
        # Prepare the input for softmax by replace cos_function ... with phi-function
        #   at index-i (already calculated in index)
        output = cos_theta * 1.0
        #   Here, it turns out to be crucial to use the annealing ...  optimization
        #    in Appendix G in SphereFace Paper (for more ... discussion, see the assignment report)
        #   We use lambda starting from LambdaMax and gradually ... decay to LambaMin
        #   for the decay rate we choose the one based on https ... ://github.com/clcarwin/sphereface_pytorch
        lamb = max(self.LambdaMin, self.LambdaMax / (1+0.1*self. ... it))
        # Implement annealing
        output[index] -= cos_theta[index] / (1. + lamb)
        output[index] += phi_theta[index] / (1. + lamb)


        # Use softmax from Pytorch for loss function
        logpt = F.log_softmax(output)
        # Calculate the loss functiong use eq.(7) in SphereFace ... paper
        loss = (-logpt.gather(1, target).view(-1)).mean()

        # ...
```

Here, we choose hyper-parameters such as k based on the reference of [5]. To be spe-

cific, $k = \frac{m\theta_{y_i,i}}{\pi}$. Another technique we use is annealing optimization in A-Softmax loss, as introduced in Appendix G in [2], where we let

$$f_{y_i} = \frac{\lambda||\mathbf{x}_i||cos(\theta_{y_i}) + ||\mathbf{x}_i||\psi(\theta_{y_i})}{1+\lambda} \tag{3}$$

Particularly, as used in [2], λ starts from a large value (1500) and gradually decreased (divided by $(1 + 0.1 \times iteration_number)$ to a small value (5). The reason that this annealing strategy is essential, as described in [6], is:

> "...when training data has too many subjects (such as **CASIA-WebFace dataset**), the convergence of L-Softmax will be more difficult than softmax loss."

and this reason also applys on A-Softmax loss.

C.2 Training Process and Evaluation Results

Now we can train the faceNet with A-Softmax loss by running *casia_train.py*. With the help of annealing optimization, the requirement evaluation accuracy on LFW ($> 90\%$) can be easily achieved just using the default arguments. Additionally, here we show 3 training process with different m and *iteration_number* in Figure 3 and corresponding evaluation results on LFW in Table 1.

m value, iterations	$m = 3$, 28k	$m = 3$, 40k	$m = 4$, 28k	$m = 4$, 40k
Accuracy	**0.9850**	0.9830	0.9757	0.9720

Table 1: Evaluation results on LFW. Evaluation Results include model trained with different m (3 vs 4) and different iteration numbers (28k vs 40k).

Here, all evaluations on LFW are have good performance ($\geq 97.2\%$). One thing that is different to the experiments in [2] is that, our results show that $m = 3$ has better evaluation accuracy than $m = 4$. However, this is understandable, since larger m is more difficult to train. Actually, from Figure 3 and the comparison between the evaluation accuracy of 28k and 40k training, we can see that the learning rate is too large in the last several thousand iterations, so that for $m = 4$, it is very possible that the training is not good enough for the used learning rate. **One thing we can try to make better performance is to use smaller learning rates and smaller decay coefficients.**

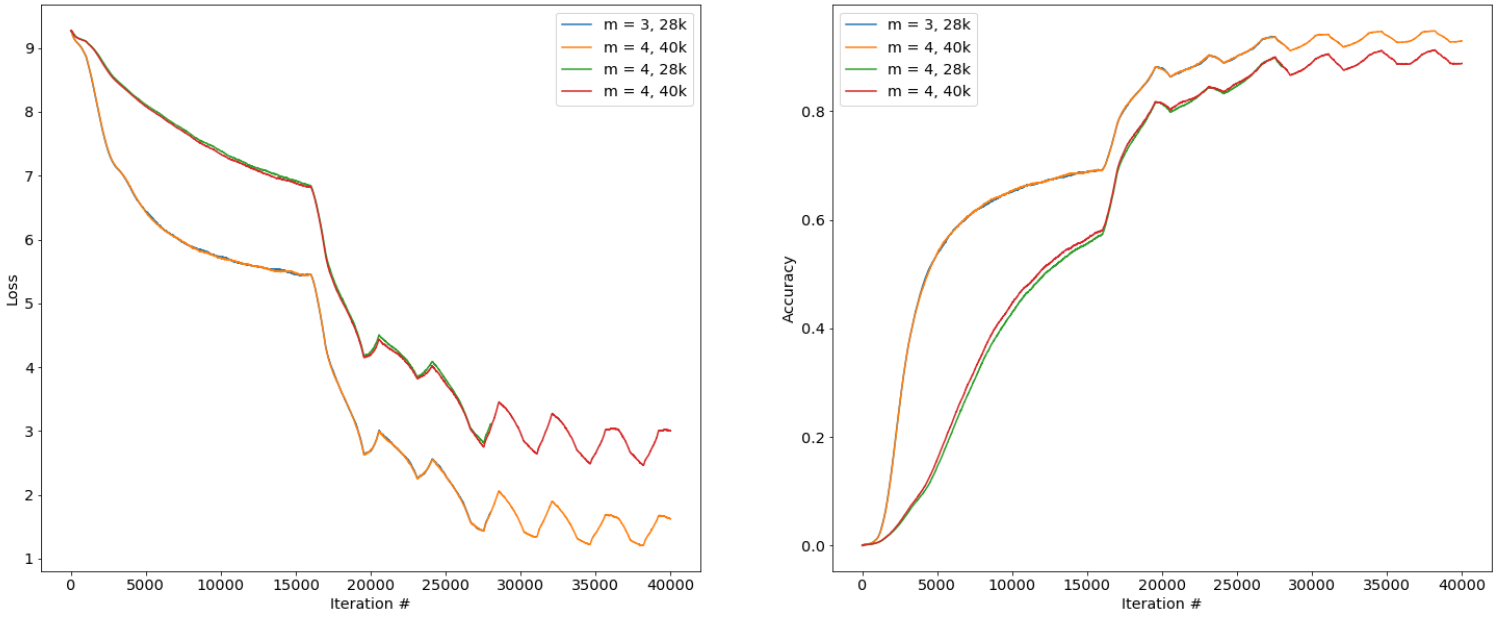

Figure 3: Training process of (a)loss and (b)accuracy of ShereFace net, including model trained with different m (3 vs 4) and different iteration numbers (28k vs 40k). **All curves are smoothed by a quadratic interloation.**

C.3 Batch Normalization Layers

In this part, we will add batch normalization (BN) layers to the original 20-layer residual network. The BN layers will be added after each convolution layer and fully connected layer. The specific way we do this with PyTorch in code, is to both add the layer ($BatchNorm2d()$ after convolutional layer and $BatchNorm1d()$ after fully connected layer) in FaceNet model, and also add the explicit implementations as a layer in *forward()* step, an example for the pair described above is:

Add BN layers in model

```
        self.conv1_1 = nn.Conv2d(3,64,3,2,1) #=>B*64*56*48
        self.bn1_1 = nn.BatchNorm2d(64)
        self.relu1_1 = nn.PReLU(64)
        self.conv1_2 = nn.Conv2d(64,64,3,1,1)
        self.bn1_2 = nn.BatchNorm2d(64)
        self.relu1_2 = nn.PReLU(64)
        self.conv1_3 = nn.Conv2d(64,64,3,1,1)
        self.bn1_3 = nn.BatchNorm2d(64)
        self.relu1_3 = nn.PReLU(64)
```

Implement BN layers in** forward() **function

```
1         x = self.relu1_1(self.bn1_1(self.conv1_1(x)))
2         x = x + self.relu1_3(self.bn1_3(self.conv1_3(self.relu1_2
              ... (self.bn1_2(self.conv1_2(x))))))
```

C.3.1 Training Process and Evaluation Results with Batch Normalization

With the new model that has batch normalization layers, we now retrain the network to see the influence of batch normalization. Here, we show 2 different training results with different total iteration number in Figure 4, and the corresponding evaluation results on LFW dataset in Table 2. Note that in this part, all training are based on fixed m value ($m = 4$), since this part aims mainly on how the BN layers influence the model. Therefore, the best evaluation here (m = 4, w/ BN, 28k) is not better than the best evaluation in previous part (m = 3, w/o BN, 28k).

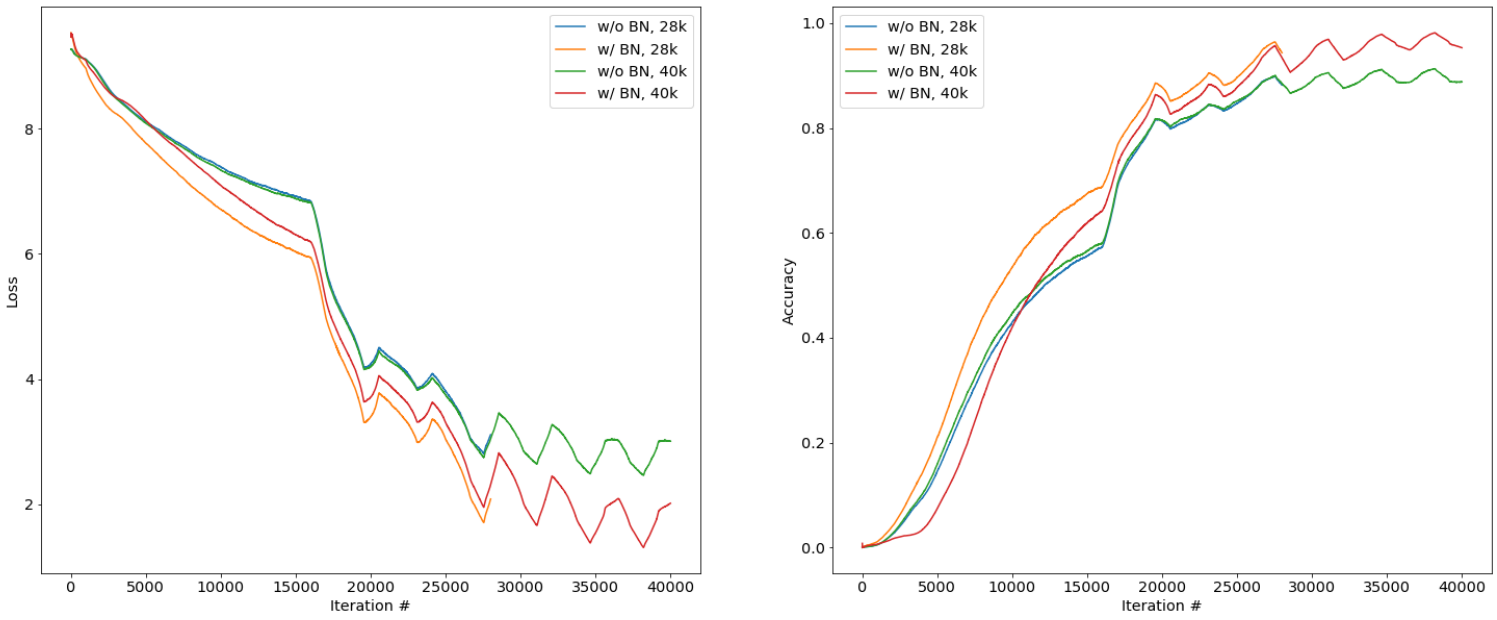

Figure 4: Training process of (a)loss and (b)accuracy of ShereFace net, including model trained with and without batch normalization (BN) layers and with different iteration numbers (28k vs 40k). **All curves are smoothed by a quadratic interpolation.**

model, iterations	w/o BN, 28k	w/ BN, 28k	w/o BN, 40k	w/ BN, 40k
Accuracy	0.9757	**0.9808**	0.9720	0.9732

Table 2: Evaluation results on LFW. Training with $m = 4$ recommended in [2]. Evaluation Results include model trained with and without batch normalization (BN) layers and with different iteration numbers (28k vs 40k). **All curves are smoothed by a quadratic interpolation.**

C.3.2 Comparison to Original Model

The batch normalization method is widely used in deep neural networks. The basic idea is to normalize the batch samples into a Gaussian distribution by

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}} \tag{4}$$

while also allows the network to squash the range if it wants to by

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)} \tag{5}$$

where $\gamma^{(k)} = \sqrt{Var[x^{(k)}]}$ and $\beta^{(k)} = E[x^{(k)}]$ can be learned by the network.

From Table 2, we can see that **with BN layers, the evaluation will have better performance** both for smaller and larger number of iterations. There are several reasons that the batch normalization (BN) can help the performance of a deep nueral network. First, BN layers limit the covariate shift by normalizing the activations of each layer, which results in learning on a more stable distribution of inputs at each layer, and also accelerates the training of the network. Second, BN makes the mean and variance of the activations independent of the values, which leads to the magnitude of the higher order interactions are going to be suppressed, allowing larger learning rates to be used. Third, the form of Gaussian distribution in BN can improve the gradient flow through the network. These all togeter makes the deep neural networks with BN layers usually outperform ones without them (Reference: [7]).

C.4 Visualization with tSNE

One interesting result to evaluate is how the network with loss function after training can separate the different categories samples. Here to visualize the result, we will implement the tSNE on the feature vectors given by our trained network on 10 random chosen identities. We will use the tSNE codes from *sklearn* package. The script that can automatically choose random identities, feed forward the data and visualize dimension-reduced feature vector with tSNE is as following:

```
1 import torch
2 from torch.autograd import Variable
3 import faceNet, dataLoader
4 from torch.utils.data import DataLoader
```

```
5
6  import os, random
7  import numpy as np
8  import matplotlib.pyplot as plt
9  from sklearn.manifold import TSNE
10
11 def prepare_data(identities_num = 10):
12     """
13     Prepare images and corresonding landmarks of 10 random
       ... identities
14     """
15     imageRoot = '../CASIA-WebFace/'
16     people = [f for f in os.listdir(imageRoot)]
17     identities = random.sample(people, 10)
18
19     alignmentRoot = './data/casia_landmark.txt'
20     with open(alignmentRoot) as file:
21         name = ''
22         landmarks = {}
23         for line in file:
24             new = line.split('/')[0]
25             if new == name:
26                 landmarks[name].append(line)
27             else:
28                 landmarks[new] = [line]
29                 name = new
30     record = []
31     identities_img_number = []
32     for i in identities:
33         identities_img_number.append(len(landmarks[i]))
34         for line in landmarks[i]:
35             record.append(line)
36     batch_size = len(record)
37
38     new_landmark = './data/casia_landmark_' + str(identities_num)
       ...  + '_identities.txt'
39     file = open(new_landmark, 'w')
40     for line in record[:-1]:
41         file.write(line)
42     file.write(record[-1][:-2])
43     file.close()
44
45     new_dic = '../CASIA-WebFace_' + str(identities_num) + '
       ... _identities/'
```

```
46      os.system('rm -rf ' + new_dic)
47      os.system('mkdir ' + new_dic)
48
49      for i in identities:
50          os.system('cp -r ../CASIA-WebFace/' + i + ' ' + new_dic +
                ...  i)
51
52      assert(sum(identities_img_number) == batch_size)
53      return identities, identities_img_number, batch_size, new_dic
            ... , new_landmark
54
55
56  identities_num = 10
57  identities, counts, batch_size, imageRoot, alignmentRoot =
        ... prepare_data(identities_num)
58  faceDataset = dataLoader.BatchLoader(
59          imageRoot = imageRoot,
60          alignmentRoot = alignmentRoot,
61          cropSize = (96, 112)
62          )
63  faceLoader = DataLoader(faceDataset, batch_size = batch_size,
        ... num_workers = 16, shuffle = False )
64  imBatch = Variable(torch.FloatTensor(batch_size, 3, 112, 96))
65
66  net = getattr(faceNet, 'faceNet')()
67  net.load_state_dict(torch.load("./checkpoint_bn/netFinal_8.pth")
        ... )
68  net.eval()
69  net.feature = True
70
71  for i, dataBatch in enumerate(faceLoader):
72          image_cpu = dataBatch['img']
73          imBatch.data.resize_(image_cpu.size() )
74          imBatch.data.copy_(image_cpu )
75          output = net(imBatch)
76          break
77
78  ff = output.data.cpu().numpy()
79  visualization = TSNE(n_components=2).fit_transform(ff)
80
81  # Note that when the dataLoader.BatchLoader read in images, it
        ... shuffles the order
82  # To visualize the tSNE result corresponding to the correct
        ... identity, we need to
```

```
# restore the shuffle using the order in dataLoader.BatchLoader.
    ... perm
dic = {x:[] for x in range(10)}
perm = np.array(faceDataset.perm)
cum_counts = [-1] + list(np.array(counts).cumsum() - 1)
for i in range(identities_num):
    dic[i] = list(np.where((perm >= cum_counts[i]+1) & (perm <=
        ... cum_counts[i+1]))[0])
    assert(len(dic[i]) == counts[i])

figure = plt.figure(figsize=(16, 12))
for i in range(identities_num):
    plt.scatter(visualization[dic[i],0], visualization[dic[i],1])
plt.legend(identities)
figure.savefig('sphere tSNE.png')
```

With the script above, we can choose then random identities for the tSNE visualization. After running several times of the experiment, we show the tSNE visualization in Figure 5, where we can clearly see that different person's feature vector will cluster seperately in the tSNE reduced space.

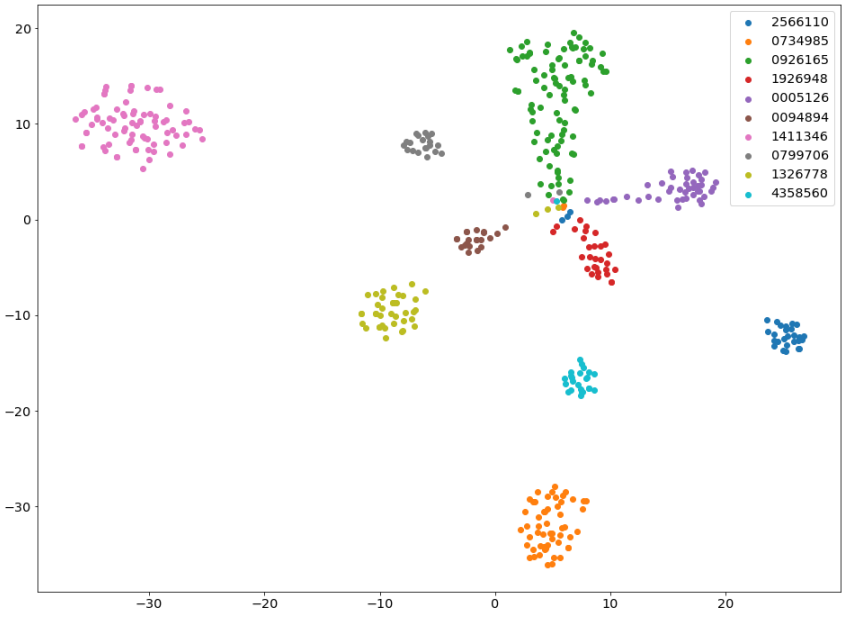

Figure 5: tSNE visualization of 10 random chosen identities, whose images are fed forward through the trained SphereFace net.

D Training CosFace on CASIA Dataset

In this section, we will use the loss function in CosFace [8] for the similar tasks in C, including training, evaluation and visualization with tSNE.

D.1 Cosine Loss and ψ Function

Similar to the way we did in C, we first follow the cosine loss function defined by Eq.[4] in [8]:

$$L_{lmc} = \frac{1}{N}\sum_{i} -log(\frac{e^{s\,\psi(\theta_{y_i,i})}}{e^{s\,\psi(\theta_{y_i,i})} + \sum_{j\neq y_i} e^{s\,cos(\theta_{j,i})}}) \tag{6}$$

where

$$\psi(\theta_{y_i,i}) = cos(\theta_{j,i}) - m \tag{7}$$

Again, in *cosface/faceNet.py*, we implement Eq.(7) under class *CustomLinear* and Eq.(6) under class *CustomLoss* respectively. The following snippets showing the the specific implementation with PyTorch:

Implement *psi_theta*

```
    # def forward(self, input):
        # ...


        # IMPLEMENT phi_theta
        # phi = s * (cos_theta - m), in this code, s is defined ... and used in CustomLoss class
        phi_theta = cos_theta - self.m

        # ...
```

Implement *Loss*

```
    # def forward(self, input, target):
        # ...

        # IMPLEMENT loss
        # Construct a tensor with index-i equals 1 ohterwise 0
        #    Initialize with all 0 in index tensor, then fill in ... 1 in corresponding places
        index = cos_theta * 0.0
        index.scatter_(1, target.data.view(-1, 1), 1)
        #    Convert index to int for further use
        index = Variable(index.to(torch.uint8))

```

```
        # Prepare the input for softmax by replace cos_function ... with phi-function
        #   at index-i (already calculated in index)
        # at [index], output is phi_theta otherwise cos_theta, ... this can be simply achived use torch.where()
        output = torch.where(index, phi_theta, cos_theta)
        # Use softmax from Pytorch for loss function
        logpt = F.log_softmax(self.s * output)
        # Calculate the loss functiong use equation from CosFace ... paper
        loss = (-logpt.gather(1, target).view(-1)).mean()

        # ...
```

D.2 Training Process and Evaluation Results with Cos-Loss

With the loss function defined above, we copied the exactly same network structure that we used in C, and trained the network with the Cos-Loss. Here, to compare with SphereFace fair and effectively, we train all the CosFace net here with **same learning rate and iteration numbers** (28k) **to the ones used in the best SphereFace net training**. The main aim this part is to play with the loss related hyperparameters (s and m), so that in the following experiments we will fix the learning rate and iteration number, and test with different combinations of (s, m). We choose two s values, which are the default value used in *casia_train.py* ($s = 30$) and in the *CustomLoss* class constructor($s = 64$), and the choices of m are based on [9]. The training process is shown in Figure 6, and the the corresponding evaluation results are shown in Table 3.

s, m	$s = 30, m = 0.35$	$s = 30, m = 0.40$	$s = 64, m = 0.35$	$s = 64, m = 0.40$
Accuracy	**0.9902**	0.9882	0.9868	0.9885

Table 3: Evaluation results of FaceNet with Cos-Loss. All training process has same learning rate and iteration number to the best SphereFace net we have in Sec. C

D.3 Compared to SphereFace

Compare the best results we have from the CosFace and SphereFace nets, we can see that the CosFace outperforms the SphereFace by a margin about 1%. Since we makes the training procedure to be same in both cases, the better performance is due to the difference between A-Softmax and Cosine Loss function, as discussed in [8, 2].

A-softmax improves the softmax loss by introducing an extra margin, such that its decision boundary is given by:

$$C_1 : cos(m\theta_1) \geq cos(\theta_2)$$

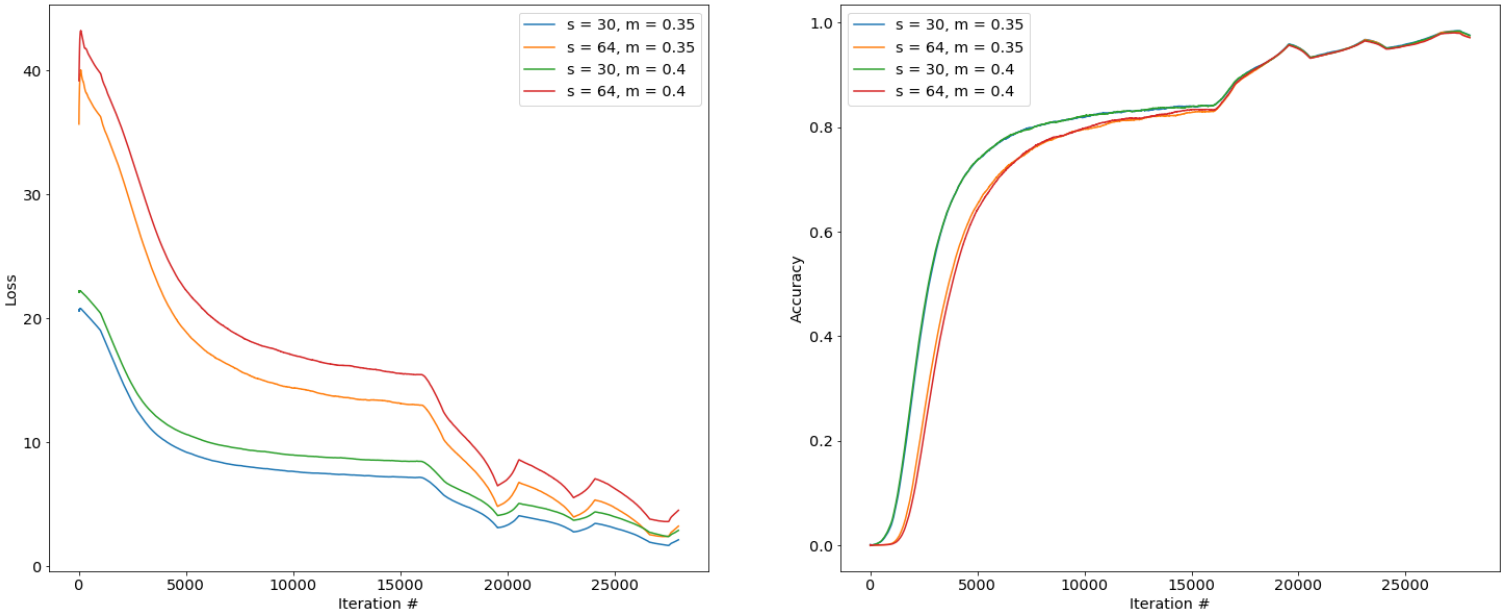

Figure 6: Training process of (a)loss and (b)accuracy of CosFace net, including model trained with different combinations of s and m. **All curves are smoothed by a quadratic interpolation.**

$$C_1 : cos(m\theta_2) \geq cos(\theta_1)$$

thus, the decision margin that classifies the two classes are $\theta_1 \leq \frac{\theta_2}{m}$ for C_1 and $\theta_2 \leq \frac{\theta_1}{m}$ for C_2.

On the other hand, for Cosine loss, the decision margin is defined in the cosine space rather than the angle space like A-softmax, thus, the boundary is given by:

$$C_1 : cos(\theta_1) \geq cos(\theta_2) + m$$

$$C_1 : cos(\theta_2) \geq cos(\theta_1) + m$$

Therefore, $cos(\theta_1)$ is maximized while $cos(\theta_2)$ being minimized for C_1 and similarly for C_2 to perform large-margin classification, and as a result, a clear margin ($\sqrt{2}m$) in the produced distribution of the consine of angle. An illustrative figure to compare A-softmax and Cosine loss is shown in Figure 7. Consequently, given the same network structure, using Cosine loss function can be more robust than A-softmax and yield a better evaluation result on LFW dataset. Actually, this is consistent to those results achieved in [8].

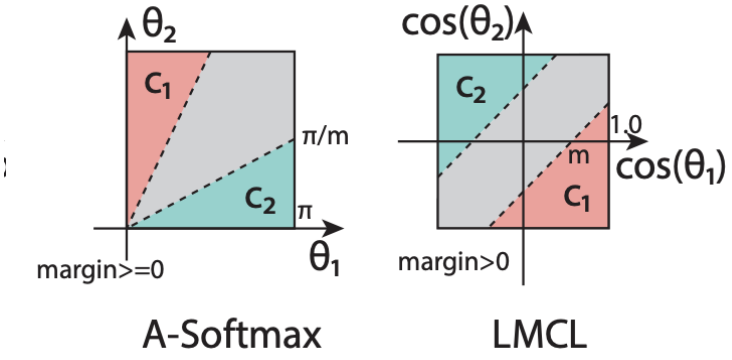

Figure 7: The comparison of decision margins between A-Softmax and Large Margin Consine Loss, for the binary-classes scenarios. Dashed line represents decision boundary, and gray areas are decision margins. Figure borrowed from [8]

D.4 Visualization with tSNE

In this last part, we will use the tSNE again to visualize the feature vectors of the 10 same identities' images. This time, those images will be fed forward to the best CosFace net we trained. The results are shown in Figure 8. We can see that the feature vector clustered more obviously than the previous SphereFace net results.

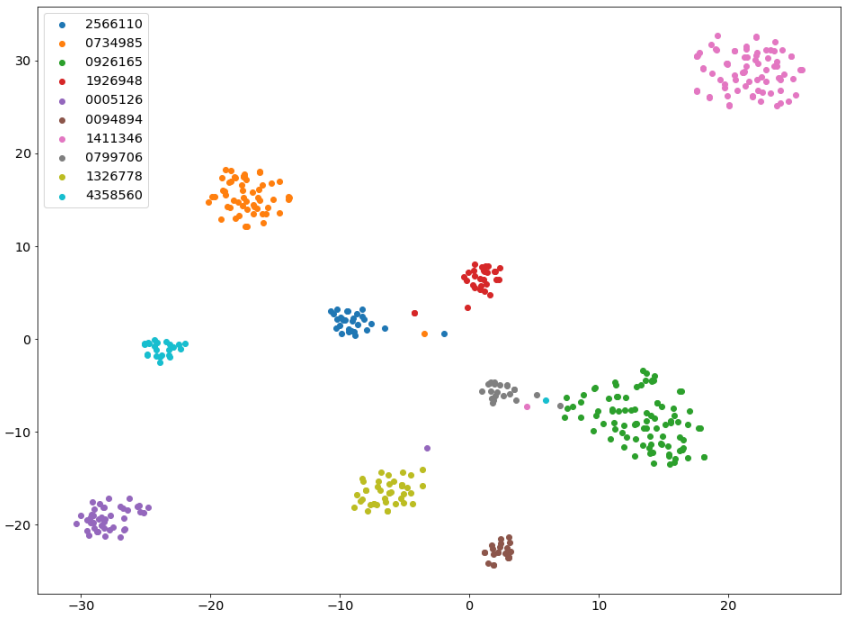

Figure 8: tSNE visualization of 10 same identities chosen in the prevoius section, whose images are fed forward through the trained CosFace net.

References

[1] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[2] Weiyang Liu, Yandong Wen, Zhiding Yu, Ming Li, Bhiksha Raj, and Le Song. Sphereface: Deep hypersphere embedding for face recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 212–220, 2017.

[3] Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li, and Yu Qiao. Joint face detection and alignment using multitask cascaded convolutional networks. IEEE Signal Processing Letters, 23(10):1499–1503, 2016.

[4] Dong Yi, Zhen Lei, Shengcai Liao, and Stan Z Li. Learning face representation from scratch. arXiv preprint arXiv:1411.7923, 2014.

[5] https://github.com/clcarwin/sphereface_pytorch

[6] Weiyang Liu, Yandong Wen, Zhiding Yu, and Meng Yang. Large-Margin Softmax Loss for Convolutional Neural Networks. In ICML, vol. 2, no. 3, p. 7. 2016.

[7] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).

[8] Hao Wang, Yitong Wang, Zheng Zhou, Xing Ji, Dihong Gong, Jingchao Zhou, Zhifeng Li, and Wei Liu. Cosface: Large margin cosine loss for deep face recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 5265–5274, 2018.

[9] https://github.com/MuggleWang/CosFace_pytorch