HW 6 Contrastive learning

This assignment aims to have you learn how to modify loss functions and simultaneously process multiple feature vectors in both face verification and face recognition tasks by using the CelebA dataset as an example.

CelebFaces Attributes Dataset (CelebA) is a large-scale face attributes dataset with more than 200K celebrity images, each with 40 attribute annotations, covering a wide range of pose variations and background clutter. The dataset is pruned so that the training time is appropriate for the assignment.

This homework is divided into three parts:

1. Face verification with contrastive loss
2. Face verification with triplet loss
3. Face verification with InfoNCE loss
4. Face verification evaluation

Import main libraries

In [ ]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os
import os.path as osp
from collections import defaultdict
from PIL import Image
import math
from tqdm.notebook import tqdm
import random

import torch
from torch import nn
import torch.nn.functional as F
from torch import optim
import torchvision
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
```

In [ ]:
```
!pwd
```

```
/kaggle/working
```

Download and unzip the dataset

In [ ]:
```
!pip install gdown
```

In [ ]:
```
!gdown --id 1_2oATmA0Jw61qs7I7kAG9Q2LHNZSQX7A
```

```
/opt/conda/lib/python3.10/site-packages/gdown/__main__.py:132: FutureWarning: Option `--i
d` was deprecated in version 4.3.1 and will be removed in 5.0. You don't need to pass it
anymore to use a file ID.
  warnings.warn(
Downloading...
From: https://drive.google.com/uc?id=1_2oATmA0Jw61qs7I7kAG9Q2LHNZSQX7A
To: /kaggle/working/large_prepared_data.zip
100%|██████████████████████████████████████| 23.4M/23.4M [00:00<00:00, 34.6MB/s]
```

In [ ]:
```
!unzip large_prepared_data.zip
```

Common dataset

In [ ]:
```
class FaceDataset(Dataset):
  def __init__(self, root_dir, transform=None):
    self.root_dir = root_dir
    self.transform = transform
    self.label_df = pd.read_csv(f'{self.root_dir}/label_df.csv')

  def __len__(self):
    return len(self.label_df)

  def __getitem__(self, idx):
    data = self.label_df.iloc[idx]
    img = Image.open(f"{self.root_dir}/{data['filename']}").convert('RGB')
    if self.transform is not None:
        transformed_img = self.transform(img)
    else:
        transformed_img = None
    identity = data['identity']
    return transformed_img, identity, np.array(img)
```

This is an example to display a face image with its identity

In [ ]:
```
import matplotlib.pyplot as plt
face_dataset = FaceDataset(root_dir='large_prepared_data/test')

for idx in range(4):
  _, identity, img = face_dataset[idx]
  plt.title(f'identity = {identity}')
  plt.imshow(img)
  plt.show()
```

Part 1: Face verification with contrastive loss

The objective of the face verification task is to validate whether the face image x has the identity y by comparing it to the face database of the claimed identity. If the face similarity score between x and the face in the database of y is above a certain threshold, the image is then verified; otherwise, the identity is rejected.

In HW3, you have learned to calculate a similarity score based on compact image representation using a PCA / Fisher projection. Therefore, in this part, you will instead implement a more contemporary method by training the NN to propose a compact representation (feature vector) by using a Siamese network and contrastive loss (Chopra et al., 2005, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1467314).

In contrast to the classification task that forces the model to learn all possible classes of objects, the general idea of object verification is based on contrastive learning, a framework that teaches the model to distinguish the two objects from each other. For a contrastive loss, the model will receive a pair of image and then learn to recognize whether both of them has the same identity by encouraging the feature vector of the same identity to come closer and different one to move away from each other.

1.1 (TODO) Dataset and DataLoader

First, we will start by writing a dataloader. The contrastive loss learns whether a pair of images have the same identity. Therefore, in this subsection, you have to write a dataloader that returns the tuple `(img1, img2, is_same_identity (bool))`.

File structure of this dataset

```
- large_prepared_data/
└──train/
|     - {img1}.jpg
|     - {img2}.jpg
|     - label_df.csv
└──val/
|     - {img3}.jpg
|     - {img4}.jpg
|     - label_df.csv
└──test/
    - {img5}.jpg
    - {img6}.jpg
    - label_df.csv
```

Note

label_df.csv has 2 columns that are the filename such as 001257.jpg and the identity such as 1691

Instructions

TODO 1-4: Fill in the missing code in the cells below. \ TODO 1-2: Organize the dataset for simple data access. \ TODO 3: Randomly select the datapoint from the dataset and format it to be ready for training. \ TODO 4: Initialize the dataloader.

In [ ]:
```
class SiameseDataset(Dataset):
    def __init__(self, root_dir, transform=None):
      '''
        The dataset of siamese network
        [Args]
        - root_dir = root directory of the dataset
        - transform = transformations for images
      '''
      np.random.seed(123)
      random.seed(123)
      self.root_dir = root_dir
      self.transform = transform
      label_df = pd.read_csv(f'{root_dir}/label_df.csv')
      self.num_images = len(label_df)
      label_df = label_df.groupby('identity')['filename'].apply(list).reset_index().renam
      self.load_images_to_memory_(label_df)

    def load_images_to_memory_(self, label_df):
      '''
        Load all images into memory
        [Args]
        - label_df = The dataframe containing the identities and the filenames of images
      '''
      # TODO 1: load images to `self.data` according to the below structure
      # and `self.images`, `self.identities` following by idx
      # Note: identity{i}: str, image{i}: PIL.Image (convert them to RGB as well)
      # e.g. self.data = {
      #   'identity1': [image1, image2],
      #   'identity2': [image3, image4, image5],
      # }
      #
      # identity{i}: str, image{i}: PIL.Image
      self.data = dict()
      for iden in label_df['identity']:
        self.data[str(iden)] = []
        for filename in label_df.loc[label_df['identity'] == iden, 'filenames'].values[0]
          img = Image.open(f"{self.root_dir}/{filename}").convert('RGB')
          if self.transform is not None:
            img = self.transform(img)
          self.data[str(iden)].append(img)
        self.data[str(iden)] = self.data[str(iden)]
      self.images = 0
      self.identities = 0

      # TODO 2: keep all unique identities as list with `self.unique_identities`
```

```
      # in `self.unique_identities` as a numpy array.
      self.unique_identities = label_df['identity'].unique().tolist()

    def __len__(self):
      return self.num_images

    def __getitem__(self, idx):
      '''
      Return a pair of image together with its label
      [Args]
      - idx: int
      [Return]
      - img1: torch.FloatTensor
      - img2: torch.FloatTensor
      - label: totch.FloatTensor = 1 (same class), 0 (different class)
      '''
      # TODO 3: randomly sample a pair of images
      # Note: idx is even, it should return the same class pair and otherwise
      # Please use label = 1 for the same class pair
      # and label = 0 for the different class pair
      if idx % 2 == 0:
        label = 1
        identity = random.choice(self.unique_identities)
        img1, img2 = random.sample(self.data[str(identity)], 2)
      else:
        label = 0
        identity1, identity2 = random.sample(self.unique_identities, 2)
        img1 = random.choice(self.data[str(identity1)])
        img2 = random.choice(self.data[str(identity2)])

      return img1, img2, torch.from_numpy(np.array([label], dtype=np.float32))
```

In [ ]:

```
img_size = 224
train_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.CenterCrop(img_size),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5319, 0.4399, 0.3929],
                         std=[0.3076, 0.2898, 0.2907])
])

val_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.CenterCrop(img_size),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5319, 0.4399, 0.3929],
                         std=[0.3076, 0.2898, 0.2907])
])

train_batch_size = 16
val_batch_size = 16
test_batch_size = 16


# TODO 4: declare the datasets and the dataloaders
train_siamese_dataset = SiameseDataset(root_dir='large_prepared_data/train', transform=tr
train_siamese_dataloader = DataLoader(train_siamese_dataset, batch_size=train_batch_size,

val_siamese_dataset = SiameseDataset(root_dir='large_prepared_data/val', transform=val_tr
val_siamese_dataloader = DataLoader(val_siamese_dataset, batch_size=val_batch_size, shuff

test_siamese_dataset = SiameseDataset(root_dir='large_prepared_data/test', transform=val_
test_siamese_dataloader = DataLoader(test_siamese_dataset, batch_size=test_batch_size, sh
```

1.2 (TODO) Siamese network

After the dataloader is initialized, we then build a siamese network. Section 1.5 will explain how a siamese network works in full detail.

Siamese network is a typical CNN that consists of three modules:

1. A feature extractor (ResNet18) for extracting the feature map from an image.
2. A global pooling for reducing the image dimension.
3. A fully connected layer for compressing the feature vector

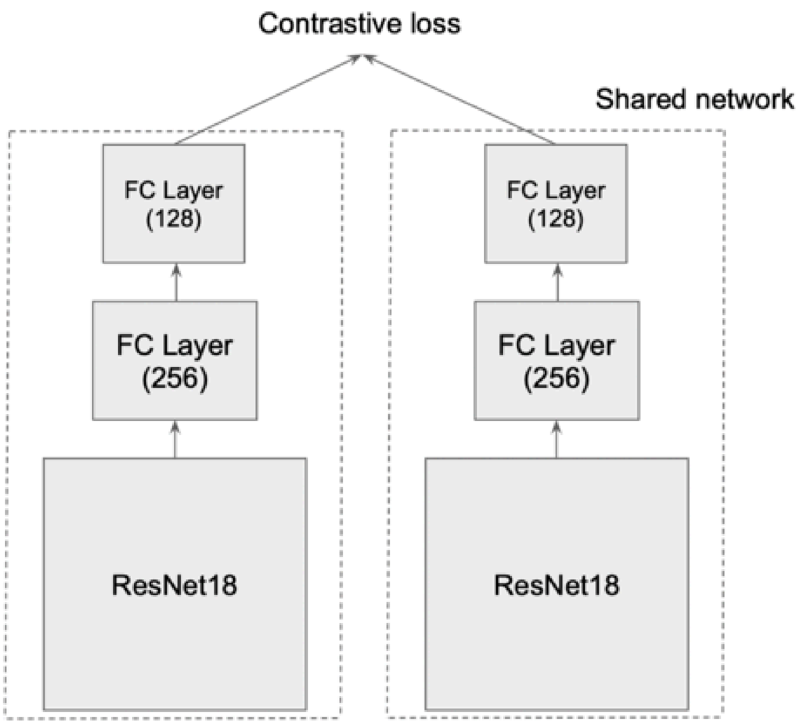
Every fully connected layer is followed by ReLU activations.

TODO 5: Implement a siamese network based on the description.

Note

1. You can use the ResNet18 from the `torchvision` library (How to use torchvision: https://pytorch.org/vision/stable/models.html)
2. We will not use pretrained weights.

Figure 1 Siamese network



In [ ]:
```
class SiameseNetwork(nn.Module):
  # TODO 5: implement the siamese network
  def __init__(self):
      super().__init__()
      self.resnet = torchvision.models.resnet18()
      self.resnet = nn.Sequential(*list(self.resnet.children())[:-1])
      self.global_pooling = nn.AdaptiveAvgPool2d((1, 1))
      self.fc = nn.Sequential(
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
      )

  def extract_feature(self, x):
      output = self.resnet(x)
      output = self.global_pooling(output)
      output = output.view(output.size(0), -1)
      output = self.fc(output)
      return output

  def forward(self, input1, input2):
      output1 = self.extract_feature(input1)
      output2 = self.extract_feature(input2)
      return output1, output2
```

1.3 (TODO) Constrastive loss

A contrastive loss is a loss used to minimize the dissimilarity between two images by encouraging the feature vector of the same identity to come closer and different one further than a constant margin m to move away from each other.

The contrastive loss is mathematically defined as:

$$L\ (contrastive\ loss) = \begin{cases} d(r_1, r_2) & if\ identity(r1) = identity(r2) \\ max(0, m - d(r_1, r_2)) & if\ identity(r1) \neq identity(r2) \end{cases}$$

where

- $d(r_1, r_2)$ = euclidean distance between r_1 and r_2
- m = margin
- $identity(x)$ = the identity of x
- r_1 = the feature vector of the first image
- r_2 = the feature vector of the second image

The term $d(r_1, r_2)$ is the distance between the two feature vectors. The contrastive loss minimizes the distance between the feature vectors of the same identity (positive pair) but maximizes the distance of the different identities (negative pair).

The margin m is used to prevent the loss from collapsing to a trivial solution. For instance, when $m = 0$, the model could achieve $L = 0$ just by exploiting the objective by setting $r_1 = r_2$, and the model, as a result, would learn nothing useful.

HINT

Many torch functions often have the same functionality as NumPy functions, even sharing the same function name. Therefore, if you are struggling with this part, you might write the loss using NumPy first and then convert it to the torch function (https://pytorch.org/docs/stable/torch.html) later. **Avoid using "if statements" to make the training faster. You have already learned many tricks that convert if statements into a single equation.**

TODO 6: Implement a contrastive loss based on the description above.

In [ ]:
```
import torch
from torch import nn
import torch.nn.functional as F

class ContrastiveLoss(torch.nn.Module):
    def __init__(self, margin):
        super(ContrastiveLoss, self).__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(output1, output2)
        loss_contrastive = torch.mean((1-label) * torch.pow(euclidean_distance, 1) +
                                      (label) * torch.pow(torch.clamp(self.margin - eucli
        return loss_contrastive
```

1.4 Initiazing the model, criterion, optimizer and scheduler

In [ ]:
```
siamese_margin = 2
learning_rate = 1e-4

siamese_model = SiameseNetwork()
siamese_criterion = ContrastiveLoss(margin=siamese_margin)
siamese_optimizer = optim.Adam(siamese_model.parameters(), lr=learning_rate)
siamese_scheduler = optim.lr_scheduler.ReduceLROnPlateau(siamese_optimizer, 'min', patien
```

1.5 (TODO) Training loop

The training procedure of a siamese network consists of the following steps:

1. Forward pass the `img1`.
2. Forward pass the `img2`.
3. Calculate the distance between the feature vector of `img1` and `img2` $(d(r_1, r_2))$.
4. Use the distance in step 3 as a loss and update the model.
5. Repeat step 1-4 until satisfied.

As you would notice, both first and second step shares the same network weights. Therefore, the word "Siamese" in the siamese network originates from the "Siamese twins" since the network performs two forward passes to compare whether the feature vectors have the same identity by using the same set of network parameters (shared parameters).

TODO 7-8: Feed pairs of images to the network, compute contrastive loss to measure the dissimilarity of pairs of face images and update the network.

TODO 9: Feed a pair of images from validation set to the network and compute the validation loss.

Trivia

The Siamese twin is a conjoined twin brother born in Siam who later move to the US (https://th.wikipedia.org/wiki/%E0%B8%AD%E0%B8%B4%E0%B8%99-%E0%B8%88%E0%B8%B1%E0%B8%99).

In [ ]:
```
num_epochs = 20
device = "cuda" if torch.cuda.is_available() else "cpu"
siamese_model.to(device)

os.makedirs('weights', exist_ok=True)
best_weights_path = 'weights/best_siamese_weights.pth'

train_losses = []
val_losses = []
min_val_loss = float('inf')

for epoch in tqdm(range(num_epochs)):
    siamese_model.train()
    total_train_loss = 0


    for img1, img2, label in tqdm(train_siamese_dataloader):

        # TODO 7: feed data to model and compute loss
        out1, out2 = siamese_model(img1.to(device), img2.to(device))
        train_loss = siamese_criterion(out1, out2, label.to(device))

        # TODO 8: back propagate
        siamese_optimizer.zero_grad()
        train_loss.backward()
        siamese_optimizer.step()

        total_train_loss += train_loss.item()


    current_train_loss = total_train_loss / len(train_siamese_dataloader)
    train_losses.append(current_train_loss)

    total_val_loss = 0
    siamese_model.eval()
    for val_img1, val_img2, val_label in val_siamese_dataloader:

        # TODO 9: feed data to model and compute loss
        val_out1, val_out2 = siamese_model(val_img1.to(device), val_img2.to(device))
```

```
        val_loss = siamese_criterion(val_out1, val_out2, val_label.to(device))

        total_val_loss += val_loss.item()
    current_val_loss = total_val_loss / len(val_siamese_dataloader)
    val_losses.append(current_val_loss)
    if current_val_loss < min_val_loss:
        min_val_loss = current_val_loss
        torch.save(siamese_model.state_dict(), best_weights_path)
    print(f'Epoch {epoch+1} - Train loss = {current_train_loss:.4f} - Val loss = {current
    siamese_scheduler.step(current_val_loss)
```

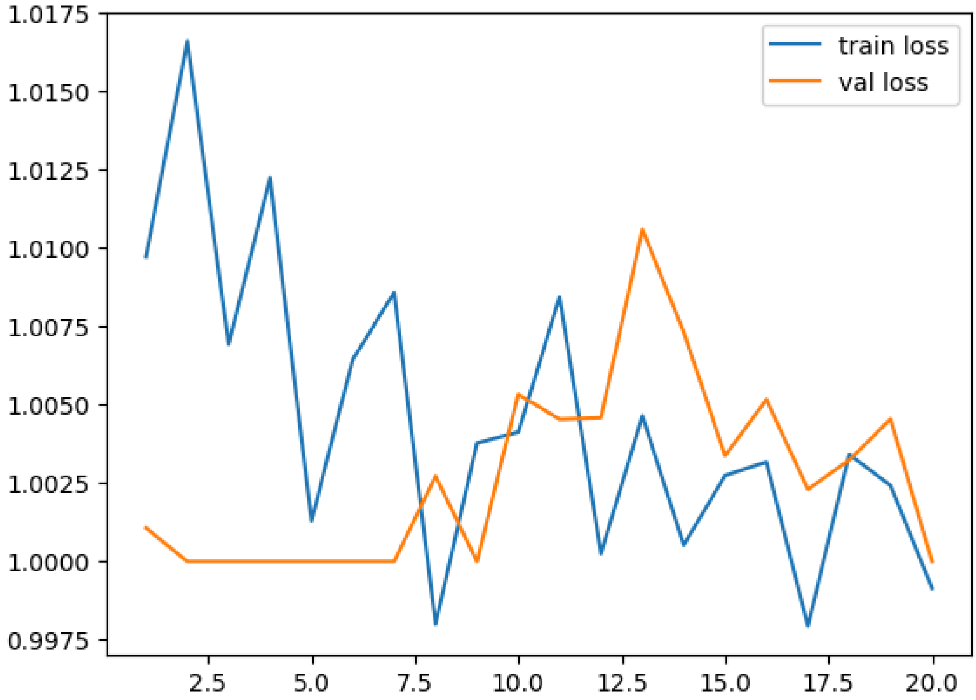
1.6 Visualization

This visualization displays pairs of images together with the distance between those pairs.

In [ ]:
```
# Showing images
def imshow(img, text=None):
    npimg = img.numpy()
    plt.axis("off")
    if text:
        plt.text(120, 8, text, style='italic',fontweight='bold',
            bbox={'facecolor':'white', 'alpha':0.8, 'pad':10})

    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


siamese_model.load_state_dict(torch.load(best_weights_path))
siamese_model.eval()

test_img1s, test_img2s, test_labels = next(iter(test_siamese_dataloader))
test_img1s, test_img2s, test_labels = test_img1s.to(device), test_img2s.to(device), test_
with torch.no_grad():
    test_out1s, test_out2s = siamese_model(test_img1s, test_img2s)

class UnNormalize(object):
    def __init__(self, mean, std):
        self.mean = mean
        self.std = std

    def __call__(self, tensor):
        """
        Args:
            tensor (Tensor): Tensor image of size (C, H, W) to be normalized.
        Returns:
            Tensor: Normalized image.
        """
        for t, m, s in zip(tensor, self.mean, self.std):
            t.mul_(s).add_(m)
            # The normalize code -> t.sub_(m).div_(s)
        return tensor

unnormalizer = UnNormalize(mean=[0.5319, 0.4399, 0.3929],
                          std=[0.3076, 0.2898, 0.2907])

for test_out1, test_out2, test_img1, test_img2, test_label in zip(test_out1s, test_out2s,
    test_img1 = unnormalizer(test_img1.detach().cpu().unsqueeze(0))
    test_img2 = unnormalizer(test_img2.detach().cpu().unsqueeze(0))
    concatenated = torch.cat((test_img1, test_img2), 0)
    distance = F.pairwise_distance(test_out1.unsqueeze(0), test_out2.unsqueeze(0))
    imshow(torchvision.utils.make_grid(concatenated), f'Label = {int(test_label[0])}, Dis
```

1.7 Plot loss history

In [ ]:
```
plt.plot(np.arange(1, len(train_losses)+1), train_losses, label='train loss')
plt.plot(np.arange(1, len(val_losses)+1), val_losses, label='val loss')
plt.legend()
plt.show()
```



1.8 (TODO) Plot t-SNE

After the training process is finished, we evaluate whether the learned representation is informative. In this task, embedding visualization is often performed to verify that the feature learned by the network is behaving as intended, i.e., feature vectors of the same identity should be close to each other and far away from other identities. Since the feature vector dimension is too high for a human to interpret, therefore, in this assignment, we use the t-SNE dimensionality reduction technique to compress the feature into a 2D space.

Instructions

TODO 10: Extract the feature vectors of the test set and store them as

`embeddings` : torch.FloatTensor = feature vectors of all images in the test set

`identities` : list or torch.Tensor or np.array = identities of all images in the test set

Hint
Use `FaceDataset` that is imported at `Common Dataset` section

WARNING!! Don't forget load its best weights and change to eval mode first

In [ ]:
```
test_batch_size = 32

# TODO 10: Extract the feature vectors of the test set and store them as
# `embeddings`: torch.FloatTensor = feature vectors of all images in the test set
# `identities`: list or torch.Tensor or np.array = identities of all images in the test s
# Hint      => Use `FaceDataset` that is imported at `Common Dataset` section
# WARNING!! => Don't forget load its best weights and change to eval mode first

siamese_model.load_state_dict(torch.load(best_weights_path))
siamese_model.eval()

embeddings = []
identities = []

for img, identity, _ in tqdm(FaceDataset(root_dir='large_prepared_data/test', transform=v
    img = img.unsqueeze(0)
    with torch.no_grad():
        embedding = siamese_model.extract_feature(img.to(device))
    embeddings.append(embedding.cpu().numpy())
```

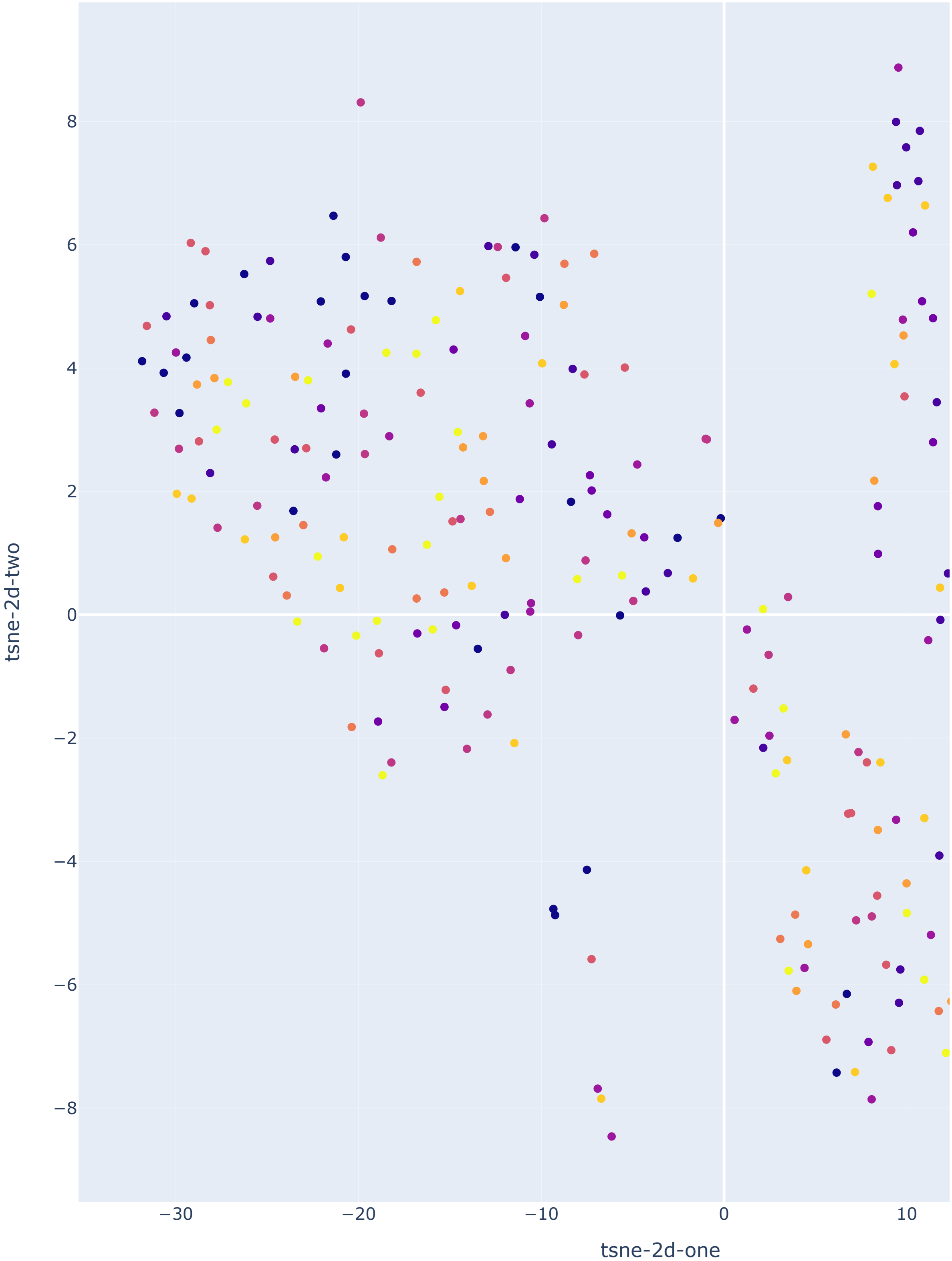
```
    identities.append(identity)

embeddings = np.array(embeddings).squeeze()
identities = np.array(identities)
```

```
  0%|          | 0/301 [00:00<?, ?it/s]
```

In [ ]:
```
import time
from sklearn.manifold import TSNE

time_start = time.time()
tsne = TSNE(n_components=2, verbose=1, perplexity=30, n_iter=3000)
tsne_result = tsne.fit_transform(embeddings, identities)
print('t-SNE done! Time elapsed: {} seconds'.format(time.time()-time_start))
```

```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 301 samples in 0.000s...
[t-SNE] Computed neighbors for 301 samples in 0.098s...
[t-SNE] Computed conditional probabilities for sample 301 / 301
[t-SNE] Mean sigma: 0.123739
[t-SNE] KL divergence after 250 iterations with early exaggeration: 50.957466
[t-SNE] KL divergence after 1650 iterations: 0.384311
t-SNE done! Time elapsed: 2.159878969192505 seconds
```

In [ ]:
```
import plotly.express as px

# relabel to be easier to see in t-SNE visualization
label = []
idx = 0
id2label = dict()
for identity in identities:
  identity = int(identity)
  if identity not in id2label:
    id2label[identity] = idx
    idx += 1
  label.append(id2label[identity])

df_subset = pd.DataFrame({'label': label})

df_subset['tsne-2d-one'] = tsne_result[:,0]
df_subset['tsne-2d-two'] = tsne_result[:,1]

fig = px.scatter(df_subset, x="tsne-2d-one", y="tsne-2d-two", color="label", height=1000,
fig.show()
```



1.9 (TODO) Analyzing feature vector visualization result

TODO 11: What could you say about the displayed visualization? Is the model working as expected?

Answer here: From the visualization, Some group of points are contains mostly same labels but it's still blurly.

Part 2: Face verification with triplet loss + center loss

In contrast to contrastive loss which learns to distinguish whether the two images have the same identity, triplet loss is proposed as an alternative by introducing an anchor image as a third input. Triplet loss receives three inputs: anchor, positive pair, and negative pair. The positive pair is an image having the same identity as the anchor while the negative pair is the one with a different identity. The loss learns to minimize the distance between the anchor and positive pair, and maximize the distance between the anchor and the negative pair. Compared to contrastive loss, triplet loss offers more training stability and better model performance.

Nevertheless, both contrastive and triplet loss also has some shortcomings as these losses only optimize on a pairwise level. This might result in feature vectors of the same identity taking the form of multiple small clusters scattering across the feature space since there is no explicit loss to bind them into a single group. Therefore, a center loss is proposed to mitigate this problem by encouraging the intra-class (same identity) feature vectors to come closer to their intra-class centroids.

In this part, you are going to implement a face verification network by jointly training **three losses**: triplet, center, and cross-entropy loss.

2.1 Dataset and DataLoader

In this section, you are going to implement a dataloader for the combined loss. The dataloader should return the tuple `(anchor_img, pos_img, neg_img, anchor_label, pos_label, neg_label)`.

- The positive image must have the same identity as the anchor image.
- The negative image must have a different identity from the anchor image.

In [ ]:
```
class TripletDataset(Dataset):
    def __init__(self, root_dir, transform=None):
      np.random.seed(123)
      random.seed(123)
      self.root_dir = root_dir
      self.transform = transform

      label_df = pd.read_csv(f'{root_dir}/label_df.csv')
      label_df = label_df.groupby('identity')['filename'].apply(list).reset_index().renam
      self.images = []
      self.labels = []
      self.label2indices = dict()
      self.load_images_to_memory_(label_df)

    def load_images_to_memory_(self, label_df):
      # load images and labels into memory
      # We have to relabel from identities to 0,1,2,...,num_classes-1
      # Relevant variables
      # 1. `self.images` = PIL images  (Also convert it to RGB)
      # 2. `self.label2indices` = the dictionary storing a label as a key and the indices
      # 3. `self.labels` = labels of images (relabeled)
      # Note: the index of images is from iterating over label_df
      self.images = []
      self.label2indices = defaultdict(list)
      self.labels = []
      img_idx = 0
      label_idx = 0
      identity2label = dict()
      for idx in tqdm(range(len(label_df))):
          row = label_df.iloc[idx]
          identity, filenames = row['identity'], row['filenames']

          if identity not in identity2label:
            identity2label[identity] = label_idx
            label_idx += 1
```

```
            label = identity2label[identity]

            for filename in filenames:
                self.images.append(Image.open(f'{self.root_dir}/{filename}').convert('RGB')
                self.labels.append(label)
                self.label2indices[label].append(img_idx)
                img_idx += 1


    def __len__(self):
      return len(self.images)

    def __getitem__(self, idx):
      # generate an anchor image, a positive image and a negative image together with anc
      # anchor image is the image according to idx
      # positive image is the image that has the same identity with the anchor image
      # negative image is the image that has the different identity with the anchor image
      anchor_img = self.images[idx]
      anchor_label = self.labels[idx]

      pos_idx = random.choice(self.label2indices[anchor_label])
      while pos_idx == idx:
        pos_idx = random.choice(self.label2indices[anchor_label])
      pos_img = self.images[pos_idx]
      pos_label = anchor_label
      neg_label = random.choice(list(set(self.labels) - {anchor_label}))
      neg_img = self.images[random.choice(self.label2indices[neg_label])]


      # utilize `self.transform' to convert images to tensors
      if self.transform is not None:
          anchor_img = self.transform(anchor_img)
          pos_img = self.transform(pos_img)
          neg_img = self.transform(neg_img)


      return anchor_img, pos_img, neg_img, anchor_label, pos_label, neg_label
```

In [ ]:

```
image_size = 224
train_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.CenterCrop(image_size),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5319, 0.4399, 0.3929],
                         std=[0.3076, 0.2898, 0.2907])
])

val_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.CenterCrop(image_size),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5319, 0.4399, 0.3929],
                         std=[0.3076, 0.2898, 0.2907])
])

train_batch_size = 16
val_batch_size = 16
test_batch_size = 16

# declare the datasets and the dataloaders

train_triplet_dataset = TripletDataset(root_dir='large_prepared_data/train', transform=tr
train_triplet_dataloader = DataLoader(train_triplet_dataset, batch_size=train_batch_size,

val_triplet_dataset = TripletDataset(root_dir='large_prepared_data/val', transform=val_tr
val_triplet_dataloader = DataLoader(val_triplet_dataset, batch_size=val_batch_size, shuff

test_triplet_dataset = TripletDataset(root_dir='large_prepared_data/test', transform=val_
test_triplet_dataloader = DataLoader(test_triplet_dataset, batch_size=test_batch_size, sh
```

```
  0%|          | 0/80 [00:00<?, ?it/s]
  0%|          | 0/10 [00:00<?, ?it/s]
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
```

2.2 Triplet Network

This network is a simpliflied version of strong baseline person re-identification task (ref: https://arxiv.org/abs/1903.07071)

Triplet network consists of 3 modules

1. Feature extractor (ResNet18) is for mapping an image to a feature map
2. Global pooling (Global average pooling or use flatten instead) is for converting a feature map to a feature vector
3. Bottleneck (Batch normalization only scale not shift) is for the consistency of training between triplet loss and crossentropy because it is difficult to optimize those losses in embedding space at the same time
4. Linear classifier is to classify who this face is

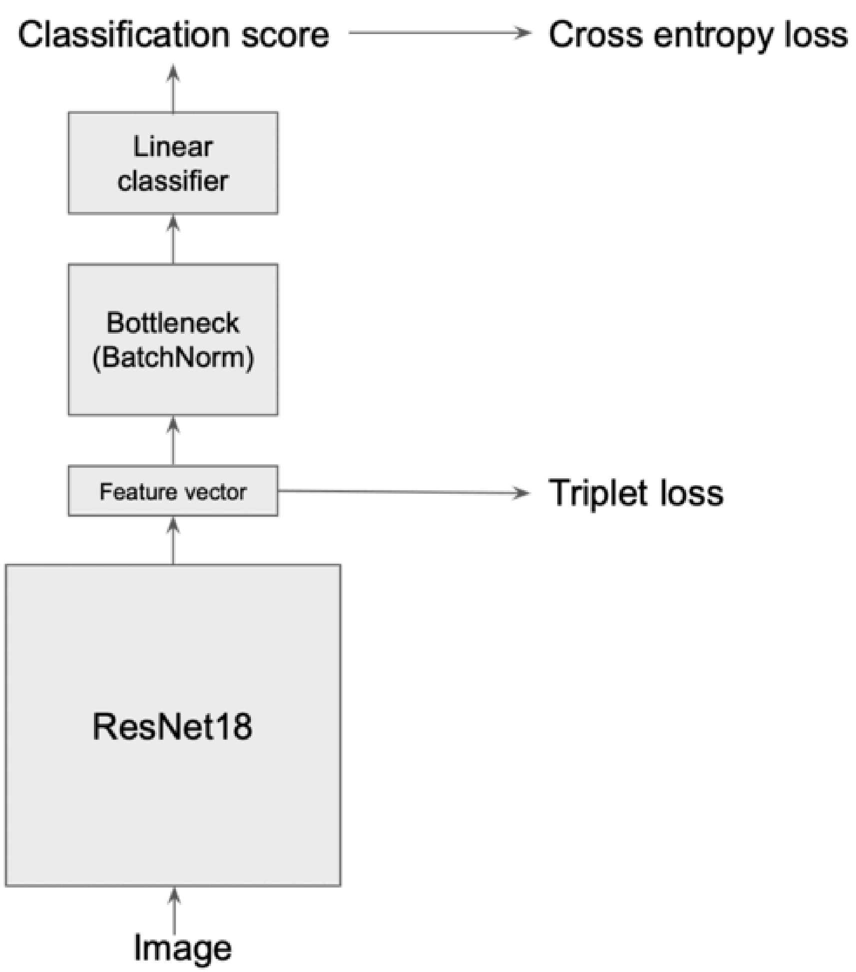
Note

1. ResNet18 can call via `torchvision` library (How to use torchvision: https://pytorch.org/vision/stable/models.html)
2. ResNet18 we use will not load pretrained weights
3. We will use ResNet18 only extracting feature maps.

Hint!

1. For global average pooling, read the documentation of `nn.AdaptiveAvgPool2d`
2. For bottleneck, try `requires_grad_` method to stop calculating gradient at batch normalization

Figure 3 Triplet network



In [ ]:
```
def weights_init_kaiming(m):
    classname = m.__class__.__name__
    if m.affine:
        nn.init.constant_(m.weight, 1.0)
        nn.init.constant_(m.bias, 0.0)

def weights_init_classifier(m):
    classname = m.__class__.__name__
    if classname.find('Linear') != -1:
        nn.init.normal_(m.weight, std=0.001)
        if m.bias:
            nn.init.constant_(m.bias, 0.0)

class TripletNetwork(nn.Module):
  # implement the triplet network
  def __init__(self, num_classes):
    super(TripletNetwork, self).__init__()
    resnet = torchvision.models.resnet18(pretrained=False)
    self.conv = torch.nn.Sequential(*(list(resnet.children())[:-1]))
    self.global_pool = nn.AdaptiveAvgPool2d(1)

    self.bottleneck = nn.BatchNorm1d(512)
    # no shift
    self.bottleneck.bias.requires_grad_(False)

    self.classifier = nn.Linear(512, num_classes, bias=False)

    self.bottleneck.apply(weights_init_kaiming)
    self.classifier.apply(weights_init_classifier)

  def extract_feature(self, x):
    x = self.conv(x)

    global_feat = self.global_pool(x)
    global_feat = global_feat.view(global_feat.size(0), -1)

    feat = self.bottleneck(global_feat)

    if not self.training:
      return feat

    cls_score = self.classifier(feat)

    return global_feat, cls_score

  def forward(self, anchor_img, pos_img, neg_img):
    anchor_feat, anchor_score = self.extract_feature(anchor_img)
    pos_feat, pos_score = self.extract_feature(pos_img)
    neg_feat, neg_score = self.extract_feature(neg_img)

    feats = torch.cat((anchor_feat, pos_feat, neg_feat))
    scores = torch.cat((anchor_score, pos_score, neg_score))
    return anchor_feat, pos_feat, neg_feat, feats, scores
```

2.3 Triplet loss

Triplet loss uses the same concept as contrastive loss that is the anchor image will pull the positive one closer and push the negative one further than the constant margin away. Hence, we have sampled three images to compute it. The three images consist of:

1. an anchor image = an initial image
2. a positive image = the image having the same identity as the anchor image
3. a negative image = the image having a different identity from the anchor image

The triplet loss is mathematically defined as:

$$L_{Triplet} = max(0, m + d(r_a, r_p) - d(r_a, r_n))$$

where

- $d(r_1, r_2)$ = euclidean distance between r_1 and r_2
- m = margin
- r_a = the feature vector of the anchor image
- r_p = the feature vector of the positive image
- r_n = the feature vector of the negative image

The loss aims to minimize $d(r_a, r_p)$ while maximizing $d(r_a, r_n)$ at the same time.

The margin m is used to prevent the loss from collapsing to a trivial solution. For instance, when $m = 0$, the model could achieve $L = 0$ just by exploiting the objective by setting $r_a = r_p = r_n$, and the model, as a result, would learn nothing useful.

In [ ]:
```
class TripletLoss(nn.Module):
  # implement the triplet loss
  def __init__(self, margin=2.0):
    super(TripletLoss, self).__init__()
    self.margin = margin
    self.euclidean_dist_fn = nn.PairwiseDistance(p=2)

  def forward(self, anchor_img, pos_img, neg_img):
    pos_dist = self.euclidean_dist_fn(anchor_img, pos_img)
    neg_dist = self.euclidean_dist_fn(anchor_img, neg_img)
    return torch.mean(torch.relu(pos_dist - neg_dist + self.margin))
```

2.4 Center loss

Reference: https://ydwen.github.io/papers/WenECCV16.pdf

Definition

Centroid: a representative of each class. There are several ways to select a representative, one way to do this is to take an average on those embeddings.

Concept

Center loss enforces the intra-class feature vectors to come closer to the centroid of their class so that all feature vectors of the same identity are clustered around a single centroid. Since the model is updated after each iteration, the centroids should also be updated accordingly. However, re-calculating the centroids by averaging the feature vectors for each class every iteration is computational-extensive on a large scale. Therefore, the centroid is instead learned from the representative of each class from the sampled data.

Implementation detail

In `__init__` method

1. For simplicity, we will store the centroids in the class `CenterLoss`. Therefore, you have to initialize the centroids as a random tensor with the size of `(num_classes, feature_dimension)`. The tensor has to be set with `nn.Paramater` so that the gradient could be calculated (ref: https://pytorch.org/docs/stable/generated/torch.nn.parameter.Parameter.html).

In the `forward` method

1. Calculate the distance between the feature vector and its center with a squared Euclidean distance
2. Clip the value in each element to be not greater than 1e+12, not lower than 1e-12, and sum them
3. Normalize the loss with its batch size

Center loss equation

$L_{Center} = \frac{1}{B}\sum_{j=1}^{B}||f_j - c_{y_j}||_2^2$

where

- f_j = a feature vector before fed into the bottleneck at index j
- c_{y_j} = a center of the class corresponded to the index j
- B = batch size

The center loss minimized the distance between the feature vector and the centroid of the corresponded identity.

Note

1. At `__init__` method in step 1. Do not forget to transfer the parameters to GPU with `.cuda()` otherwise, it will not utilize GPU on this part.
2. Clip the magnitude of the loss at the final stage before updating both upper bound and lower bound to avoid vanishing / exploding gradients.

Hint

1. In step 2 of the `forward` method, you should use `torch.clamp` (ref: https://pytorch.org/docs/stable/generated/torch.clamp.html) to clip the lower bound and the upper bound as you want.

In [ ]:
```
class CenterLoss(nn.Module):
  def __init__(self, num_classes, feat_dim):
    super(CenterLoss, self).__init__()
    self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
    self.num_classes = num_classes
    self.feat_dim = feat_dim
    self.centers = nn.Parameter(torch.randn(self.num_classes, self.feat_dim).to(self.devi

  def forward(self, x, labels):
    batch_size = x.size(0)
    # compute squared euclidean distance
    distmat = torch.pow(x, 2).sum(dim=1, keepdim=True).expand(batch_size, self.num_classe
      torch.pow(self.centers, 2).sum(dim=1, keepdim=True).expand(self.num_classes, batch_
    distmat.addmm_(1, -2, x, self.centers.t())

    classes = torch.arange(self.num_classes).long().to(self.device)
    labels = labels.unsqueeze(1).expand(batch_size, self.num_classes)
    mask = labels.eq(classes.expand(batch_size, self.num_classes))

    dist = distmat * mask.float()
    loss = dist.clamp(min=1e-12, max=1e+12).sum() / batch_size
    return loss
```

In [ ]:
```
center_loss_fn = CenterLoss(80, 512)
feats = torch.randn(32,512).to(device)
labels = torch.randn(32).to(device)
center_loss_fn(feats, labels)
```

```
/tmp/ipykernel_34/782439988.py:14: UserWarning:

This overload of addmm_ is deprecated:
        addmm_(Number beta, Number alpha, Tensor mat1, Tensor mat2)
Consider using one of the following signatures instead:
        addmm_(Tensor mat1, Tensor mat2, *, Number beta, Number alpha) (Triggered interna
lly at /usr/local/src/pytorch/torch/csrc/utils/python_arg_parser.cpp:1519.)
```

Out[ ]:
```
tensor(8.0000e-11, device='cuda:0', grad_fn=<DivBackward0>)
```

2.5 Declare model, criterions, optimizers, hyparameters and scheduler

In [ ]:
```
# for triplet loss
triplet_margin = 3.0

# get the number of classes to construct a linear classifier and the parameters in center
num_classes = len(set(train_triplet_dataset.labels))
print(f'num_classes = {num_classes}')

# declare the triplet model and the triplet criterion
triplet_model = TripletNetwork(num_classes)
triplet_criterion = TripletLoss(margin=triplet_margin)

triplet_optimizer = optim.Adam(triplet_model.parameters(), lr=5e-4)
triplet_scheduler = optim.lr_scheduler.ReduceLROnPlateau(triplet_optimizer, 'min', patien

# declare cross entropy loss
crossentropy_criterion = nn.CrossEntropyLoss()

# For center loss
# declare the center criterion
triplet_center_criterion = CenterLoss(num_classes, 512)

triplet_center_optimizer = optim.Adam(triplet_center_criterion.parameters(), lr=0.5)
triplet_center_loss_weight = 5e-2
```

```
/opt/conda/lib/python3.10/site-packages/torchvision/models/_utils.py:208: UserWarning:

The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, ple
ase use 'weights' instead.

/opt/conda/lib/python3.10/site-packages/torchvision/models/_utils.py:223: UserWarning:

Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and
may be removed in the future. The current behavior is equivalent to passing `weights=None
`.
```

```
num_classes = 80
```

2.6 Training loop

The figure below shows how the center loss is updated in the original paper. L_s L_c, W, θ_c and c_j stand for classification loss, center loss, classification head's weight, CNN weight, and centroid of the class j, respectively. Line 6 shows how centroids are updated.

Algorithm 1. The discriminative feature learning algorithm

Input: Training data $\{\boldsymbol{x}_i\}$. Initialized parameters θ_C in convolution layers. Parameters W and $\{\boldsymbol{c}_j | j = 1, 2, ..., n\}$ in loss layers, respectively. Hyperparameter λ, α and learning rate μ^t. The number of iteration $t \leftarrow 0$.

Output: The parameters θ_C.

1: **while** not converge **do**
2: $t \leftarrow t + 1$.
3: Compute the joint loss by $\mathcal{L}^t = \mathcal{L}^t_S + \mathcal{L}^t_C$.
4: Compute the backpropagation error $\frac{\partial \mathcal{L}^t}{\partial \boldsymbol{x}^t_i}$ for each i by $\frac{\partial \mathcal{L}^t}{\partial \boldsymbol{x}^t_i} = \frac{\partial \mathcal{L}^t_S}{\partial \boldsymbol{x}^t_i} + \lambda \cdot \frac{\partial \mathcal{L}^t_C}{\partial \boldsymbol{x}^t_i}$.
5: Update the parameters W by $W^{t+1} = W^t - \mu^t \cdot \frac{\partial \mathcal{L}^t}{\partial W^t} = W^t - \mu^t \cdot \frac{\partial \mathcal{L}^t_S}{\partial W^t}$.
6: Update the parameters $\boldsymbol{c}_j$ for each j by $\boldsymbol{c}^{t+1}_j = \boldsymbol{c}^t_j - \alpha \cdot \Delta \boldsymbol{c}^t_j$.
7: Update the parameters θ_C by $\theta^{t+1}_C = \theta^t_C - \mu^t \sum^m_i \frac{\partial \mathcal{L}^t}{\partial \boldsymbol{x}^t_i} \cdot \frac{\partial \boldsymbol{x}^t_i}{\partial \theta^t_C}$.
8: **end while**

Joint loss equation

$$L_{Joint} = L_{ID} + L_{Triplet} + \beta L_{Center}$$

where

- L_{Joint} = joint loss
- L_{ID} = identity loss (cross entropy loss in this implementation)
- $L_{Triplet}$ = triplet loss
- L_{Center} = center loss
- β = center loss weights (affects only the triplet network, not center update)

The training procedure of the triplet network consists of the following steps:

1. Forward pass the `anchor_img`
2. Forward pass the `positive_img`
3. Forward pass the `negative_img`
4. Calculate the distance between the feature vectors of `anchor_img` , `positive_img` and `negative_img` and calculate a triplet loss ($L_{Triplet}$).
5. Calculate the classification loss (L_{ID}).
6. Calculate the center loss (L_{Center}).
7. Scale the center loss with the predetermined weight (β) (only used to update the triplet network not to update the centroids).
8. Sum the losses from steps 4-7 and update the triplet network. (Line 3-5)
9. Rescale the gradients with $\frac{1}{\beta}$ at the loss in step 7 and update the centroids. (Line 6-7)
10. Repeat steps 1-9 till converge

As you would notice, steps 1- 3 share the same model weight for feature extraction. Thus, the word "triplet" in triplet loss originates in a similar fashion as the "Siamese twins" but this loss function utilizes three feature vectors simultaneously instead of two.

Hint

1. At the gradient rescaling for the centroids, we should iterate parameters in center loss first with `.parameters()` and adjust the property of each parameter with `.grad.data`

In [ ]:

```
num_epochs = 20
device = "cuda" if torch.cuda.is_available() else "cpu"

os.makedirs('weights', exist_ok=True)
best_weights_path = 'weights/best_triplet_weights.pth'
triplet_model.to(device)
min_val_loss = float('inf')

train_losses = []
train_triplet_losses = []
train_crossentropy_losses = []
train_center_losses = []

val_losses = []
val_triplet_losses = []

for epoch in tqdm(range(num_epochs)):
    triplet_model.train()
    total_train_loss = 0
    total_train_triplet_loss = 0
    total_train_crossentropy_loss = 0
    total_train_center_loss = 0
    for anchor_img, pos_img, neg_img, anchor_label, pos_label, neg_label in tqdm(train_tr

        # feed data to the triplet model and compute triplet loss
        anchor_img, pos_img, neg_img = anchor_img.to(device), pos_img.to(device), neg_img
        anchor_label, pos_label, neg_label = anchor_label.to(device), pos_label.to(device
        labels = torch.cat((anchor_label, pos_label, neg_label))

        anchor_feat, pos_feat, neg_feat, feats, scores = triplet_model(anchor_img, pos_im
        train_triplet_loss = triplet_criterion(anchor_feat, pos_feat, neg_feat)

        # compute cross entropy loss
        train_crossentropy_loss = crossentropy_criterion(scores, labels)
```

```
        # compute center loss
        train_center_loss = triplet_center_loss_weight * triplet_center_criterion(feats,

        train_loss = train_triplet_loss + train_crossentropy_loss + train_center_loss
        total_train_loss += train_loss.item()
        total_train_triplet_loss += train_triplet_loss.item()
        total_train_crossentropy_loss += train_crossentropy_loss.item()
        total_train_center_loss += train_center_loss.item()

        # set zero gradients at two optimizers
        triplet_optimizer.zero_grad()
        triplet_center_optimizer.zero_grad()

        # back propagate at triplet network and step the main optimizer
        train_loss.backward()
        triplet_optimizer.step()

        # rescale gradients of centers because `triplet_center_loss_weight` should not af
        # and step the center optimizer
        for param in triplet_center_criterion.parameters():
          param.grad.data *= (1. / triplet_center_loss_weight)
        triplet_center_optimizer.step()

    current_train_loss = total_train_loss / len(train_triplet_dataloader)
    current_train_triplet_loss = total_train_triplet_loss / len(train_triplet_dataloader)
    current_train_crossentropy_loss = total_train_crossentropy_loss / len(train_triplet_d
    current_train_center_loss = total_train_center_loss / len(train_triplet_dataloader)
    train_losses.append(current_train_loss)
    train_triplet_losses.append(current_train_triplet_loss)
    train_crossentropy_losses.append(current_train_crossentropy_loss)
    train_center_losses.append(current_train_center_loss)

    total_val_loss = 0
    total_val_triplet_loss = 0
    triplet_model.eval()
    for val_anchor_img, val_pos_img, val_neg_img, _, _, _ in val_triplet_dataloader:

      # feed data to the triplet model and compute triplet loss
      val_anchor_img, val_pos_img, val_neg_img = val_anchor_img.to(device), val_pos_img.t
      with torch.no_grad():
          val_anchor_feat = triplet_model.extract_feature(val_anchor_img)
          val_pos_feat = triplet_model.extract_feature(val_pos_img)
          val_neg_feat = triplet_model.extract_feature(val_neg_img)
          val_triplet_loss = triplet_criterion(val_anchor_feat, val_pos_feat, val_neg_fea

      val_loss = val_triplet_loss
      total_val_loss += val_loss.item()
      total_val_triplet_loss += val_triplet_loss.item()
    current_val_loss = total_val_loss / len(val_triplet_dataloader)
    current_val_triplet_loss = total_val_triplet_loss / len(val_triplet_dataloader)
    val_losses.append(current_val_loss)
    val_triplet_losses.append(current_val_triplet_loss)

    if current_val_loss < min_val_loss:
        min_val_loss = current_val_loss
        torch.save(triplet_model.state_dict(), best_weights_path)
    print(f'Epoch {epoch+1} - Train loss = {current_train_loss:.4f} - Train triplet loss
    triplet_scheduler.step(current_val_loss)
```

2.7 Visualization

The visualization below displays an anchor, positive, and negative image and their respective distance.

In [ ]:

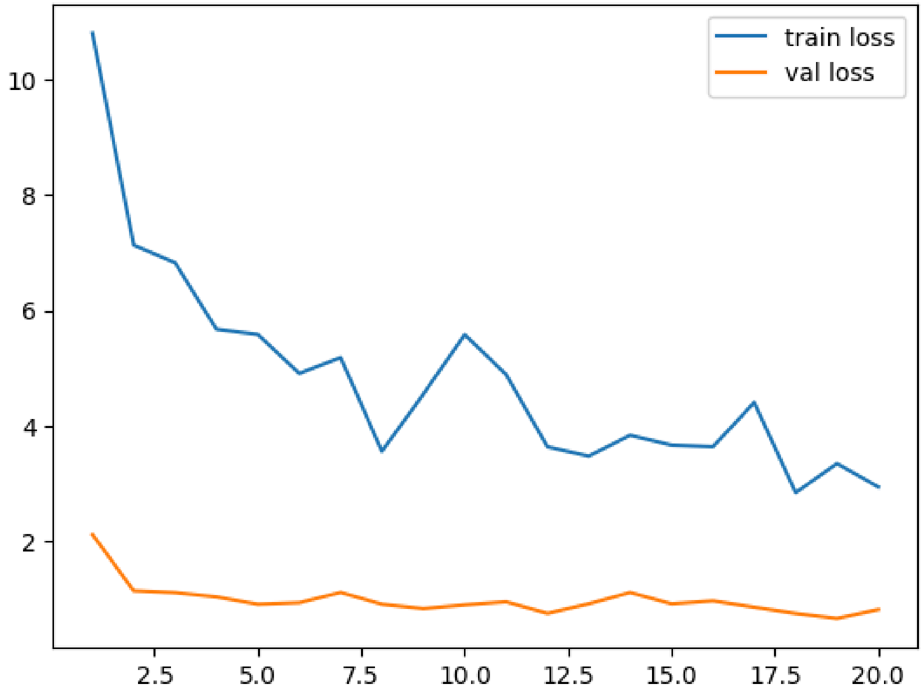
```
# Showing images
def imshow(img, text=None):
    npimg = img.numpy()
    plt.axis("off")
    if text:
```

```
        plt.text(20, 8, text, style='italic',fontweight='bold',
            bbox={'facecolor':'white', 'alpha':0.8, 'pad':10})

    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

best_weights_path = 'weights/best_triplet_weights.pth'
triplet_model.to(device)
triplet_model.load_state_dict(torch.load(best_weights_path))
triplet_model.eval()

test_anchor_imgs, test_pos_imgs, test_neg_imgs, test_anchor_ids, _, _ = next(iter(test_tr
test_anchor_imgs, test_pos_imgs, test_neg_imgs = test_anchor_imgs.to(device), test_pos_im
with torch.no_grad():
    test_anchor_feats = triplet_model.extract_feature(test_anchor_imgs)
    test_pos_feats = triplet_model.extract_feature(test_pos_imgs)
    test_neg_feats = triplet_model.extract_feature(test_neg_imgs)

class UnNormalize(object):
    def __init__(self, mean, std):
        self.mean = mean
        self.std = std

    def __call__(self, tensor):
        """
        Args:
            tensor (Tensor): Tensor image of size (C, H, W) to be normalized.
        Returns:
            Tensor: Normalized image.
        """
        for t, m, s in zip(tensor, self.mean, self.std):
            t.mul_(s).add_(m)
            # The normalize code -> t.sub_(m).div_(s)
        return tensor

unnormalizer = UnNormalize(mean=[0.5319, 0.4399, 0.3929],
                           std=[0.3076, 0.2898, 0.2907])

zip_test_data = zip(test_anchor_feats, test_pos_feats, test_neg_feats, test_anchor_imgs,
for test_anchor_feat, test_pos_feat, test_neg_feat, test_anchor_img, test_pos_img, test_n
    test_anchor_img = unnormalizer(test_anchor_img.detach().cpu().unsqueeze(0))
    test_pos_img = unnormalizer(test_pos_img.detach().cpu().unsqueeze(0))
    test_neg_img = unnormalizer(test_neg_img.detach().cpu().unsqueeze(0))
    concatenated = torch.cat((test_anchor_img, test_pos_img, test_neg_img), 0)
    anc_pos_distance = F.pairwise_distance(test_anchor_feat.unsqueeze(0), test_pos_feat.u
    anc_neg_distance = F.pairwise_distance(test_anchor_feat.unsqueeze(0), test_neg_feat.u
    imshow(torchvision.utils.make_grid(concatenated), f'Anc-Pos Distance: {anc_pos_distan
```

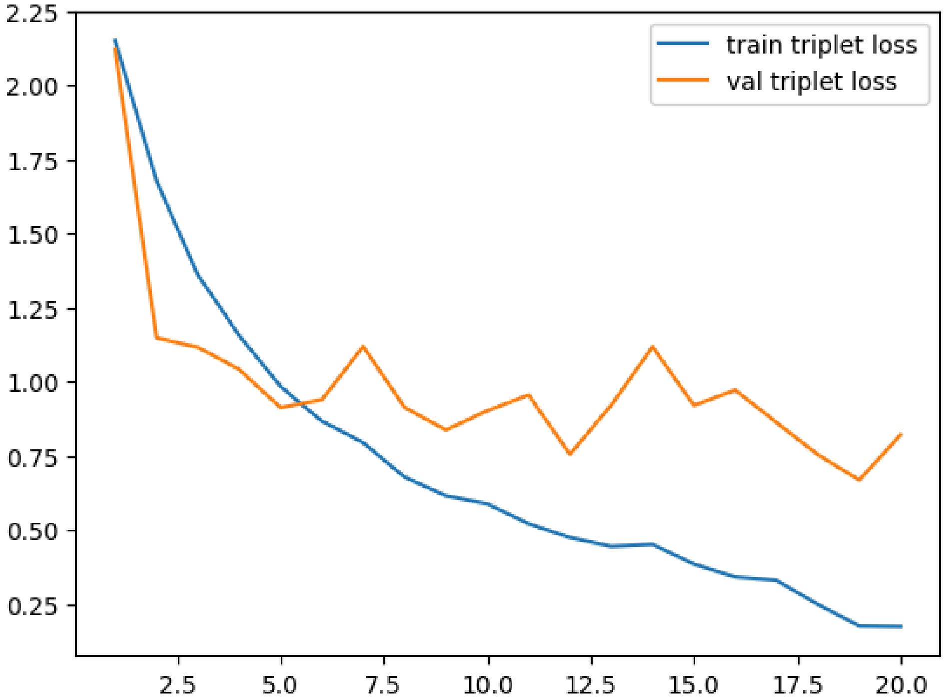
2.8 Plot loss history

In [ ]:

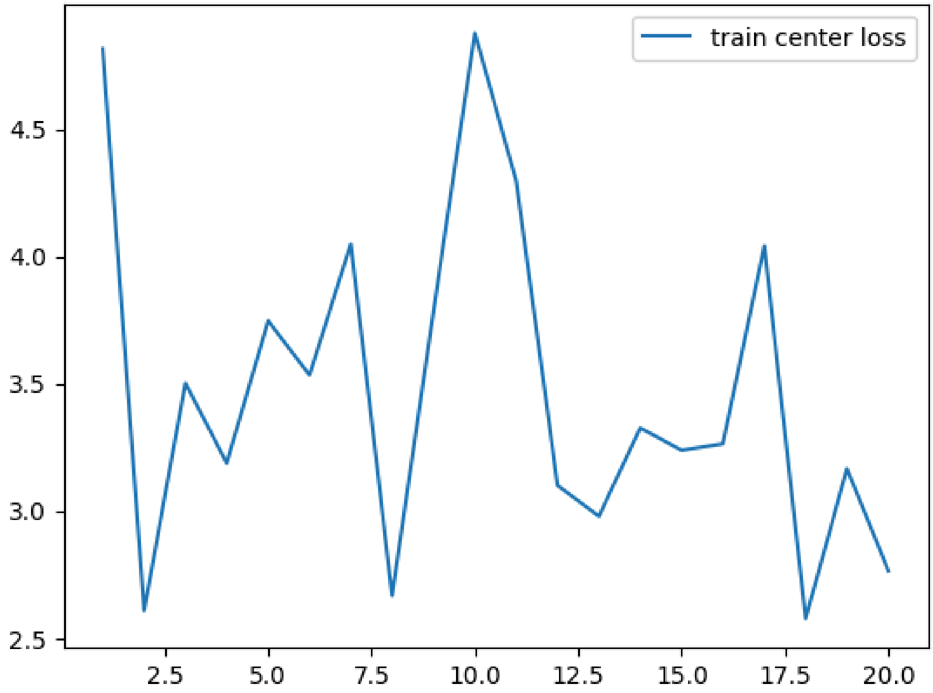
```
import matplotlib.pyplot as plt
plt.plot(np.arange(1, len(train_losses)+1), train_losses, label='train loss')
plt.plot(np.arange(1, len(val_losses)+1), val_losses, label='val loss')
plt.legend()
plt.show()
```



In [ ]:
```
import matplotlib.pyplot as plt
plt.plot(np.arange(1, len(train_losses)+1), train_triplet_losses, label='train triplet lo
plt.plot(np.arange(1, len(val_losses)+1), val_triplet_losses, label='val triplet loss')
plt.legend()
plt.show()
```



In [ ]:
```
import matplotlib.pyplot as plt
plt.plot(np.arange(1, len(train_losses)+1), train_center_losses, label='train center loss
plt.legend()
plt.show()
```



2.9 (TODO) Plot t-SNE

This section is similar to section 1.8 but the network is instead optimized using a combined triplet, center, and cross-entropy loss.

Instructions

TODO 12: Extract the feature vectors of the test set and store them as

`embeddings` : torch.FloatTensor = feature vectors of all images in the test set

`identities` : list or torch.Tensor or np.array = identities of all images in the test set

Hint
Use `FaceDataset` that is imported at `Common Dataset` section

WARNING!! Don't forget load its best weights and change to eval mode first

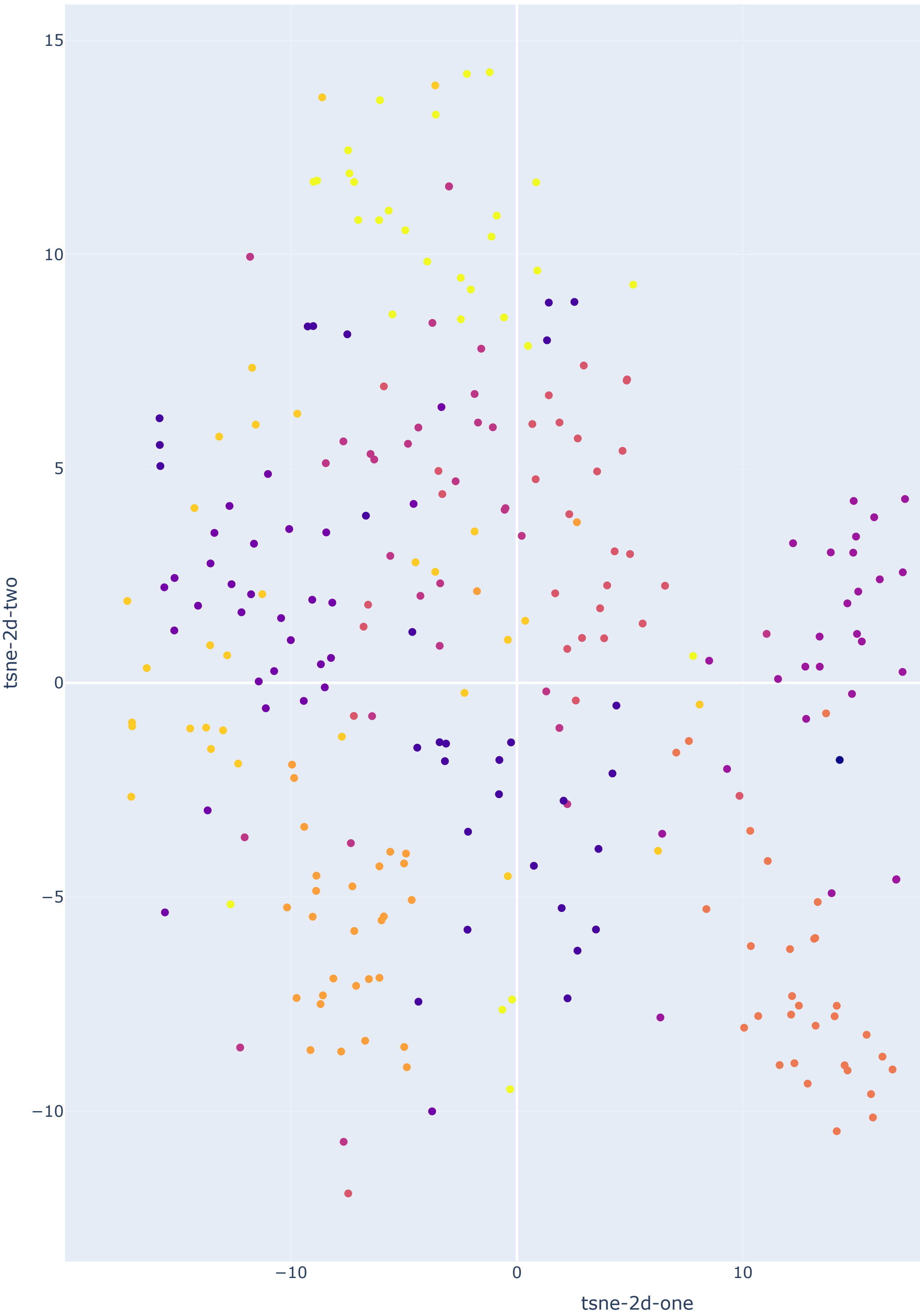
In [ ]:
```
test_batch_size = 32

# TODO 12: Extract the feature vectors of the test set and store them as
# `embeddings`: torch.FloatTensor = feature vectors of all images in the test set
# `identities`: list or torch.Tensor or np.array = identities of all images in the test s
# Hint      => Use `FaceDataset` that is imported at `Common Dataset` section
# WARNING!! => Don't forget load its best weights and change to eval mode first

triplet_model.load_state_dict(torch.load(best_weights_path))
triplet_model.eval()

embeddings = []
identities = []

for img, identity, _ in tqdm(FaceDataset(root_dir='large_prepared_data/test', transform=v
    img = img.unsqueeze(0)
    with torch.no_grad():
        embedding = triplet_model.extract_feature(img.to(device))
    embeddings.append(embedding.cpu().numpy())
    identities.append(identity)

embeddings = np.array(embeddings).squeeze()
identities = np.array(identities)
```

```
  0%|          | 0/301 [00:00<?, ?it/s]
```

In [ ]:
```
import time
from sklearn.manifold import TSNE

time_start = time.time()
tsne = TSNE(n_components=2, verbose=1, perplexity=30, n_iter=3000)
tsne_result = tsne.fit_transform(embeddings)
print('t-SNE done! Time elapsed: {} seconds'.format(time.time()-time_start))
```

```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 301 samples in 0.001s...
[t-SNE] Computed neighbors for 301 samples in 0.009s...
[t-SNE] Computed conditional probabilities for sample 301 / 301
[t-SNE] Mean sigma: 9.766650
[t-SNE] KL divergence after 250 iterations with early exaggeration: 58.167614
[t-SNE] KL divergence after 1300 iterations: 0.864549
t-SNE done! Time elapsed: 1.7905049324035645 seconds
```

In [ ]:
```
import plotly.express as px

label = []
idx = 0
id2label = dict()
for identity in identities:
  identity = int(identity)
  if identity not in id2label:
    id2label[identity] = idx
    idx += 1
  label.append(id2label[identity])

df_subset = pd.DataFrame({'label': label})

df_subset['tsne-2d-one'] = tsne_result[:,0]
df_subset['tsne-2d-two'] = tsne_result[:,1]

fig = px.scatter(df_subset, x="tsne-2d-one", y="tsne-2d-two", color="label", height=1000,
fig.update_layout(margin=dict(l=0, r=0, b=0, t=0))
fig.show()
```



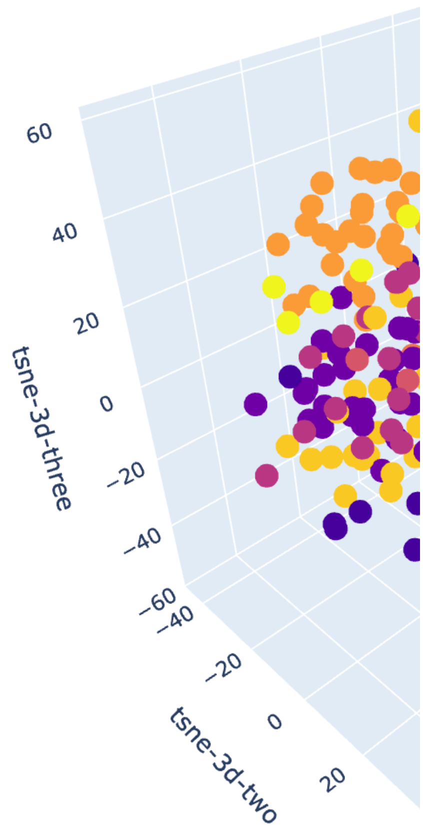
In [ ]:

```
import time
from sklearn.manifold import TSNE

time_start = time.time()
tsne = TSNE(n_components=3, verbose=1, perplexity=30, n_iter=3000)
tsne_result = tsne.fit_transform(embeddings)
print('t-SNE done! Time elapsed: {} seconds'.format(time.time()-time_start))
```

```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 301 samples in 0.001s...
[t-SNE] Computed neighbors for 301 samples in 0.010s...
[t-SNE] Computed conditional probabilities for sample 301 / 301
[t-SNE] Mean sigma: 9.766650
[t-SNE] KL divergence after 250 iterations with early exaggeration: 57.810661
[t-SNE] KL divergence after 850 iterations: 0.801615
t-SNE done! Time elapsed: 2.791259527206421 seconds
```

In [ ]:

```
import plotly.express as px

label = []
idx = 0
id2label = dict()
for identity in identities:
  identity = int(identity)
  if identity not in id2label:
    id2label[identity] = idx
    idx += 1
  label.append(id2label[identity])

df_subset = pd.DataFrame({'label': label})

df_subset['tsne-3d-one'] = tsne_result[:,0]
df_subset['tsne-3d-two'] = tsne_result[:,1]
df_subset['tsne-3d-three'] = tsne_result[:,2]

fig = px.scatter_3d(df_subset, x="tsne-3d-one", y="tsne-3d-two", z="tsne-3d-three", color
fig.update_layout(margin=dict(l=0, r=0, b=0, t=0))
fig.show()
```



2.10 (TODO) Embedding comparison

TODO 13: Compare the visualization of the triplet network to the siamese network. Which one is better and why?

Answer here: The same label points in Triplet network are closer compared to Siamese network.

(Optional) Try other tricks to get higher quality of face embeddings such as

1. GeM Pooling (ref: https://amaarora.github.io/2020/08/30/gempool.html)
2. Arcface (ref: https://arxiv.org/abs/1801.07698)
3. Hard negative mining (ref: https://omoindrot.github.io/triplet-loss)
4. Semi-hard mining

and plot t-SNE to compare with vanilla triplet loss

Part 3 : Face recognition using InfoNCE loss

In contrast to the contrastive and triplet loss that only pulls an image pair of the same identity to come closer and push a different one away, some alternative approaches, such as the N-pair (ref: paper) and InfoNCE (ref: paper) loss, take multiple positive images or/and multiple negative images into consideration. In this part, you are going to implement a variant of the InfoNCE loss where multiple positive and negative images are simutaneously utilized during the learning process. Moreover, the loss function implemented in this homework also adopts temperature scaling, a concept widely applied in many state-of-the-art self-supervised learning such as SimCLR (ref: paper), instead of the margin used in triplet and vanilla contrastive loss.

3.1 (TODO) Dataset and DataLoader

In this section, you are going to implement a dataloader for the combined loss. The dataloader should return the tuple `(anchor_img, pos_imgs, neg_imgs, anchor_label, pos_labels, neg_labels)`.

- The positive image must have the same identity as the anchor image.
- The negative image must have a different identity from the anchor image.

this dataloader will sample both multiple positive images and multiple negative images per anchor image

In [ ]:
```
import copy

class InfoNCEDataset(Dataset):
    def __init__(self, root_dir, transform=None, num_pos=4, num_neg=8):
      np.random.seed(123)
      random.seed(123)
      self.root_dir = root_dir
      self.transform = transform
      self.num_pos = num_pos
      self.num_neg = num_neg

      label_df = pd.read_csv(f'{root_dir}/label_df.csv')
      label_df = label_df.groupby('identity')['filename'].apply(list).reset_index().renam
      self.images = []
      self.labels = []
      self.label2indices = dict()
      self.load_images_to_memory_(label_df)

    def load_images_to_memory_(self, label_df):
      # load images and labels into memory
      # We have to relabel from identities to 0,1,2,...,num_classes-1
      # Relevant variables
      # 1. `self.images` = PIL images  (Also convert it to RGB)
      # 2. `self.label2indices` = the dictionary storing a label as a key and the indices
      # 3. `self.labels` = labels of images (relabeled)
      # Note: the index of images is from iterating over label_df
      self.images = []
      self.label2indices = defaultdict(list)
```

```
        self.labels = []
        img_idx = 0
        label_idx = 0
        identity2label = dict()
        for idx in tqdm(range(len(label_df))):
            row = label_df.iloc[idx]
            identity, filenames = row['identity'], row['filenames']

            if identity not in identity2label:
              identity2label[identity] = label_idx
              label_idx += 1

            label = identity2label[identity]

            for filename in filenames:
                self.images.append(Image.open(f'{self.root_dir}/{filename}').convert('RGB')
                self.labels.append(label)
                self.label2indices[label].append(img_idx)
                img_idx += 1


    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        '''
        sample images labels
        [args]
        - idx is an index of an image

        [intermediate]
        - anchor image is the image according to idx
        - positive images (dim: [num_pos, num_feat]) is a list of image that has the same i
        - negative images (dim: [num_neg, num_feat]) is a list of image that has the differ

        [return]
        - transformed_anchor_img is the transformed anchor image
        - transformed_pos_imgs are the tensor of transformed positive images
        - transformed_neg_img are the tensor of transformed negative images
        - anchor_label is the label of the anchor image
        - pos_labels are the list of positive labels
        - neg_labels are the list of negative labels
        '''

        anchor_img = self.images[idx]
        anchor_label = self.labels[idx]

        # TODO 14: sample positive images corresponding with the identity of anchor image
        ## condition:
        ## - there is no duplicated positive images
        ### Hint: Please use self.num_pos to determine the number of sampled positive image

        pos_indices = random.sample(self.label2indices[anchor_label], self.num_pos)

        pos_labels = [anchor_label] * self.num_pos
        pos_labels = torch.tensor(pos_labels)
        pos_imgs = [self.images[pos_idx] for pos_idx in pos_indices]


        # TODO 15: sample negative images that their identities differs from the identity o
        ## condition:
        ## - the list of negative images can contain identical identities
        ## - there is no duplicated negative images
        ### Hint: Please use self.num_neg to determine the number of sampled negative image

        neg_labels = np.random.choice(list(set(self.labels) - {anchor_label}), self.num_neg
        neg_indices = [random.choice(self.label2indices[neg_label]) for neg_label in neg_la
        neg_imgs = [self.images[neg_idx] for neg_idx in neg_indices]

        # TODO 16: utilize `self.transform' to convert anchor image, positive images, and n
        # WARNING!: Don't forget to convert to be tensors
        if self.transform is not None:
            anchor_img = self.transform(anchor_img)
            pos_imgs = [self.transform(pos_img) for pos_img in pos_imgs]
```

```
            pos_imgs = torch.stack(pos_imgs)
            neg_imgs = [self.transform(neg_img) for neg_img in neg_imgs]
            neg_imgs = torch.stack(neg_imgs)

        return anchor_img, pos_imgs, neg_imgs, anchor_label, pos_labels, neg_labels
```

In [ ]:
```
temp_ds = InfoNCEDataset(root_dir='large_prepared_data/train', transform=train_transform)
```

```
  0%|          | 0/80 [00:00<?, ?it/s]
```

In [ ]:
```
next(iter(temp_ds))[3:]
```

Out[ ]:
```
(0, tensor([0, 0, 0, 0]), array([67, 18, 58, 48, 74, 33, 47, 26]))
```

In [ ]:
```
image_size = 224
train_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.CenterCrop(image_size),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5319, 0.4399, 0.3929],
                         std=[0.3076, 0.2898, 0.2907])
])

val_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.CenterCrop(image_size),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5319, 0.4399, 0.3929],
                         std=[0.3076, 0.2898, 0.2907])
])

train_batch_size = 16
val_batch_size = 16
test_batch_size = 16

# TODO 17: declare the datasets and the dataloaders
train_infonce_dataset = InfoNCEDataset(root_dir='large_prepared_data/train', transform=tr
train_infonce_dataloader = DataLoader(train_infonce_dataset, batch_size=train_batch_size,

val_infonce_dataset = InfoNCEDataset(root_dir='large_prepared_data/val', transform=val_tr
val_infonce_dataloader = DataLoader(val_infonce_dataset, batch_size=val_batch_size, shuff

test_infonce_dataset = InfoNCEDataset(root_dir='large_prepared_data/test', transform=val_
test_infonce_dataloader = DataLoader(test_infonce_dataset, batch_size=test_batch_size, sh
```

```
  0%|          | 0/80 [00:00<?, ?it/s]
  0%|          | 0/10 [00:00<?, ?it/s]
  0%|          | 0/10 [00:00<?, ?it/s]
```

3.2 (TODO) Image encoder

Image encoder consists of 3 modules

1. Feature extractor (ResNet18) is for mapping an image to a feature map
2. Global pooling (Global average pooling or use flatten instead) is for converting a feature map to a feature vector
3. Projection layer consists of 2 MLP layers to reduce dimension of feature vector together with a relu activation function at the middle of MLP layers

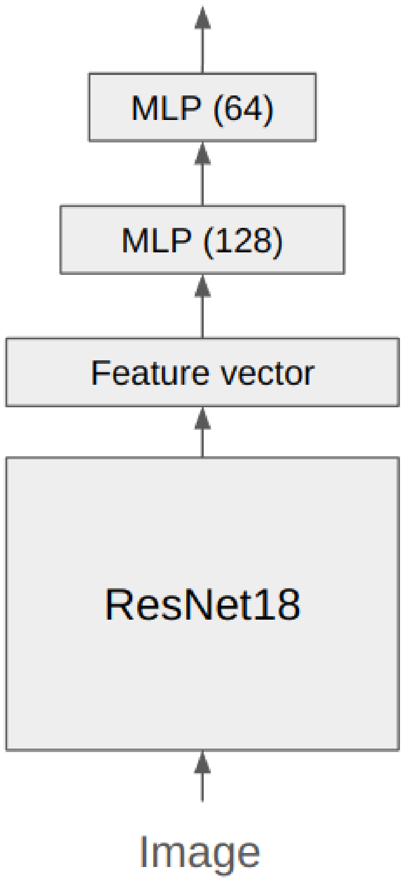
Note

1. ResNet18 can call via `torchvision` library (How to use torchvision: https://pytorch.org/vision/stable/models.html)
2. ResNet18 we use will not load pretrained weights
3. We will use ResNet18 only extracting feature maps.

Hint!

1. For global average pooling, read the documentation of `nn.AdaptiveAvgPool2d`

Figure 8 Image Encoder



In [ ]:
```
class ImageEncoder(nn.Module):
  # TODO 18: implement the image encoder according to the above figure
  def __init__(self):
      super(ImageEncoder, self).__init__()
      self.resnet = torchvision.models.resnet18()
      self.resnet = nn.Sequential(*(list(self.resnet.children())[:-1]))
      self.global_pool = nn.AdaptiveAvgPool2d(1)

      self.mlp = nn.Sequential(
        nn.Linear(512, 128),
        nn.ReLU(),
        nn.Linear(128, 64)
      )


  def extract_feature(self, x):
      x = self.resnet(x)
      x = self.global_pool(x)
      x = x.view(x.size(0), -1)
      x = self.mlp(x)
      return x

  def forward(self, input1, input2, input3):
      output1 = self.extract_feature(input1)
      output2 = [self.extract_feature(input2[i]) for i in range(input2.size(0))]
      output2 = torch.stack(output2)
      output3 = [self.extract_feature(input3[i]) for i in range(input3.size(0))]
      output3 = torch.stack(output3)

      return output1, output2, output3
```

3.3 (TODO) InfoNCE loss

Information Noise Contrastive Estimation (InfoNCE) loss (ref: blog, paper) is a self-supervised training objective aiming to maximizes the agreement between positive samples and minimizes the agreement between negative samples in the learned representation space. Although primarily designed for self-supervised learning, it could also be used in supervised setting by utilizing information from the labeled data.

When under self-supervision, images are sampled in batches, each of which is then augmented using two different augmentation policies. The contrastive loss is subsequently computed to maximize the similarity of the images of the same origin before augmentation while minimizing the similarity of the ones with different origins. In supervised learning, the objective maximizes the similarity of the images with the same identity (class) instead of origin. **In this part, we will train the InfoNCE loss under the supervised setting.**

To put it simply, the InfoNCE loss revolves around three types of images:

1. Anchor image: an initial image
2. Positive images: the sampled images with the same identity as the anchor image
3. Negative images: the sampled images whose identity differs from that of the anchor image

The objective is to maximize the cosine similarity between groups 1 and 2 while pushing groups 1 and 3 away. Thus, the InfoNCE maximizes the following objective:

$$InfoNCE\ Loss_i = \frac{1}{B \cdot P} \sum_{i=1}^{B} \sum_{p \in P(i)} \frac{e^{(cos(f_i, f_p)/\tau)}}{(e^{cos(f_i, f_p)/\tau)} + \sum_{n \in N(i)} e^{(cos(f_i, f_n)/\tau)}} \tag{1}$$

Notation

- B = batch size
- P = the number of positive images per anchor
- i = index of the anchor feature vector
- $P(i)$ = the sampled positive feature vectors
- $N(i)$ = the sampled negative feature vectors
- $cos(f_i, f_p)$ = cosine similarity between f_i and f_p
- f_i = anchor feature vector
- f_p = positive feature vector
- f_n = negative feature vector
- τ = temperature (a scaling factor for increasing/reducing the magnitude of logits)

Hint use `nn.CrossEntropy` to compute the exponential part

In [ ]:
```
class InfoNCELoss(nn.Module):
  # TODO 19: implement the InfoNCE loss
  def __init__(self, device, temperature):
    super(InfoNCELoss, self).__init__()
    self.device = device
    self.temperature = temperature

  def forward(self, anchor_feat, pos_feats, neg_feats):
    e_neg = torch.exp(torch.cosine_similarity(anchor_feat.unsqueeze(1), neg_feats, dim=2)
    sum_e_neg = torch.sum(e_neg, dim=1)
    sum_e_neg = sum_e_neg.unsqueeze(1).repeat(1, len(pos_feats[0]))

    e_pos = torch.exp(torch.cosine_similarity(anchor_feat.unsqueeze(1), pos_feats, dim=2)

    sum_e_pos = torch.sum(e_pos/(e_pos + sum_e_neg), dim=1)

    loss = (1/(len(anchor_feat)*len(pos_feats[0]))) * torch.sum(sum_e_pos)

    return loss
```

3.4 (TODO) Initializing model, criterions, optimizers, hyparameters and scheduler

In [ ]:
```
temperature = 0.2

infonce_model = ImageEncoder()
device = 'cuda' if torch.cuda.is_available() else 'cpu'
train_infonce_criterion = InfoNCELoss(device, temperature)
val_infonce_criterion = InfoNCELoss(device, temperature)

infonce_optimizer = optim.Adam(infonce_model.parameters(), lr=5e-4)
infonce_scheduler = optim.lr_scheduler.ReduceLROnPlateau(infonce_optimizer, 'min', patien
```

3.5 (TODO) Training loop

Figure 9 How to connect the InfoNCE loss

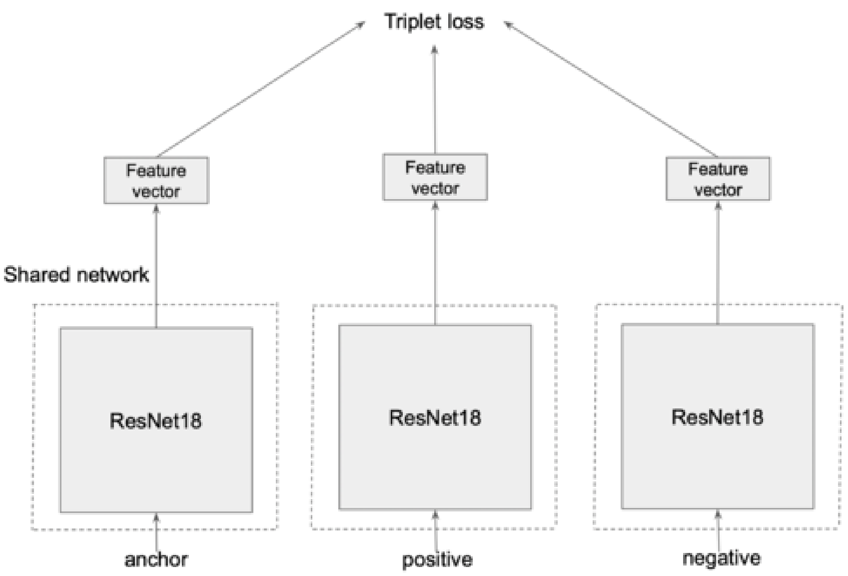


Fig 9 demonstrates how to connect multiple feature vectors to compute InfoNCE loss. The feature extractor shares all parameters between input images.

The training procedure of a infonce network consists of the following steps:

1. Foward pass `anchor_img`
2. Foward pass `pos_imgs`
3. Foward pass `neg_img`
4. Calculate all pairs cosine similarities between `anchor_img` and `pos_imgs`.
5. Calculate all pairs cosine similarities between `anchor_img` and `neg_imgs`.
6. Compute InfoNCE loss and take an average over them
7. Use the loss in step 6 to update the model
8. Repeat step 1-8 till converged

TODO 20: move training data to GPU memory, extract anchor, positive, negative features from images, and then use them to compute InfoNCE loss

TODO 21: set zero gradients

TODO 22 : back propagate at infonce network and step the optimizer

TODO 23: move validation data to GPU memory, extract anchor, positive, negative features from images, and then use them to compute InfoNCE loss

In [ ]:
```
num_epochs = 10
device = "cuda" if torch.cuda.is_available() else "cpu"

os.makedirs('weights', exist_ok=True)
best_weights_path = 'weights/best_infonce_weights.pth'
infonce_model.to(device)
min_val_loss = float('inf')

train_losses = []
val_losses = []

for epoch in tqdm(range(num_epochs)):
    infonce_model.train()
    total_train_loss = 0
    for anchor_img, pos_imgs, neg_imgs, anchor_label, pos_labels, neg_labels in tqdm(trai

        # TODO 20: feed data to the infonce model and compute infonce loss
        anchor_img, pos_imgs, neg_imgs = anchor_img.to(device), pos_imgs.to(device), neg_
        train_anchor_feat, train_pos_feats, train_neg_feats = infonce_model(anchor_img, p
        train_loss = train_infonce_criterion(train_anchor_feat, train_pos_feats, train_ne
        total_train_loss += train_loss.item()

        # TODO 21: set zero gradients
        infonce_optimizer.zero_grad()

        # TODO 22 : back propagate at infonce network and step the optimizer
        train_loss.backward()

    current_train_loss = total_train_loss / len(train_infonce_dataloader)
    train_losses.append(current_train_loss)

    total_val_loss = 0
    infonce_model.eval()

    for val_anchor_img, val_pos_imgs, val_neg_imgs, _, _, _ in val_infonce_dataloader:
      # TODO 23: feed data to the infonce model and compute infonce loss
      val_anchor_img, val_pos_imgs, val_neg_imgs = val_anchor_img.to(device), val_pos_img
      with torch.no_grad():
          val_anchor_feat, val_pos_feats, val_neg_feats = infonce_model(val_anchor_img, v
          val_infonce_loss = val_infonce_criterion(val_anchor_feat, val_pos_feats, val_ne

      val_loss = val_infonce_loss
      total_val_loss += val_loss.item()
    current_val_loss = total_val_loss / len(val_infonce_dataloader)
    val_losses.append(current_val_loss)

    if current_val_loss < min_val_loss:
        min_val_loss = current_val_loss
        torch.save(infonce_model.state_dict(), best_weights_path)
    print(f'Epoch {epoch+1} '
      f'- Train loss = {current_train_loss:.4f} '
      f'- Val loss = {current_val_loss:.4f} '
      f'- best min_val_loss = {min_val_loss:.4f} '
      f'- lr = {infonce_optimizer.param_groups[0]["lr"]:.8f}'
    )
    infonce_scheduler.step(current_val_loss)
```

3.6 Visualization

The visualization below displays an anchor, positive, and negative image and their respective similarity.

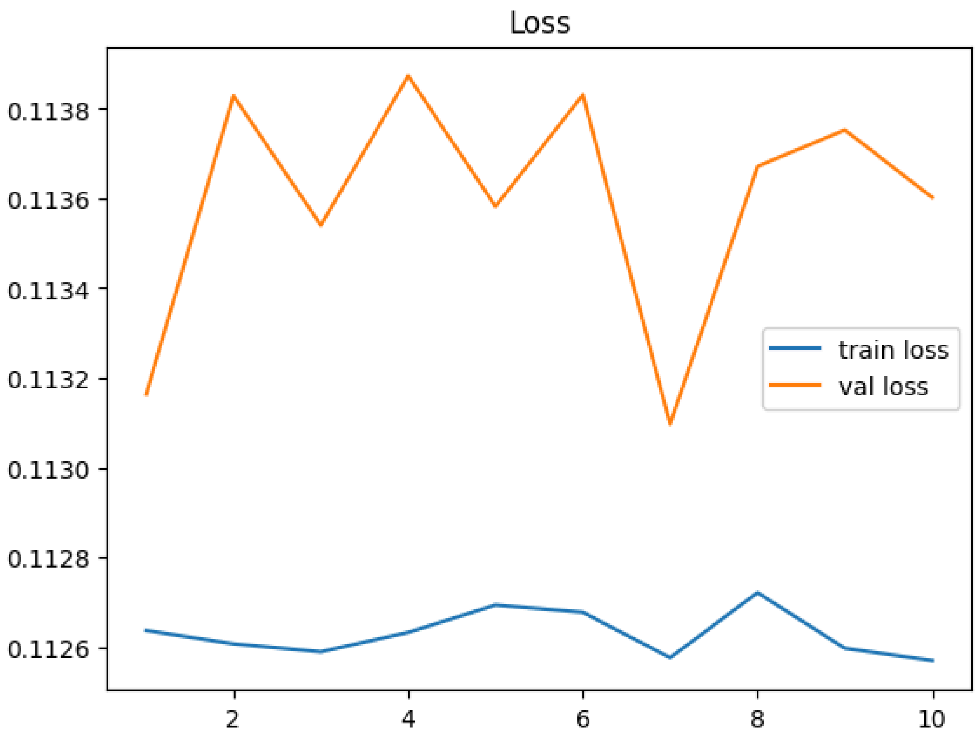
In [ ]:
```
# Showing images
def imshow(img, text=None):
    npimg = img.numpy()
    plt.axis("off")
    if text:
        plt.text(20, 8, text, style='italic',fontweight='bold',
            bbox={'facecolor':'white', 'alpha':0.8, 'pad':10})

    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

```
best_weights_path = 'weights/best_infonce_weights.pth'
infonce_model.to(device)
infonce_model.load_state_dict(torch.load(best_weights_path))
infonce_model.eval()

test_anchor_imgs, batch_test_pos_imgs, batch_test_neg_imgs, test_anchor_ids, _, _ = next(

test_pos_imgs = batch_test_pos_imgs[:, 0, :, :]
test_neg_imgs = batch_test_neg_imgs[:, 0, :, :]

test_anchor_imgs, test_pos_imgs, test_neg_imgs = test_anchor_imgs.to(device), test_pos_im
with torch.no_grad():
    test_anchor_feats = infonce_model.extract_feature(test_anchor_imgs)
    test_pos_feats = infonce_model.extract_feature(test_pos_imgs)
    test_neg_feats = infonce_model.extract_feature(test_neg_imgs)

class UnNormalize(object):
    def __init__(self, mean, std):
        self.mean = mean
        self.std = std

    def __call__(self, tensor):
        """
        Args:
            tensor (Tensor): Tensor image of size (C, H, W) to be normalized.
        Returns:
            Tensor: Normalized image.
        """
        for t, m, s in zip(tensor, self.mean, self.std):
            t.mul_(s).add_(m)
            # The normalize code -> t.sub_(m).div_(s)
        return tensor

unnormalizer = UnNormalize(mean=[0.5319, 0.4399, 0.3929],
                           std=[0.3076, 0.2898, 0.2907])

zip_test_data = zip(test_anchor_feats, test_pos_feats, test_neg_feats, test_anchor_imgs,
for test_anchor_feat, test_pos_feat, test_neg_feat, test_anchor_img, test_pos_img, test_n
    test_anchor_img = unnormalizer(test_anchor_img.detach().cpu().unsqueeze(0))
    test_pos_img = unnormalizer(test_pos_img.detach().cpu().unsqueeze(0))
    test_neg_img = unnormalizer(test_neg_img.detach().cpu().unsqueeze(0))
    concatenated = torch.cat((test_anchor_img, test_pos_img, test_neg_img), 0)
    anc_pos_similarity = F.cosine_similarity(test_anchor_feat.unsqueeze(0), test_pos_feat
    anc_neg_similarity = F.cosine_similarity(test_anchor_feat.unsqueeze(0), test_neg_feat
    imshow(torchvision.utils.make_grid(concatenated), f'Anc-Pos similarity: {anc_pos_simi
```

3.7 Plot loss history

In [ ]:

```
import matplotlib.pyplot as plt
plt.plot(np.arange(1, len(train_losses)+1), train_losses, label='train loss')
plt.plot(np.arange(1, len(val_losses)+1), val_losses, label='val loss')
plt.legend()
plt.title('Loss')
plt.show()
```



3.8 (TODO) Plot t-SNE

This section is similar to section 1.8

Instructions

TODO 24: Extract the feature vectors of the test set and store them as

`embeddings` : torch.FloatTensor = feature vectors of all images in the test set

`identities` : list or torch.Tensor or np.array = identities of all images in the test set

Hint
Use `FaceDataset` that is imported at `Common Dataset` section

WARNING!! Don't forget load its best weights and change to eval mode first

In [ ]:
```
test_batch_size = 32

# TODO 24: Extract the feature vectors of the test set and store them as
# `embeddings`: torch.FloatTensor = feature vectors of all images in the test set
# `identities`: list or torch.Tensor or np.array = identities of all images in the test s
# Hint      => Use `FaceDataset` that is imported at `Common Dataset` section
# WARNING!! => Don't forget load its best weights and change to eval mode first

infonce_model.load_state_dict(torch.load(best_weights_path))
infonce_model.eval()

embeddings = []
identities = []

for img, identity, _ in tqdm(FaceDataset(root_dir='large_prepared_data/test', transform=v
    img = img.unsqueeze(0)
    with torch.no_grad():
        embedding = infonce_model.extract_feature(img.to(device))
    embeddings.append(embedding.cpu().numpy())
    identities.append(identity)

embeddings = np.array(embeddings).reshape(301, -1)
identities = np.array(identities).reshape(301, -1)
```

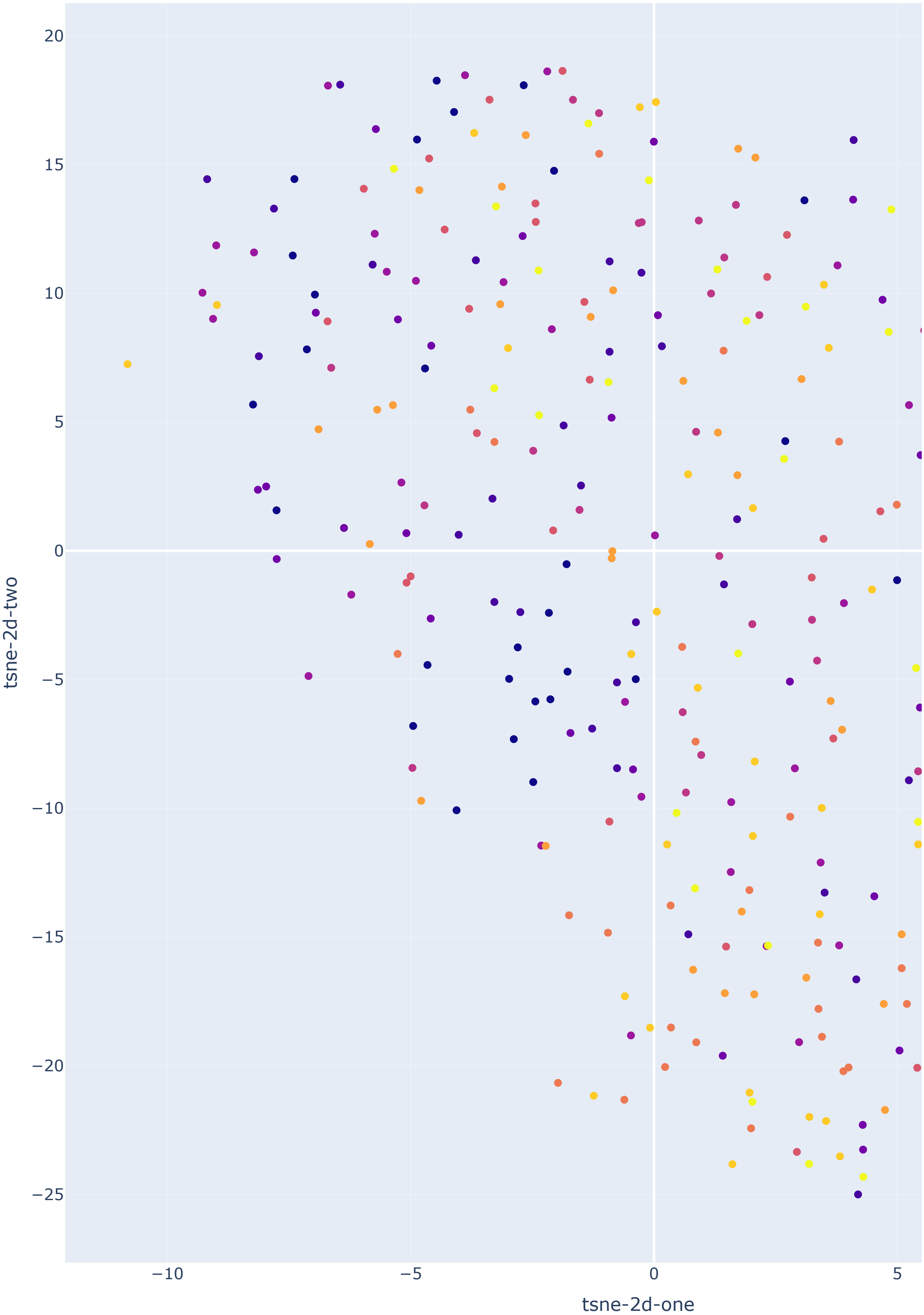
```
  0%|          | 0/301 [00:00<?, ?it/s]
```

In [ ]:
```
import time
from sklearn.manifold import TSNE

time_start = time.time()
tsne = TSNE(n_components=2, verbose=1, perplexity=30, n_iter=3000)
tsne_result = tsne.fit_transform(embeddings)
print('t-SNE done! Time elapsed: {} seconds'.format(time.time()-time_start))
```

```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 301 samples in 0.000s...
[t-SNE] Computed neighbors for 301 samples in 0.012s...
[t-SNE] Computed conditional probabilities for sample 301 / 301
[t-SNE] Mean sigma: 0.141362
[t-SNE] KL divergence after 250 iterations with early exaggeration: 57.020065
[t-SNE] KL divergence after 1350 iterations: 0.823604
t-SNE done! Time elapsed: 1.8997933864593506 seconds
```

In [ ]:
```
import plotly.express as px

label = []
idx = 0
id2label = dict()
for identity in identities:
  identity = int(identity)
  if identity not in id2label:
    id2label[identity] = idx
    idx += 1
  label.append(id2label[identity])

df_subset = pd.DataFrame({'label': label})

df_subset['tsne-2d-one'] = tsne_result[:,0]
df_subset['tsne-2d-two'] = tsne_result[:,1]

fig = px.scatter(df_subset, x="tsne-2d-one", y="tsne-2d-two", color="label", height=1000,
fig.update_layout(margin=dict(l=0, r=0, b=0, t=0))
fig.show()
```

```
/tmp/ipykernel_34/3459061270.py:7: DeprecationWarning:

Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future.
Ensure you extract a single element from your array before performing this operation. (De
precated NumPy 1.25.)
```

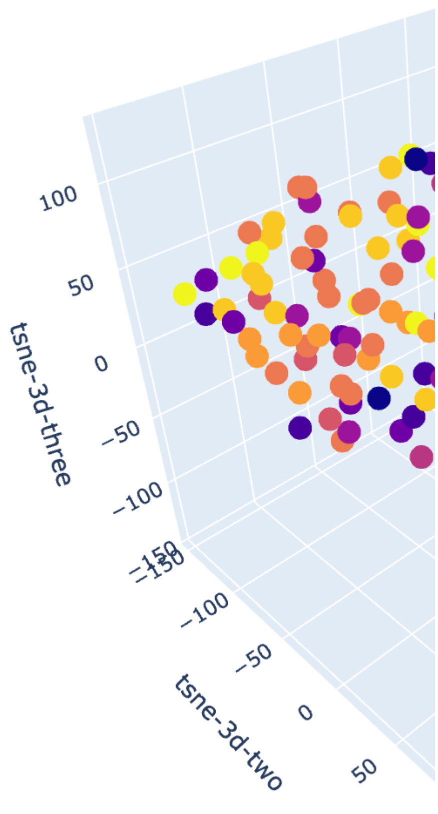


In [ ]:
```
import time
from sklearn.manifold import TSNE

time_start = time.time()
tsne = TSNE(n_components=3, verbose=1, perplexity=30, n_iter=3000)
tsne_result = tsne.fit_transform(embeddings)
print('t-SNE done! Time elapsed: {} seconds'.format(time.time()-time_start))
```

```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 301 samples in 0.000s...
[t-SNE] Computed neighbors for 301 samples in 0.007s...
[t-SNE] Computed conditional probabilities for sample 301 / 301
[t-SNE] Mean sigma: 0.141362
[t-SNE] KL divergence after 250 iterations with early exaggeration: 60.707893
[t-SNE] KL divergence after 3000 iterations: 0.733796
t-SNE done! Time elapsed: 9.166933298110962 seconds
```

In [ ]:
```python
import plotly.express as px

label = []
idx = 0
id2label = dict()
for identity in identities:
  identity = int(identity)
  if identity not in id2label:
    id2label[identity] = idx
    idx += 1
  label.append(id2label[identity])

df_subset = pd.DataFrame({'label': label})

df_subset['tsne-3d-one'] = tsne_result[:,0]
df_subset['tsne-3d-two'] = tsne_result[:,1]
df_subset['tsne-3d-three'] = tsne_result[:,2]

fig = px.scatter_3d(df_subset, x="tsne-3d-one", y="tsne-3d-two", z="tsne-3d-three", color
fig.update_layout(margin=dict(l=0, r=0, b=0, t=0))
fig.show()
```

```
/tmp/ipykernel_34/3123562770.py:7: DeprecationWarning:

Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future.
Ensure you extract a single element from your array before performing this operation. (De
precated NumPy 1.25.)
```



3.9 (TODO) Please analyze and compare between the t-SNE of triplet network the t-SNE of InfoNCE network. Why are those t-SNEs not in the same way?

TODO 25: Answer the above question

Hint The reason is about our dataset

Answer: Triplet network are work better way. In the other hand, InfoNCE is better than Siamese but it's still blurly.

Part 4 : Face verification evaluation

In HW3, we use the ROC curve to measure the performance of the face verification task. Similarly, we will use this metric to measure the performance of our NN.

TODO 26: Use the Siamese network, network trained using triplet loss, and network trained using InfoNCE loss to extract the image features from the test set.

In [ ]:
```
val_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.CenterCrop(img_size),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5319, 0.4399, 0.3929],
                         std=[0.3076, 0.2898, 0.2907])
])


test_siamese_dataset = SiameseDataset(root_dir='large_prepared_data/test', transform=val_
test_siamese_dataloader = DataLoader(test_siamese_dataset, batch_size=test_batch_size, sh

siamese_embeddings1 = []
siamese_embeddings2 = []
triplet_embeddings1 = []
triplet_embeddings2 = []
infonce_embeddings1 = []
infonce_embeddings2 = []
labels = []

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

siamese_model.to(device)
triplet_model.to(device)
infonce_model.to(device)

siamese_model.load_state_dict(torch.load('weights/best_siamese_weights.pth'))
triplet_model.load_state_dict(torch.load('weights/best_triplet_weights.pth'))
infonce_model.load_state_dict(torch.load('weights/best_infonce_weights.pth'))

siamese_model.eval()
triplet_model.eval()
infonce_model.eval()

for batch_img1, batch_img2, batch_label in tqdm(test_siamese_dataloader):
  batch_img1, batch_img2, batch_label = batch_img1.to(device), batch_img2.to(device), bat
  with torch.no_grad():
    # TODO 26: extract features with both siamese network, triplet network, infonce netwo
    # and keep embeddings in provided lists according to variable names
    siamese_embedding1 = siamese_model.extract_feature(batch_img1)
    siamese_embedding2 = siamese_model.extract_feature(batch_img2)
    triplet_embedding1 = triplet_model.extract_feature(batch_img1)
    triplet_embedding2 = triplet_model.extract_feature(batch_img2)
    infonce_embedding1 = infonce_model.extract_feature(batch_img1)
    infonce_embedding2 = infonce_model.extract_feature(batch_img2)
```

```
    siamese_embeddings1.append(siamese_embedding1)
    siamese_embeddings2.append(siamese_embedding2)
    triplet_embeddings1.append(triplet_embedding1)
    triplet_embeddings2.append(triplet_embedding2)
    infonce_embeddings1.append(infonce_embedding1)
    infonce_embeddings2.append(infonce_embedding2)

  labels.append(batch_label)

siamese_embeddings1 = torch.cat(siamese_embeddings1)
siamese_embeddings2 = torch.cat(siamese_embeddings2)
triplet_embeddings1 = torch.cat(triplet_embeddings1)
triplet_embeddings2 = torch.cat(triplet_embeddings2)
infonce_embeddings1 = torch.cat(infonce_embeddings1)
infonce_embeddings2 = torch.cat(infonce_embeddings2)
labels = torch.cat(labels)
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
```

TODO 27: Measure the similarity score between the two feature vectors with cosine similarity.

HINT You can use `nn.CosineSimilarity` (ref: https://pytorch.org/docs/stable/generated/torch.nn.CosineSimilarity.html).

In [ ]:

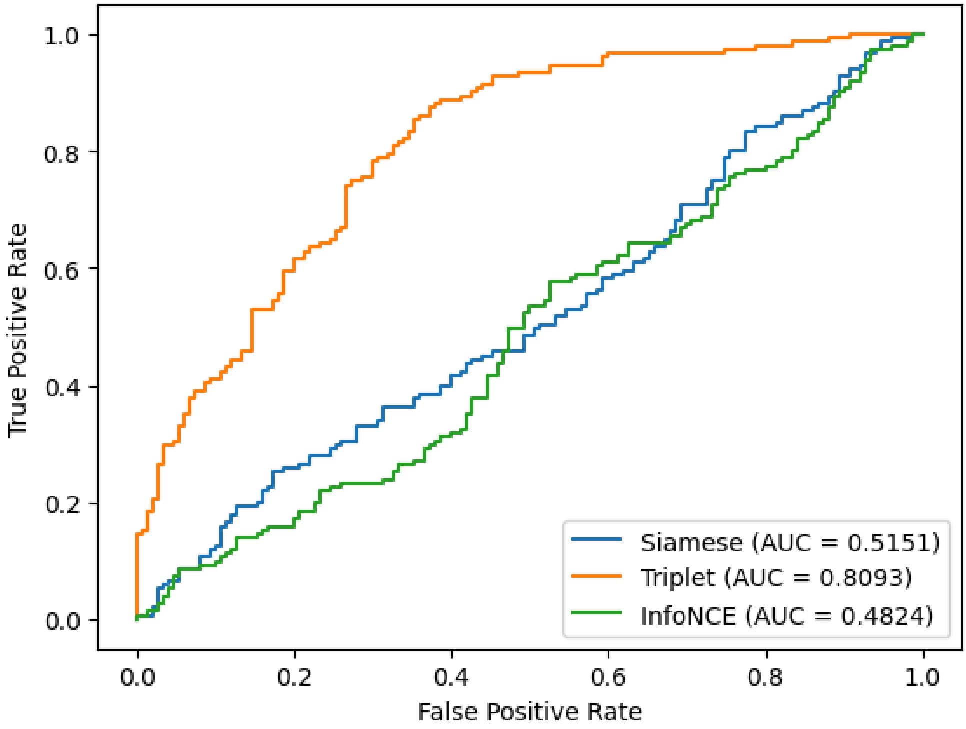
```
def compute_pairs_cosine_sim(input1, input2):
  # TODO 27: implement cosine similarity function that can compute multiple pairs at the

  cosine_sim = F.cosine_similarity(input1, input2, dim=1)
  return cosine_sim

siamese_scores = compute_pairs_cosine_sim(siamese_embeddings1, siamese_embeddings2)
triplet_scores = compute_pairs_cosine_sim(triplet_embeddings1, triplet_embeddings2)
infonce_scores = compute_pairs_cosine_sim(infonce_embeddings1, infonce_embeddings2)
```

TODO 28

Plot a ROC curve to compare the performance siamese, triplet, infonce networks. Which one is better and why?

In [ ]:

```
import matplotlib.pyplot as plt
# TODO 28: calculate true positive rate and false positive rate
# and plot both the ROC curve of siamese network and the ROC curve of triplet network
# together with AUC score

from sklearn.metrics import roc_curve, roc_auc_score

siamese_fpr, siamese_tpr, _ = roc_curve(labels.cpu().numpy(), siamese_scores.cpu().numpy(
siamese_auc = roc_auc_score(labels.cpu().numpy(), siamese_scores.cpu().numpy())

triplet_fpr, triplet_tpr, _ = roc_curve(labels.cpu().numpy(), triplet_scores.cpu().numpy(
triplet_auc = roc_auc_score(labels.cpu().numpy(), triplet_scores.cpu().numpy())

infonce_fpr, infonce_tpr, _ = roc_curve(labels.cpu().numpy(), infonce_scores.cpu().numpy(
infonce_auc = roc_auc_score(labels.cpu().numpy(), infonce_scores.cpu().numpy())

plt.plot(siamese_fpr, siamese_tpr, label=f'Siamese (AUC = {siamese_auc:.4f})')
plt.plot(triplet_fpr, triplet_tpr, label=f'Triplet (AUC = {triplet_auc:.4f})')
plt.plot(infonce_fpr, infonce_tpr, label=f'InfoNCE (AUC = {infonce_auc:.4f})')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.show()
```



From ROC curve Triplet is best network because the area under curve is more