```python
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 = {}
    self.images = []
    self.identities = []

    for _, row in label_df.iterrows():
        identity = row['identity']
        filenames = row['filenames']

        self.data[identity] = [Image.open(f"{self.root_dir}/{filename}").convert('RGB') for filename in filenames]
        self.images.extend(self.data[identity])
        self.identities.append(identity)

    # TODO 2: keep all unique identities as list with `self.unique_identities`
    # in `self.unique_identities` as a numpy array.
    self.unique_identities = np.array(list(self.data.keys()))
```

```python
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:
        identity = np.random.choice(self.unique_identities)
        img1, img2 = np.random.choice(self.data[identity], 2, replace=False)
        label = 1
    else:
        identity1, identity2 = np.random.choice(self.unique_identities, 2, replace=False)
        img1 = np.random.choice(self.data[identity1])
        img2 = np.random.choice(self.data[identity2])
        label = 0

    if self.transform:
        img1 = self.transform(img1)
        img2 = self.transform(img2)

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

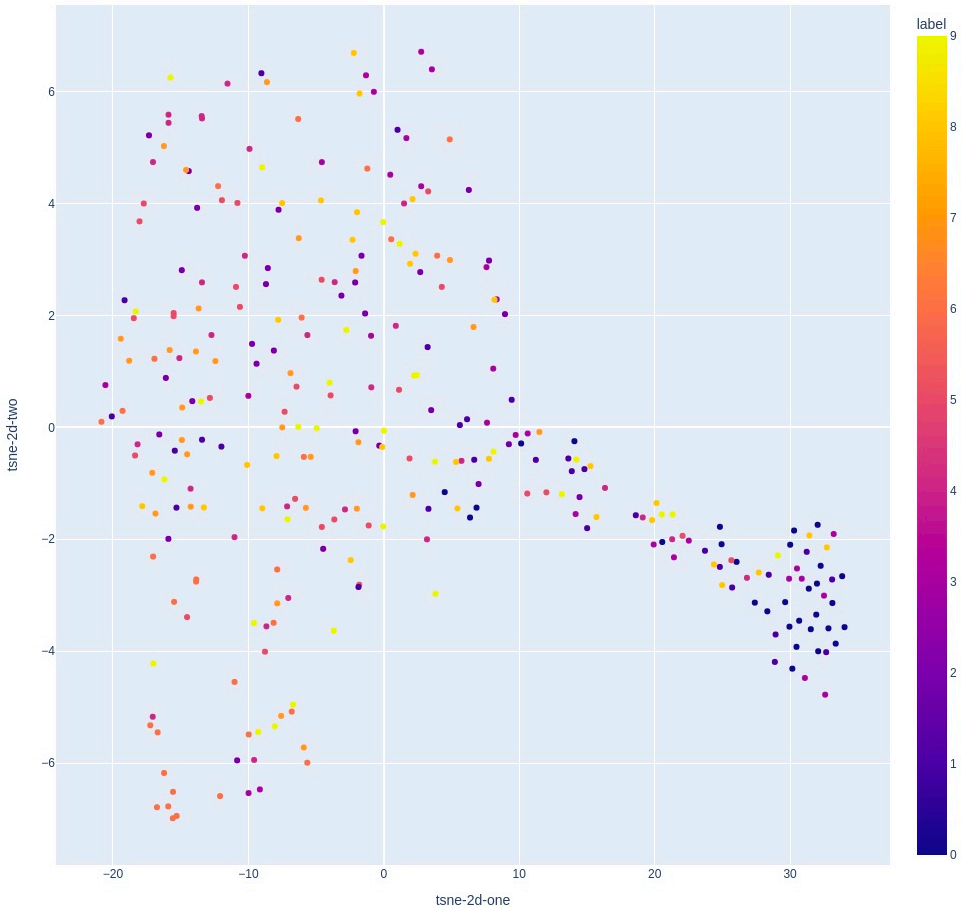
```python
# TODO 4: declare the datasets and the dataloaders
train_siamese_dataset = SiameseDataset(root_dir='large_prepared_data/train', transform=train_transform)
train_siamese_dataloader = DataLoader(train_siamese_dataset, batch_size=train_batch_size, shuffle=True, num_workers=2)

val_siamese_dataset = SiameseDataset(root_dir='large_prepared_data/val', transform=val_transform)
val_siamese_dataloader = DataLoader(val_siamese_dataset, batch_size=val_batch_size, shuffle=False, num_workers=2)

test_siamese_dataset = SiameseDataset(root_dir='large_prepared_data/test', transform=val_transform)
test_siamese_dataloader = DataLoader(test_siamese_dataset, batch_size=test_batch_size, shuffle=False, num_workers=2)
```

```python
class SiameseNetwork(nn.Module):
    # TODO 5: implement the siamese network
    def __init__(self):
        super().__init__()

        self.resnet18 = torchvision.models.resnet18(pretrained=False)
        self.features = nn.Sequential(*(list(self.resnet18.children())[:-1]))

        self.fc1 = nn.Linear(self.resnet18.fc.in_features, 256)
        self.fc2 = nn.Linear(256, 128)

        self.relu = nn.ReLU()  # ReLU activation

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

        x = nn.functional.adaptive_avg_pool2d(x, (1, 1))

        x = x.view(x.size(0), -1)

        x = self.relu(self.fc1(x))
        output = self.relu(self.fc2(x))

        return output

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

```python
class ContrastiveLoss(torch.nn.Module):
    # TODO 6: implement the contrastive loss
    def __init__(self, margin):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(output1, output2, keepdim=True)

        # Similar pairs (label=1)
        similar_loss = (label) * euclidean_distance

        # Dissimilar pairs (label=0)
        not_similar_loss = (1 - label) * torch.clamp(self.margin - euclidean_distance, min=0.0)

        loss_contrastive = similar_loss + not_similar_loss

        loss_contrastive = torch.mean(loss_contrastive)
        return loss_contrastive
```

```python
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
        img1, img2, label = img1.to(device), img2.to(device), label.to(device)
        siamese_optimizer.zero_grad()
        output1, output2 = siamese_model(img1, img2)
        train_loss = siamese_criterion(output1, output2, label)

        # TODO 8: back propagate
        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_img1, val_img2, val_label = val_img1.to(device), val_img2.to(device), val_label.to(device)
        output1, output2 = siamese_model(val_img1, val_img2)
        val_loss = siamese_criterion(output1, output2, val_label)

        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_val_loss:.4f} - best min_val_loss = {min_val_loss:.4f} - lr = {siamese_optimizer.param_groups[0]["lr"]:.8f}')
    siamese_scheduler.step(current_val_loss)
```

```python
# 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
embeddings = []
identities = []

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

with torch.no_grad():
    for img, identity, _ in tqdm(FaceDataset(root_dir='large_prepared_data/test', transform=val_transform)):
        img = img.unsqueeze(0).to(device)

        output = siamese_model.extract_feature(img)

        embeddings.append(output.cpu())
        identities.append(identity)
embeddings = torch.cat(embeddings, dim=0)
identities = np.array(identities)
```

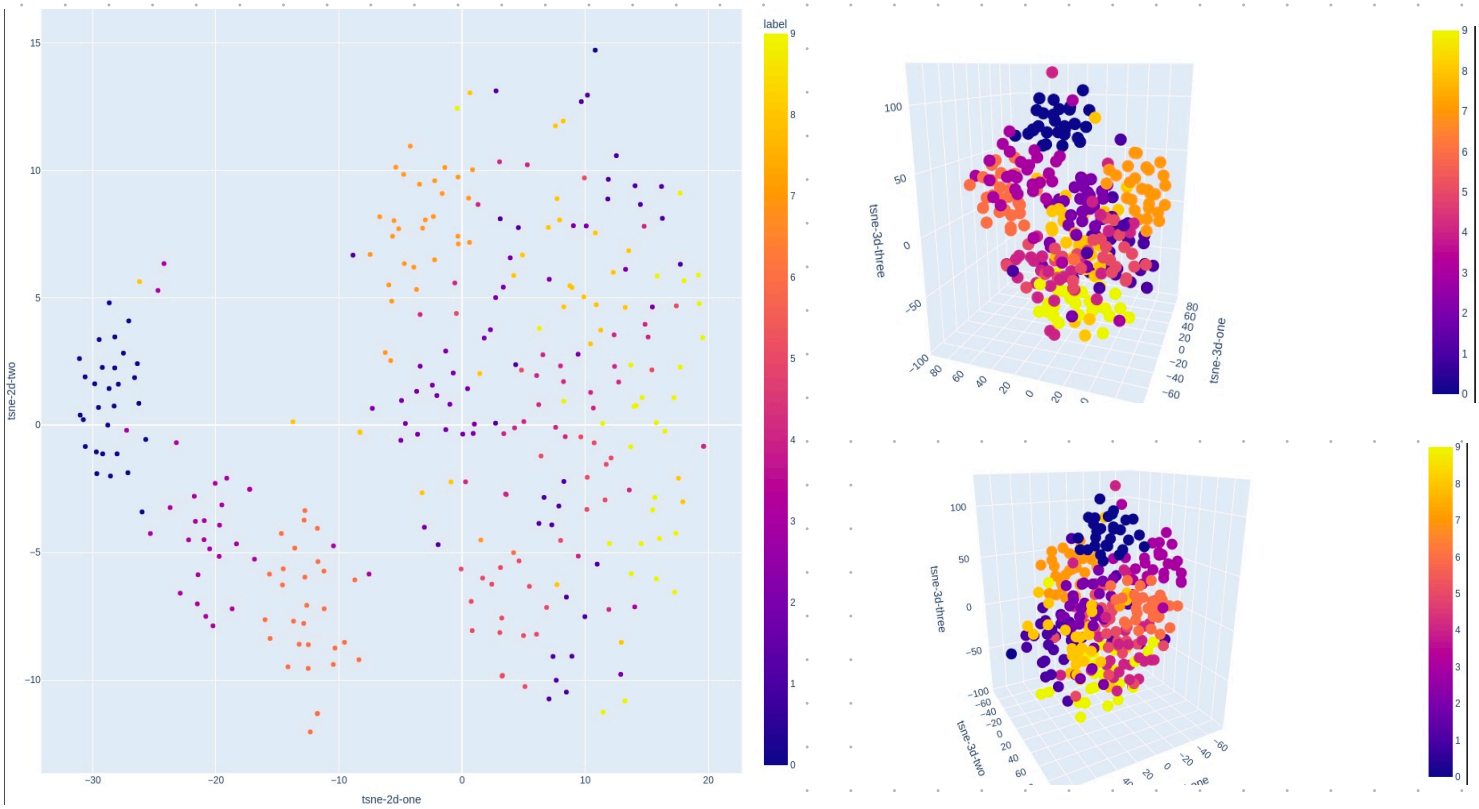


TODO 11: What could you say about the displayed visualization? Is the model working as expected?

Answer here:

The t-SNE plot suggests that the Siamese model has learned to group certain images together, as indicated by clusters of the same color, which implies similarity according to the model. However, there's noticeable overlap between different colors, meaning the model is not perfectly distinguishing all classes of images. Some classes are well separated, while others are mixed, which could be due to inherent similarities between certain classes, variability within classes, or a need for model refinement.

```python
# 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
embeddings = []
identities = []

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

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

embeddings = torch.cat(embeddings, dim=0).cpu()
identities = np.array(identities)
```



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

Answer here:

The triplet network shows less overlap between different colors—indicating distinct classes—than the Siamese network visualization. The triplet network's clusters are more separated and defined, suggesting that it's better at discriminating between different classes of images.

The improved performance of the triplet network could be due to its training objective, which explicitly learns to increase the distance (difference) between an anchor and a negative example while decreasing the distance between the anchor and a positive example. This can lead to more robust feature representations for classification or similarity assessment tasks.

Therefore, based on the visualizations, the triplet network appears to be performing better than the Siamese network.

```python
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 identity with the anchor image (no duplicated images in this list)
  - negative images (dim: [num_neg, num_feat]) is a list of image that has the different identity with the anchor image (no duplicated images in this list)

  [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 images
  pos_indices = self.label2indices[anchor_label]
  pos_indices = [i for i in pos_indices if i != idx]
  pos_samples = np.random.choice(pos_indices, self.num_pos, replace=False)
  pos_imgs = [self.images[i] for i in pos_samples]
  pos_labels = [self.labels[i] for i in pos_samples]

  # TODO 15: sample negative images that their identities differs from the identity of anchor image
  ## 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 images
  neg_labels = [label for label in self.label2indices.keys() if label != anchor_label]
  neg_indices = []
  while len(neg_indices) < self.num_neg:
    neg_label = np.random.choice(neg_labels)
    neg_list = self.label2indices[neg_label]
    neg_indices.extend(np.random.choice(neg_list, 1, replace=False))
  neg_imgs = [self.images[i] for i in neg_indices]
  neg_labels = [self.labels[i] for i in neg_indices]


  # TODO 16: utilize `self.transform' to convert anchor image, positive images, and negative images to tensors
  # WARNING!: Don't forget to convert to be tensors
  if self.transform:
    anchor_img = self.transform(anchor_img)
    pos_imgs = torch.stack([self.transform(img) for img in pos_imgs])
    neg_imgs = torch.stack([self.transform(img) for img in neg_imgs])

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

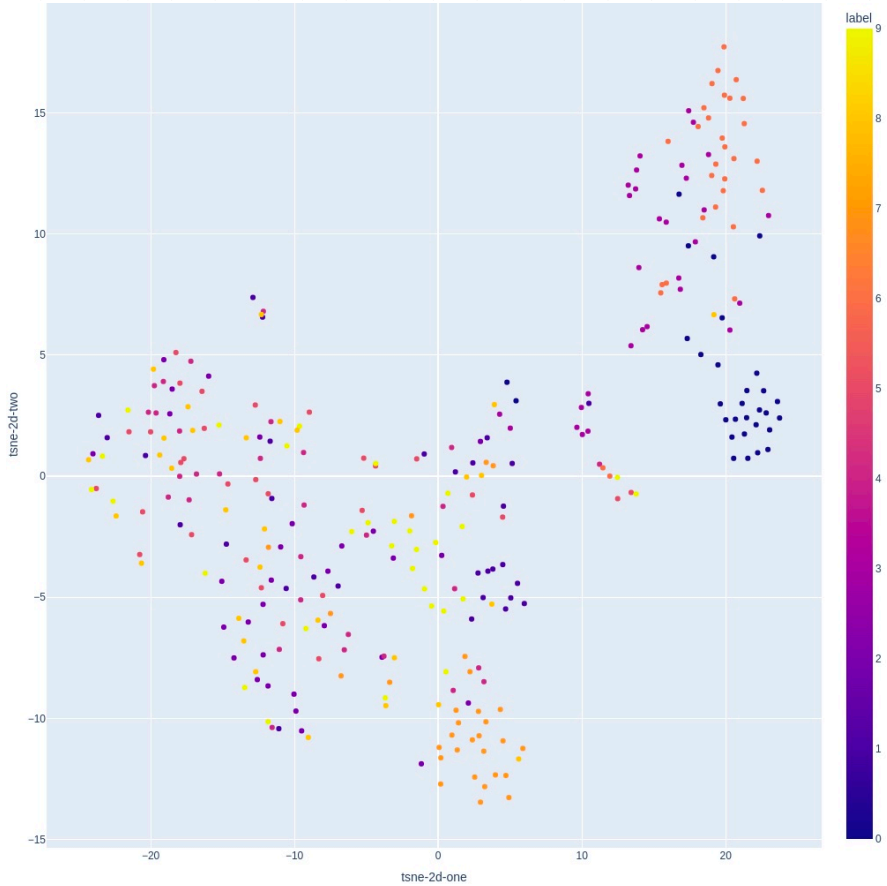
```python
# TODO 17: declare the datasets and the dataloaders
train_infonce_dataset = InfoNCEDataset(root_dir='large_prepared_data/train', transform=train_transform, num_pos=4, num_neg=8)
train_infonce_dataloader = DataLoader(train_infonce_dataset, batch_size=train_batch_size, shuffle=True, num_workers=2)

val_infonce_dataset = InfoNCEDataset(root_dir='large_prepared_data/val', transform=val_transform, num_pos=4, num_neg=8)
val_infonce_dataloader = DataLoader(val_infonce_dataset, batch_size=val_batch_size, shuffle=False, num_workers=2)

test_infonce_dataset = InfoNCEDataset(root_dir='large_prepared_data/test', transform=val_transform, num_pos=4, num_neg=8)
test_infonce_dataloader = DataLoader(test_infonce_dataset, batch_size=test_batch_size, shuffle=False, num_workers=2)
```

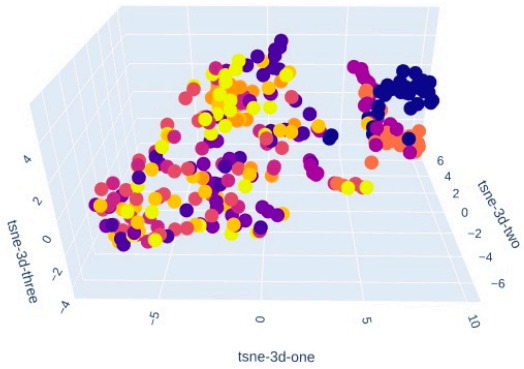
```python
class ImageEncoder(nn.Module):
  # TODO 18: implement the image encoder according to the above figure
  def __init__(self):
      super(ImageEncoder, self).__init__()

      resnet18 = torchvision.models.resnet18(pretrained=False)

      self.feature_extractor = nn.Sequential(*(list(resnet18.children())[:-1]))
      feature_vector_size = resnet18.fc.in_features

      self.fc1 = nn.Linear(feature_vector_size, 128)
      self.fc2 = nn.Linear(128, 64)
      self.relu = nn.ReLU()

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

      x = x.view(x.size(0), -1)

      x = self.relu(self.fc1(x))
      output = self.relu(self.fc2(x))
      return output

  def forward(self, input1, input2, input3):
      input2 = input2.view(-1, *input2.shape[2:])
      input3 = input3.view(-1, *input3.shape[2:])

      output1 = self.extract_feature(input1)
      output2 = self.extract_feature(input2)
      output3 = self.extract_feature(input3)

      output2 = output2.view(-1, 4, output2.size(1))
      output3 = output3.view(-1, 8, output3.size(1))

      return output1, output2, output3
```

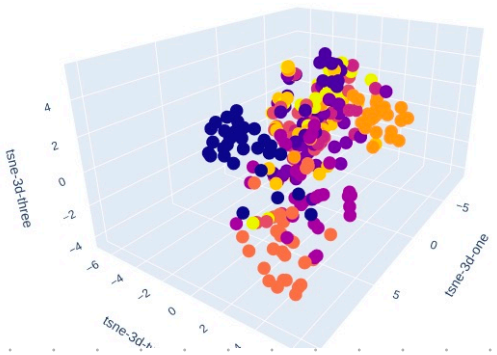
```python
class InfoNCELoss(nn.Module):
  # TODO 19: implement the InfoNCE loss
  def __init__(self, device, temperature):
    super(InfoNCELoss, self).__init__()
    self.device = device
    self.temperature = temperature
    self.criterion = nn.CrossEntropyLoss()

  def forward(self, anchor_feat, pos_feats, neg_feats):
    B, P = pos_feats.size(0), pos_feats.size(1)

    anchor_feat_exp = anchor_feat.unsqueeze(1).expand(-1, P, -1).contiguous().view(B * P, -1)

    pos_feats = pos_feats.contiguous().view(B * P, -1)
    pos_sim = F.cosine_similarity(anchor_feat_exp, pos_feats).unsqueeze(1) / self.temperature

    neg_feats = neg_feats.repeat(1, P, 1).view(B * P, -1, anchor_feat.size(1))
    anchor_feat_exp_neg = anchor_feat_exp.unsqueeze(1).expand(-1, neg_feats.size(1), -1)
    neg_sim = F.cosine_similarity(anchor_feat_exp_neg, neg_feats, dim=2) / self.temperature

    logits = torch.cat([pos_sim, neg_sim], dim=1)
    labels = torch.zeros(B * P, dtype=torch.long, device=anchor_feat.device)

    loss = self.criterion(logits, labels)

    return loss / P
```

```python
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(train_infonce_dataloader):

        # 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_imgs.to(device)
        anchor_feat, pos_feats, neg_feats = infonce_model(anchor_img, pos_imgs, neg_imgs)
        train_loss = train_infonce_criterion(anchor_feat, pos_feats, neg_feats)

        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()
        infonce_optimizer.step()

    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_imgs.to(device), val_neg_imgs.to(device)
      val_anchor_feat, val_pos_feats, val_neg_feats = infonce_model(val_anchor_img, val_pos_imgs, val_neg_imgs)
      val_loss = val_infonce_criterion(val_anchor_feat, val_pos_feats, val_neg_feats)

      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)
```

```python
# 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
embeddings = []
identities = []

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

with torch.no_grad():
    for img, identity, _ in tqdm(FaceDataset(root_dir='large_prepared_data/test', transform=val_transform)):
        img = img.unsqueeze(0).to(device)

        output = infonce_model.extract_feature(img)

        embeddings.append(output.cpu())
        identities.append(identity)
embeddings = torch.cat(embeddings, dim=0)
identities = np.array(identities)
```







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:

The CelebA dataset presents unique challenges for models like InfoNCE and triplet networks due to its nature:

- High Intra-Class Variation: CelebA contains various images of celebrities with diverse facial expressions, poses, lighting, and accessories, leading to significant intra-class variations.
- Subtle Inter-Class Differences: Different celebrities might look alike, making it challenging for models to distinguish between subtle inter-class differences.

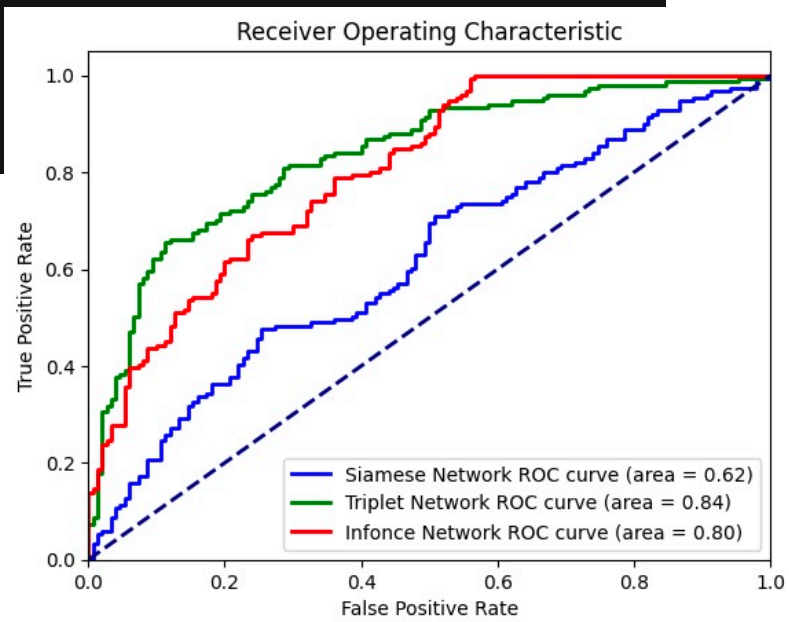
And Loss function of thos two model:

- Triplet Loss: Might excel because it directly optimizes the relative distance between matched and unmatched pairs, which could help when dealing with subtle inter-class differences and when learning from the high intra-class variation.
- InfoNCE Loss: Could be less effective if it's not capturing the nuanced differences as effectively as the triplet loss. Since InfoNCE averages the effect of multiple negatives, it may dilute the learning signal if those negatives include a mix of easy (very dissimilar) and hard (very similar) examples.

In CelebA, the distinction between different individuals (inter-class separation) and the variance in images of the same individual (intra-class compactness) are critical. The triplet approach, with its emphasis on margin optimization, might be better suited for handling this dataset's specific challenges.

```python
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), batch_label.to(device)
    with torch.no_grad():
        # TODO 26: extract features with both siamese network, triplet network, infonce network
        # and keep embeddings in provided lists according to variable names
        siamese_embeddings1.append(siamese_model.extract_feature(batch_img1))
        siamese_embeddings2.append(siamese_model.extract_feature(batch_img2))

        triplet_embeddings1.append(triplet_model.extract_feature(batch_img1))
        triplet_embeddings2.append(triplet_model.extract_feature(batch_img2))

        infonce_embeddings1.append(infonce_model.extract_feature(batch_img1))
        infonce_embeddings2.append(infonce_model.extract_feature(batch_img2))

    labels.append(batch_label)
```

```python
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, auc

siamese_scores_np = siamese_scores.cpu().detach().numpy()
triplet_scores_np = triplet_scores.cpu().detach().numpy()
infonce_scores_np = infonce_scores.cpu().detach().numpy()
labels_np = labels.cpu().detach().numpy()

# Calculate ROC curve and ROC area for Siamese network
fpr_siamese, tpr_siamese, _ = roc_curve(labels_np, siamese_scores_np)
roc_auc_siamese = auc(fpr_siamese, tpr_siamese)

# Calculate ROC curve and ROC area for Triplet network
fpr_triplet, tpr_triplet, _ = roc_curve(labels_np, triplet_scores_np)
roc_auc_triplet = auc(fpr_triplet, tpr_triplet)

# Calculate ROC curve and ROC area for Infonce network
fpr_infonce, tpr_infonce, _ = roc_curve(labels_np, infonce_scores_np)
roc_auc_infonce = auc(fpr_infonce, tpr_infonce)

# Plot of a ROC curve for a specific class
plt.figure()
plt.plot(fpr_siamese, tpr_siamese, color='blue', lw=2, label='Siamese Network ROC curve (area = %0.2f)' % roc_auc_siamese)
plt.plot(fpr_triplet, tpr_triplet, color='green', lw=2, label='Triplet Network ROC curve (area = %0.2f)' % roc_auc_triplet)
plt.plot(fpr_infonce, tpr_infonce, color='red', lw=2, label='Infonce Network ROC curve (area = %0.2f)' % roc_auc_infonce)

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```



- Siamese Network: The ROC curve for the Siamese network has an AUC of 0.62. It's underperforming compared to the other two, indicating it has a higher rate of false positives at most thresholds.
- Triplet Network: The ROC curve for the Triplet network shows an AUC of 0.84, which suggests it has a good balance between TPR and FPR and outperforms the Siamese network.
- InfoNCE Network: The InfoNCE network ROC curve has an AUC of 0.80. It performs better than the Siamese network and slightly worse than the Triplet network.

The Triplet network appears to be the best performing model according to the ROC curves, as it achieves the highest AUC value.

The performance differences are likely due to the inherent characteristics of the loss functions and how they encourage the models to learn discriminative features. The Triplet loss explicitly encourages the model to create a margin between the distances of the positive and negative examples to the anchor, which might be particularly effective in the context of the CelebA dataset with its fine-grained visual distinctions. The InfoNCE network also performs well, indicating its effectiveness in a supervised setting, but it might not be as fine-tuned for the specific task as the Triplet network. The Siamese network, while still useful, may not be enforcing as strict of a margin between classes as the Triplet loss, leading to its lower AUC.