

به نام خدا



دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده برق و کامپیوتر



درس تحلیل و طراحی شبکه‌های عصبی عمیق

تمرین شماره ۵

نام و نام خانوادگی : علیرضا حسینی – کیانا هوشانفر

شماره دانشجویی : ۸۱۰۱۰۱۱۴۲ – ۸۱۰۱۰۱۳۶۱

بهمن ماه ۱۴۰۲

۴	مقدمه
۶	آماده سازی مجموعه دادگان
۶	آماده سازی دیتاست triplet
۶	- درست کردن فایل csv
۷	- بررسی صحت دیتاست آماده شده
۸	- تقسیم بندی داده ها به آموزش و ارزیابی
۸	- پیش پردازش داده ها
۹	- دیتالودرها
۱۰	آماده سازی دیتاست contrastive
۱۰	- درست کردن فایل csv
۱۳	تعریف مدل efficientnet_b0
۱۴	Triplet Loss
۱۴	تعریف
۱۵	پیاده سازی triplet loss
۱۶	آموزش مدل با TRIPLET LOSS
۱۹	لود کردن مدل آموزش دیده
۱۹	پیدا کردن top 10
۲۲	محاسبه ROC
۲۶	CONTRASTIVE Loss
۲۶	تعریف
۲۷	پیاده سازی CONTRASTIVE LOSS
۲۸	آموزش مدل با CONTRASTIVE
۳۰	پیدا کردن top 10
۳۱	محاسبه ROC
۳۲	Sensitivity over margin برای ۲ بخش قبل
۴۰	Fisher Discriminant Contrastive Loss
۴۰	- تعریف
۴۰	- پیاده سازی FDC

۴۲	- دیتالودر
۴۳	- آموزش مدل
۴۵	- منحنی ROC برای FDC
۴۶	- FDC برای top 10
۴۷	Fisher Discriminant Triplet Loss
۴۷	- تعریف
۴۷	- پیاده سازی FDT
۴۸	- دیتالودر
۴۸	- حلقه آموزش
۵۱	- منحنی ROC برای FDT
۵۲	- FDT برای TOP 10
۵۳	- آنالیز و نتیجه گیری

شبکه‌های سیامی شبکه‌های کارآمدی در استخراج ویژگی‌ها و یادگیری متریک هستند. در پیاده‌سازی شبکه‌های سیامی از توابع هزینه‌ای متفاوتی مانند تریپلت و کانتراستیو استفاده می‌شود. توابع هزینه متریک زمانی به کار می‌روند که هدف اصلی، یادگیری یک تعبیر فضایی مناسب است که اطلاعات شباهت و تفاوت بین نمونه‌ها را به خوبی حفظ کند. ایده اصلی این توابع هزینه آن است که جفت‌های نمونه‌هایی که از یک کلاس هستند را به هم نزدیک کرده و جفت‌های نمونه‌هایی که از دو کلاس مختلف هستند را از یکدیگر دور نگه دارند. این توابع هزینه معمولاً در وظایفی که شباهت یا تفاوت بین جفت نمونه‌ها اهمیت دارد، مورد استفاده قرار می‌گیرند.

با توجه به اینکه بخش‌های بسیاری مانند لود کردن مدل و خروجی گرفتن top10 و رسم منحنی ROC و لوپ ترین و لود کردن دادها برای تمامی موارد تقریباً یکسان می‌باشد و تنها در ۴ بخش مختلف class های عوض شده است در این گزارش جهت جلوگیری از توضیح اضافات تمامی کدهای بخش‌های مختلف در بخش triplet loss آمده است و در سایر بخش‌ها تنها نتایج آمده است (در ضمن کدهای تمامی بخش‌ها به پیوست ارسال شده است).

سیستم مورد استفاده برای حل این سوال Cpu core i9 با 64G رم و ۲ واحد پردازش گرافیکی ۳۰۹۰ با حافظه 24G می‌باشد.

1	[0.7%]	7	[0.0%]	13	[0.0%]	19	[0.0%]
2	[0.0%]	8	[0.7%]	14	[0.7%]	20	[0.0%]
3	[0.0%]	9	[0.0%]	15	[0.7%]	21	[0.0%]
4	[0.0%]	10	[0.0%]	16	[2.0%]	22	[0.0%]
5	[0.0%]	11	[0.7%]	17	[0.7%]	23	[0.0%]
6	[0.7%]	12	[0.0%]	18	[0.0%]	24	[0.7%]
Mem[] 4.81G/62.6G						Tasks: 101, 863 thr; 1 running					
Swp[] 5.85G/8.00G						Load average: 1.71 2.26 2.58					
						Uptime: 15 days, 06:33:10					

NVIDIA-SMI 535.104.12				Driver Version: 535.104.12				CUDA Version: 12.2			

GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC				
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	MIG M.				
=====											
0	NVIDIA GeForce RTX 3090	Off	00000000:01:00.0	Off			N/A				
39%	57C	P8	8W / 370W	306MiB / 24576MiB	0%	Default	N/A				

1	NVIDIA GeForce RTX 3090	Off	00000000:03:00.0	Off			N/A				
36%	45C	P8	9W / 370W	6336MiB / 24576MiB	0%	Default	N/A				

Processes:											
GPU	GI	CI	PID	Type	Process name	GPU Memory Usage					
	ID	ID									
=====											
0	N/A	N/A	1029142	C	...t/miniconda3/envs/torch2/bin/python	298MiB					
1	N/A	N/A	1029142	C	...t/miniconda3/envs/torch2/bin/python	6328MiB					

شکل 1 : مشخصات سیستم استفاده شده

آماده سازی مجموعه دادگان

آماده سازی دیتاست TRIPLET

- درست کردن فایل CSV

ابتدا باید دیتاست داده شده را با مقادیر مثبت و منفی و anchor ها در یک فایل CSV بریزیم.

برای اینکار از کد زیر استفاده میکنیم.

در این کد با توجه به نام فایل موجود در دایرکتوری دیتاست کلاس آن را پیدا کرده و در ادامه دیتا ها را به صورت anchor و positive و negative در یک فایل CSV ذخیره میکنیم.

```
# Path to your dataset folder
dataset_folder = 'Homework5_dataset/'

# List all image files in the dataset folder
image_files = [f for f in os.listdir(dataset_folder) if f.endswith('.jpg')]

# Create a DataFrame to store file paths and corresponding classes
df = pd.DataFrame({'File': image_files})

# Extract classes from the file names
df['Class'] = df['File'].apply(lambda x: x.split('_')[1])

# Create a dictionary to map classes to their corresponding image files
class_to_files = df.groupby('Class')['File'].apply(list).to_dict()

# Generate triplets from the dataset
def generate_triplets(class_to_files):
    triplets = []
    for c in class_to_files.keys():
        # Create all possible combinations of images for each class
        combinations_list = list(combinations(class_to_files[c], 3))
        triplets.extend(combinations_list)
    return triplets

# Generate triplets from the dataset
triplets = generate_triplets(class_to_files)

# Save all triplets to a single CSV file
columns = ['Anchor', 'Negative', 'Positive']
all_triplets_df = pd.DataFrame(triplets, columns=columns)
```

```
csv_path = 'triplets.csv'
all_triplets_df.to_csv(csv_path, index=False)
```

این اسکریپت از کتابخانه های OS, Pandas و scikit-learn برای پردازش مجموعه داده ای از تصاویر برای آموزش مبتنی بر triplet loss در زمینه یادگیری ماشین استفاده می کند. با فهرست کردن همه فایل های jpg. در یک پوشه داده مشخص شروع می شود و یک Pandas DataFrame برای ذخیره مسیرهای فایل و کلاس های مربوطه تجزیه شده از نام فایل ها ایجاد می کند. سپس اسکریپت با تشکیل تمام ترکیبات ممکن از سه تصویر برای هر کلاس، triplets را تولید می کند. triplet حاصل، متشکل از تصاویر anchor، مثبت و منفی، در یک DataFrame جدید ذخیره می شوند و به یک فایل CSV به نام triplets.csv صادر می شوند. این داده های سه گانه در triplet loss استفاده می شوند، جایی که یک مدل آموزش داده می شود تا فاصله بین تصاویر مشابه (anchor و مثبت) را به حداقل برساند و در عین حال فاصله بین تصاویر غیر مشابه (anchor-negative) را به حداکثر برساند. علاوه بر این، اسکریپت کتابخانه PIL را برای پردازش تصویر وارد می کند و از تابع train_test_split از scikit-learn برای استفاده در پارتیشن بندی داده ها استفاده می کند.

- بررسی صحت دیتاست آماده شده

در ادامه میتوان به کمک کتاب خانه pandas دیتاست را لود کرد.

```
df = pd.read_csv('dataset_train.csv')
df.head(n=5)
```

به عنوان مثال برای یکی از row میتوان به کمک کد زیر داده ها را visual کرد.

```
row = df.iloc[12]

# Read images using PIL
A_img = Image.open(image_dir + row['Anchor'])
P_img = Image.open(image_dir + row['Positive'])
N_img = Image.open(image_dir + row['Negative'])

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize = (5,5))

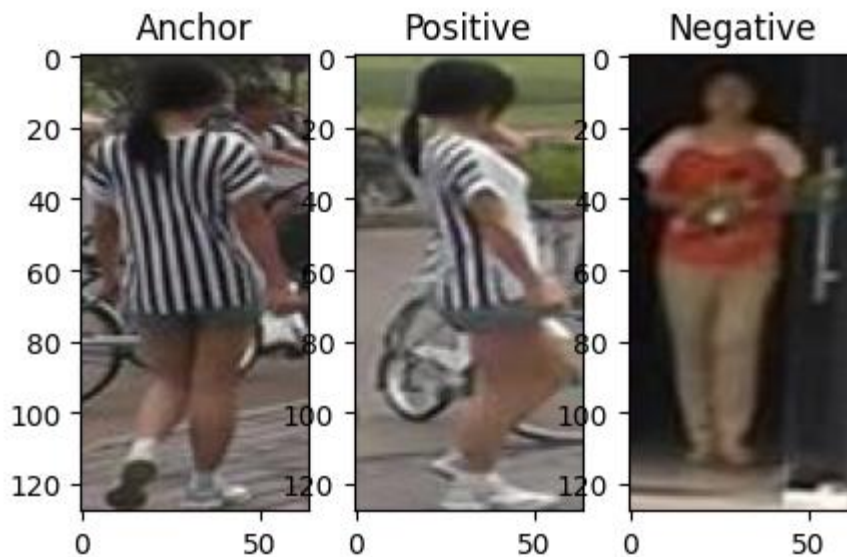
ax1.set_title("Anchor")
ax1.imshow(A_img)

ax2.set_title("Positive")
```

```
ax2.imshow(P_img)
```

```
ax3.set_title("Negative")
```

```
ax3.imshow(N_img)
```



شکل ۲: نمونه داده های دیتاست آماده شده **anchor** و **negative** و **positive**

- تقسیم بندی داده ها به آموزش و ارزیابی

در پایان نیز میتوان به کمک کد زیر داده های دیتاست را به ۲ دسته آموزش و ارزیابی تقسیم بندی کرد.

```
train, valid = train_test_split(df, test_size = 0.20, random_state = 42)
```

- پیش پردازش داده ها

جهت پیش پردازش داده ها نیاز است که تمامی داده ها نرمال شده و به ابعاد ۲۲۴ در ۲۲۴ نیز **resize** شوند که این کار به کمک کد زیر انجام میشود.

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((224, 224)),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

```
def load_image(file_path):
    img = Image.open(data_dir + file_path).convert('RGB')
```



```

img_tensor = transform(img)
return img_tensor

def load_triplet(row):
    anchor_img = load_image(row.Anchor)
    positive_img = load_image(row.Positive)
    negative_img = load_image(row.Negative)

    return anchor_img, positive_img, negative_img

train_triplets = [load_triplet(row) for _, row in train.iterrows()]
valid_triplets = [load_triplet(row) for _, row in valid.iterrows()]

```

پس از نرمال سازی برای تست میتوان به صورت زیر عمل کرد.

```

anchor, positive, negative = train_triplets[20]

f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(5, 5))

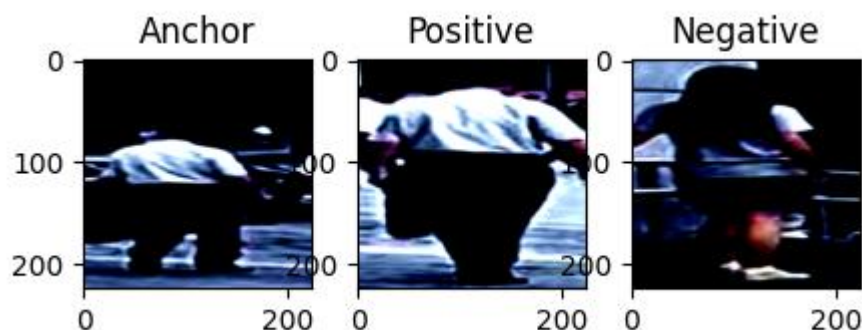
ax1.set_title('Anchor')
ax1.imshow(anchor.numpy().transpose((1, 2, 0)), cmap='gray')

ax2.set_title('Positive')
ax2.imshow(positive.numpy().transpose((1, 2, 0)), cmap='gray')

ax3.set_title('Negative')
ax3.imshow(negative.numpy().transpose((1, 2, 0)), cmap='gray')

```

که خروجی آن به صورت زیر میباشد.



شکل ۳: نمونه خروجی داده های triplet پس از پیش پردازش

- دیتالودرها

```

train_loader = DataLoader(train_triplets, batch_size = 32, shuffle = True)

```

```
valid_loader = DataLoader(valid_triplets, batch_size = 32)
```

به کمک کد فوق دیتالودر ها را تعریف میکنیم.

آماده سازی دیتاست CONTRASTIVE

- درست کردن فایل CSV

همانطور که در فرمول **contrastive loss** خواهید دید مشخص می باشد در اینجا دیتالودر نباید ۳ گانه باشد بلکه باید بدین صورت باشد که جفت های مثبت لیبل ۱ داشته و باشند و جفت های منفی نیز لیبل صفر داشته باشند برای اینکار از کد زیر استفاده میشود.

دیتاست از پیش آماده شده قبلی را میخوانیم و به **anchor** و **positive** لیبل مثبت میدهیم و به **anchor** و **negative** نیز لیبل منفی میدهیم.

```
# Load the dataset from the original CSV file
original_df = pd.read_csv('dataset_train.csv')

# Randomly select 2000 rows from the anchor and positive columns where label is 1
positive_samples = original_df.sample(n=2000)[['Anchor', 'Positive']]
positive_samples['label'] = 1

# Exclude the selected positive samples from the original dataset
remaining_df = original_df.drop(positive_samples.index)

# Randomly select 2000 rows from the remaining anchor and negative columns where
label is 0
negative_samples = remaining_df.sample(n=2000)[['Anchor', 'Negative']]
negative_samples['label'] = 0

# Merge the 'Negative' and 'Positive' columns into a new column 'Merged'
positive_samples['Merged'] = positive_samples['Positive']
negative_samples['Merged'] = negative_samples['Negative']

# Concatenate the positive and negative samples into a new DataFrame
final_df = pd.concat([positive_samples[['Anchor', 'Merged', 'label']],
negative_samples[['Anchor', 'Merged', 'label']]])

# Save the final DataFrame to a new CSV file
final_df.to_csv('contrastive_dataset.csv', index=False)

# Rename columns
final_df.columns = ['Positive', 'Negative', 'Label']
```

```
# Save to CSV
final_df.to_csv('contrastive_dataset.csv', index=False)
```

این کد به طور تصادفی جفت‌هایی از تصاویر را از داده‌های ورودی برای آموزش انتخاب می‌کند، که هر جفت شامل یک مثال مثبت (تصویر با برچسب مشابه تصویر فعلی) و یک مثال منفی (تصویر با برچسب متفاوت) است. نسبت به تصویر فعلی، برای هر ردیف، کلاس مجموعه داده دو تصویر را برمی‌گرداند، برچسبی که نشان می‌دهد نمونه‌های مثبت یا منفی هستند (این کار به صورت تصادفی انجام می‌شود)، و برچسب تصویر اصلی. تصویر دوم به صورت تصادفی انتخاب می‌شود و اینکه تصویر دوم هم کلاس باشد یا نباشد نیز تصادفی با ۵۰-۵۰ شانس است. این تضمین می‌کند که مجموعه داده ۵۰٪ جفت مثبت و ۵۰٪ جفت منفی را برمی‌گرداند.

در زمینه از **contrastive loss** و شبکه‌های سیامی که برای کارهایی مانند شباهت تصویر یا تأیید استفاده می‌شوند، توزیع ۵۰-۵۰ جفت‌های مثبت و منفی اغلب برای متعادل کردن فرآیند آموزش انتخاب می‌شود. هدف این است که از تعصب مدل نسبت به نمونه‌های مثبت یا منفی جلوگیری شود و از عملکرد قوی‌تر و کلی‌تر اطمینان حاصل شود.

۱. آموزش متعادل: با داشتن تعداد مساوی جفت مثبت و منفی، مدل در طول آموزش در معرض مجموعه‌ای متعادل از مثال‌ها قرار می‌گیرد. این از تمایل مدل بیشتر به یک کلاس نسبت به کلاس دیگر جلوگیری می‌کند، که اگر یک کلاس بر مجموعه آموزشی تسلط داشته باشد، ممکن است اتفاق بیفتد.

۲. اجتناب از سوگیری‌ها: اگر مجموعه داده نامتعادل باشد (یعنی نمونه‌های بسیار بیشتری از یک کلاس نسبت به کلاس دیگر وجود دارد)، مدل ممکن است یاد بگیرد که برای کلاس اکثریت بهینه شود و کلاس اقلیت را نادیده بگیرد. این می‌تواند منجر به پیش‌بینی‌های مغرضانه و کاهش عملکرد در طبقه اقلیت شود.

۳. تقویت قدرت تمایز: در زمینه این لاس، هدف مدل یادگیری تعبیه‌هایی است که در آن نمونه‌های مشابه در فضای تعبیه نزدیک هستند و نمونه‌های متفاوت از هم دور هستند. توزیع متعادل تضمین می‌کند که مدل صرفاً بر یادگیری جدا کردن یک نوع جفت و نادیده گرفتن نوع دیگر تمرکز نمی‌کند.

۴. پویایی train پایدار: داشتن توزیع مساوی از جفت های مثبت و منفی به پویایی train با ثبات تر کمک می کند. از همگرایی خیلی سریع مدل یا گیر افتادن در راه حل های نابهینه جلوگیری می کند.

Training set label distribution:

Label

1 1600

0 1600

Name: count, dtype: int64

Validation set label distribution:

Label

1 400

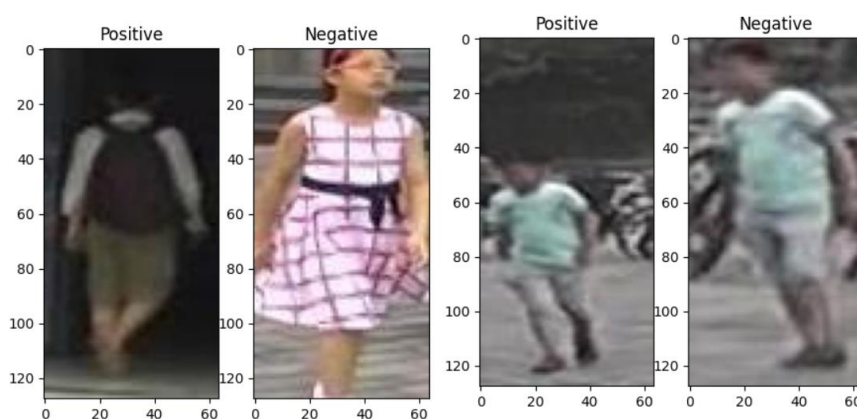
0 400

Name: count, dtype: int64

همچنین باید چک کنیم که کلاس ها متوازن انتخاب شده باشند که در اینجا میبینیم که کلاس ها متوازن هستند.

در ادامه نیز همانند قبل (کد ها مشابه قبل میباشد) داده ها را تقسیم و نرمال و ۲۲۴ در ۲۲۴ میکنیم.

در ادامه نمونه ای از دیتاست را مشاهده میکنید:



شکل ۴ - نمونه ای از دیتاست بخش دوم

تعریف مدل *EFFICIENTNET_B0*

با توجه به صورت سوال مطرح شده در اینجا از یک `efficientnet_b0` استفاده میشود که همانطور که میدانید در شبکه های سایامی هدف مقایسه نهایی خروجی ها در فضای `embedding` میباشد بنابراین نیازی به بخش طبقه بند نمیباشد.

مدل به کمک فریم وورک پایتورچ به صورت زیر تعریف میشود.

در نهایت نیز مدل را لود کرده و در GPU لود میکنیم.

```
# Model Definition
class TripModel(nn.Module):
    def __init__(self, emb_size=512):
        super(TripModel, self).__init__()
        self.efficientnet = models.efficientnet_b0(pretrained=True)
        in_features = self.efficientnet.classifier[-1].in_features
        self.efficientnet.classifier[-1] = nn.Linear(in_features, emb_size)

    def forward(self, images):
        embeddings = self.efficientnet(images)
        return embeddings

model = TripModel()
model.to("cuda:1")
print("Model Loaded")
```

کد ارائه شده یک کلاس `TripModel` را به عنوان زیر کلاس `nn.Module` PyTorch تعریف می کند. `TripModel` از معماری `EfficientNet-B0`، یک شبکه عصبی کانولوشنال از پیش آموزش دیده برای طبقه بندی تصاویر استفاده می کند. با اندازه `embedding` ۵۱۲ مقداردهی اولیه می شود و آخرین لایه کاملاً متصل `EfficientNet` را با یک لایه خطی جدید با اندازه تعبیه شده جایگزین می کند. روش فوروارد تصاویر ورودی را می گیرد و جاسازی ها را با استفاده از `EfficientNet` اصلاح شده محاسبه می کند. سپس مدل نمونه سازی می شود، به GPU منتقل می شود.

یک مفهوم اساسی در زمینه یادگیری ماشین و شبکه های عصبی عمیق است، به ویژه در وظایف مربوط به تشخیص چهره، بازیابی تصویر و یادگیری شباهت. این تابع $loss$ با هدف افزایش تعبیه ویژگی های یک مدل با حصول اطمینان از اینکه فاصله بین نمونه های $anchor$ و نمونه های مثبت به حداقل می رسد در حالی که فاصله بین نمونه های $anchor$ و نمونه های منفی را در فضای ویژگی به حداکثر می رساند، افزایش می دهد. به عبارت دیگر، مدل را تشویق می کند تا آیتم های مشابه (مثلاً چهره های یک شخص) را در فضای $embed$ به هم نزدیک کند، در حالی که موارد غیرمشابه را از هم دورتر می کند.

فرمول آن به صورت زیر می باشد.

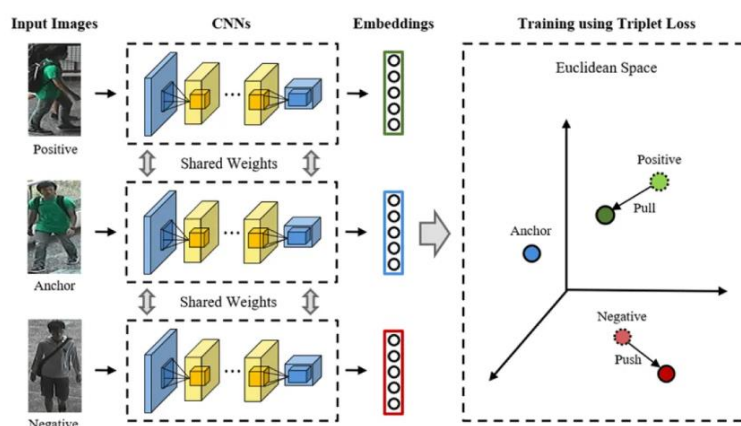
$$L(A, P, N) = \max(d(A, P) - d(A, N) + \text{margin}, 0)$$

- $L(A, P, N)$ نشان دهنده triplet loss برای یک نمونه $anchor$ (A)، یک نمونه مثبت (P) و یک نمونه منفی (N) است.

- $d(A, P)$ نشان دهنده فاصله اقلیدسی (یا هر متریک فاصله مناسب) بین $anchor$ و نمونه های مثبت در فضای $embedding$ است.

- $d(A, N)$ نشان دهنده فاصله اقلیدسی بین $anchor$ و نمونه های منفی در فضای $embedding$ است.

- " $margin$ " یک هاپرپارامتر است که حداقل مارجین مورد نظر را بین فواصل جفت $anchor$ مثبت و جفت $anchor$ منفی تعریف می کند. اگر $margin$ برآورده نشود، $loss$ صفر می شود. در غیر این صورت، مدل را تشویق می کند که نمونه مثبت را به $anchor$ نزدیکتر کند و نمونه منفی را دورتر کند.



شکل ۵ - ساز و کار triplet loss

پایاده سازی TRIPLET LOSS

برای پیاده سازی triplet loss از تابع زیر استفاده میشود.

```
class CustomTripletMarginLoss(nn.Module):
    def __init__(self, margin=1.0):
        super(CustomTripletMarginLoss, self).__init__()
        self.margin = margin

    def euclidean_distance(self, x1, x2):
        return torch.sqrt(torch.sum((x1 - x2) ** 2, dim=1))

    def forward(self, anchor, positive, negative):
        distance_positive = self.euclidean_distance(anchor, positive)
        distance_negative = self.euclidean_distance(anchor, negative)

        loss = torch.clamp(distance_positive - distance_negative + self.margin,
                             min=0.0).mean()

        return loss
```

کلاس CustomTripletMarginLoss یک triplet margin loss را برای آموزش شبکه های سه گانه تعریف می کند. تلفات بر اساس فواصل اقلیدسی بین نمونه های anchor، مثبت و منفی در فضای تعبیه شده محاسبه می شود. روش فوروارد از سه تانسور ورودی استفاده می کند که نشان دهنده جاسازی نمونه های anchor، مثبت و منفی است. فواصل اقلیدسی بین anchor و مثبت و همچنین نمونه های anchor و منفی را محاسبه می کند. سپس تلفات به عنوان تلفات محاسبه می شود و حاشیه ای را برای

اعمال حداقل تفکیک بین فواصل جفت‌های مثبت و منفی وارد می‌کند. اگر تفاوت بین فاصله جفت‌های مثبت و منفی از حاشیه مشخص شده بیشتر شود، لاس روی صفر تنظیم می‌شود. در غیر این صورت، این مقدار حاشیه بیش از حد است. ضرر **loss** نهایی میانگین این تلفات محاسبه شده در سراسر دسته است. این تنظیمات مدل را تشویق می‌کند تا جاسازی‌هایی را یاد بگیرد که جفت‌های مثبت حداقل با حاشیه مشخص شده به هم نزدیک‌تر از جفت‌های منفی باشند.

در نهایت نیز میتوان برای **triplet loss** یک **criterion** به صورت زیر تشکیل داد.

```
criterion = CustomTripletMarginLoss(margin=1.0)
```

ناگفته نماند که خود پایتورچ نیز متودی برای **triplet loss** دارد.

آموزش مدل با TRIPLET LOSS

به کمک کد زیر یک حلقه آموزش برای مدل نوشته و مدل را با **TRIPLET LOSS** آموزش میدهد.

در این حلقه مدل بر اساس بهترین مدل براساس کمترین **Loss** بر روی داده های ارزیابی ذخیره میشود.

در این حلقه از بهینه ساز و **lr** و **scheduler** زیر استفاده شده است.

```
optimizer = torch.optim.Adam(model.parameters(), lr= 0.001)
scheduler = StepLR(optimizer, step_size=2, gamma=0.1)
```

کد حلقه آموزش به صورت زیر میباشد.

```
device = "cuda:1" if torch.cuda.is_available() else "cpu"
model.to(device)
best_valid_loss = np.Inf

train_losses = []
valid_losses = []

for epoch in range(20):
    # Training
    model.train()
    total_train_loss = 0.0

    for A, P, N in tqdm(train_loader):
        A, P, N = A.to(device), P.to(device), N.to(device)

        A_embs = model(A)
        P_embs = model(P)
```



```

N_embs = model(N)

loss = criterion(A_embs, P_embs, N_embs)

optimizer.zero_grad()
loss.backward()
optimizer.step()

total_train_loss += loss.item()

train_loss = total_train_loss / len(train_loader)
train_losses.append(train_loss)

# Validation
model.eval()
total_valid_loss = 0.0

with torch.no_grad():
    for A, P, N in tqdm(valid_loader):
        A, P, N = A.to(device), P.to(device), N.to(device)

        A_embs = model(A)
        P_embs = model(P)
        N_embs = model(N)

        loss = criterion(A_embs, P_embs, N_embs)

        total_valid_loss += loss.item()

valid_loss = total_valid_loss / len(valid_loader)
valid_losses.append(valid_loss)

# Check if validation loss improved and save the model
if valid_loss < best_valid_loss:
    torch.save(model.state_dict(), "best_model_triplet.pt")
    best_valid_loss = valid_loss
    print("SAVED_WEIGHT_SUCCESS")

print(f"EPOCHS: {epoch+1} train_loss: {train_loss} valid_loss: {valid_loss}")

```

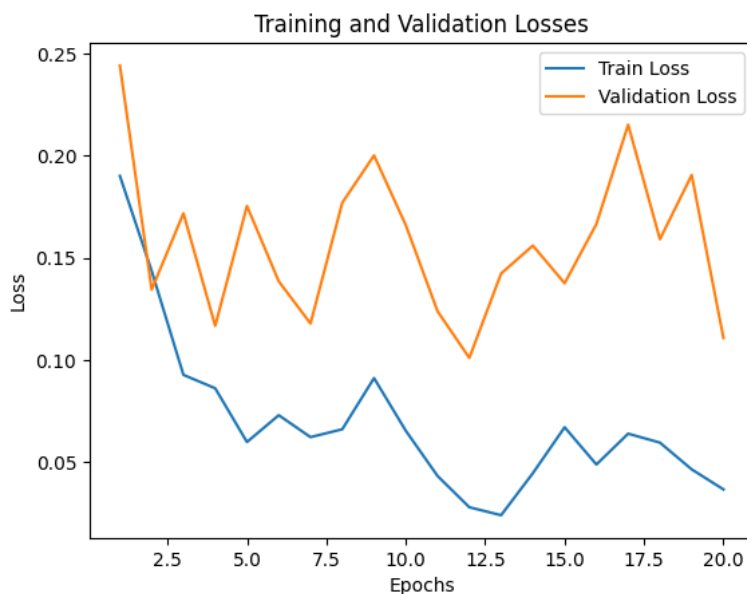
این کد یک شبکه triplet را برای تعداد مشخصی از دوره ها (۲۰) با استفاده از مجموعه داده های آموزشی و اعتبار سنجی آموزش می دهد. در داخل حلقه آموزشی، برای هر دسته از بارگذار آموزشی، مدل تعبیه‌هایی

را برای نمونه‌های **A anchor** ، مثبت **P** و منفی **N** ایجاد می‌کند. سپس **loss** بر اساس این تعبیه‌ها محاسبه می‌شود و پارامترهای مدل با استفاده از **backpropagation** و نزول گرادیان به‌روزرسانی می‌شوند. لاس **Train** به عنوان میانگین تلفات در تمام دسته‌ها محاسبه می‌شود. به طور مشابه، حلقه اعتبارسنجی مدل را روی مجموعه داده اعتبارسنجی ارزیابی می‌کند و **loss** اعتبارسنجی را محاسبه می‌کند. اگر افت اعتبار در مقایسه با بهترین مواردی که تاکنون دیده شده است بهبود یابد، وضعیت مدل ذخیره می‌شود. پیشرفت، از جمله شماره دوره، از دست دادن آموزش، و از دست دادن اعتبار، پس از هر دوره چاپ می‌شود. این اسکریپت به عنوان یک حلقه آموزشی پایه برای یک شبکه سه گانه عمل می‌کند، با هدف یادگیری جاسازی‌هایی که فاصله بین جفت‌های مثبت را به حداقل می‌رساند و فاصله بین جفت‌های منفی را به حداکثر می‌رساند.

در ادامه به کمک کد زیر منحنی‌های **loss** برای داده‌های آموزش و ارزیابی ذخیره می‌شود.

```
# Plotting
plt.plot(range(1, 21), train_losses, label='Train Loss')
plt.plot(range(1, 21), valid_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Losses')
plt.legend()
plt.show()
```

منحنی به صورت زیر می‌باشد:



شکل ۶: منحنی آموزش و ارزیابی پس از آموزش مدل با **triplet loss**

همانطور که مشاهده میشود **loss** به خوبی کاهش یافته است البته نوساناتی در **val** وجود دارد که با تغییر **lr** یا اضافه کردن **weight decay** و .. تا حدی میتوان آن را کنترل کرد که با توجه به اینکه خروجی در این حالت مناسب و دقت **ROC** بالا بود دیگر آن را خیلی تغییر ندادیم.

لود کردن مدل آموزش دیده

به کمک کد زیر مدل را لود میکنیم.

```
model.load_state_dict(torch.load("best_model_triplet2.pt"))
print("Model Loaded")
```

پیدا کردن TOP 10

پس از لود کردن مدل در اینجا ابتدا یک تابع جهت محاسبه فاصله مینویسیم.

```
def euclidean_dist(img_enc, anc_enc_arr):
    dist = np.sqrt(np.dot(img_enc-anc_enc_arr, (img_enc - anc_enc_arr).T))
    return dist
```

در ادامه به کمک کد زیر داده به مدل داده و فاصله ها را محاسبه کرده و ۱۰ تای اول آن ها را به کمک کتابخانه **network** رسم میکنیم. (ترتیب به صورت پاد ساعت گرد میباشد).

به کمک کد زیر این کار را انجام میدهیم.

```
def process_image(idx, df_enc, model, data_dir):
    img_name = df_enc["Anchor"].iloc[idx]
    img_path = data_dir + img_name

    img = io.imread(img_path)
    img = torch.from_numpy(img).permute(2, 0, 1) / 255.0

    model.eval()
    with torch.no_grad():
        img = img.to("cuda:0")
        img_enc = model(img.unsqueeze(0))
        img_enc = img_enc.detach().cpu().numpy()

    anc_enc_arr = df_enc.iloc[:, 1:].to_numpy()
    anc_img_names = df_enc["Anchor"]

    distance = []

    for i in range(anc_enc_arr.shape[0]):
        dist = euclidean_dist(img_enc, anc_enc_arr[i : i+1, :])
```

```

        distance = np.append(distance, dist)

    closest_idx = np.argsort(distance)

    return anc_img_names, data_dir, img, img_path, closest_idx, distance

def load_image(image_path):
    # Use PIL to open the image
    image = Image.open(image_path)
    return image

def create_graph(anc_img_names, DATA_DIR, img_path, closest_idx, distance,
no_of_closest=10):
    G = nx.Graph()

    S_name = [img_path.split('/')[-1]]

    for s in range(no_of_closest):
        S_name.append(anc_img_names.iloc[closest_idx[s]])

    for i, img_name in enumerate(S_name):
        image = load_image(DATA_DIR + img_name)
        G.add_node(i, image=image)

    for j in range(1, no_of_closest + 1):
        G.add_edge(0, j, weight=distance[closest_idx[j - 1]])

    return G, S_name

def plot_graph(G, S_name):
    pos = nx.kamada_kawai_layout(G)

    fig, ax = plt.subplots(figsize=(20, 20))
    ax.set_aspect('equal')
    nx.draw_networkx_edges(G, pos, ax=ax)

    plt.xlim(-1.5, 1.5)
    plt.ylim(-1.5, 1.5)

    trans = ax.transData.transform
    trans2 = fig.transFigure.inverted().transform

    piesize = 0.1 # this is the image size
    p2 = piesize / 2.0
    for n in G:
        xx, yy = trans(pos[n]) # figure coordinates

```

```

    xa, ya = trans2((xx, yy)) # axes coordinates
    a = plt.axes([xa - p2, ya - p2, piesize, piesize])
    a.set_aspect('equal')
    a.imshow(G.nodes[n]['image'])
    a.set_title(S_name[n][0:4])
    a.axis('off')

    ax.axis('off')
    plt.show()
def plot_closest_imgs(anc_img_names, DATA_DIR, img_path, closest_idx, distance,
no_of_closest=10):
    G, S_name = create_graph(anc_img_names, DATA_DIR, img_path, closest_idx,
distance, no_of_closest)
    plot_graph(G, S_name)

result = process_image(10, df_enc, model, data_dir)
anc_img_names, data_dir, img, img_path, closest_idx, distance = result
plot_closest_imgs(anc_img_names, data_dir, img_path, closest_idx, distance,
no_of_closest=10)

```

تابع **process_image** یک فهرست، یک دیتافریم **df_enc** حاوی نام فایل‌های تصویر و جاسازی‌های آن‌ها، یک «مدل» از پیش آموزش‌دیده و یک فهرست داده **data_dir** می‌گیرد. تصویر مشخص شده توسط ایندکس را بارگذاری می‌کند، آن را از طریق مدل پردازش می‌کند تا جاسازی آن را به دست آورد، و سپس فاصله اقلیدسی بین این جاسازی و جاسازی همه تصاویر دیگر در قاب داده را محاسبه می‌کند. نزدیک‌ترین تصاویر با مرتب‌سازی فاصله‌ها تعیین می‌شوند و تابع اطلاعاتی مانند نام، مسیرها و فواصل این نزدیک‌ترین تصاویر را برمی‌گرداند. تابع **load_image** از کتابخانه تصویربرداری پایتون **PIL** برای باز کردن یک تصویر مشخص شده توسط مسیر آن استفاده می‌کند. تابع **create_graph** یک نمودار تولید می‌کند که در آن گره‌ها تصاویر را نشان می‌دهند و یال‌ها تصویر ورودی را بر اساس فواصل محاسبه شده به نزدیک‌ترین همسایگان خود متصل می‌کنند. تابع **Plot_graph** این نمودار را با تصاویری که بر اساس روابط آنها قرار گرفته‌اند، تجسم می‌کند. در نهایت، تابع **plot_closest_imgs** این مراحل را برای پردازش یک تصویر، ایجاد یک نمودار، و ترسیم نزدیک‌ترین تصاویر، با توجه به تعداد خاصی **no_of_closest**، در اینجا: ۱۰ از نزدیک‌ترین همسایگان ترکیب می‌کند. مثال در پایان این توابع را با پردازش یک تصویر، ایجاد یک نمودار و تجسم نزدیک‌ترین تصاویر با استفاده از توابع مشخص شده نشان می‌دهد.

خروجی برای یک نمونه **anchor** به صورت زیر می‌باشد.

مشاهده میکنیم که مدل به خوبی توانسته که عکس های مشابه را نزدیک کند.



شکل ۷ : top 10 برای triplet loss

محاسبه ROC

برای محاسبه ROC ابتدا نیاز میباشد که از مدل خروجی بگیریم و اگر به هم نزدیک بودند label برابر ۱ بدهیم و در غیر این صورت label را صفر بدهیم و با label های اصلی مقایسه کنیم.

```
# Function to calculate similarity scores
def calculate_similarity(embedding1, embedding2):
    # Cosine similarity is a common choice
    return torch.nn.functional.cosine_similarity(embedding1,
embedding2).cpu().numpy()
    return -torch.sqrt(torch.sum((embedding1 - embedding2) ** 2)).cpu().numpy()

# Prepare data for ROC
similarity_scores = []
true_labels = []

with torch.no_grad():
    for batch in valid_loader:
        anchor_images, positive_images, negative_images = batch
        anchor_images, positive_images, negative_images =
anchor_images.to(device), positive_images.to(device), negative_images.to(device)
```

```

# Forward pass for anchor-positive pairs
anchor_embeddings = model(anchor_images)
positive_embeddings = model(positive_images)
scores_positive = calculate_similarity(anchor_embeddings,
positive_embeddings)
similarity_scores.extend(scores_positive)
true_labels.extend([1] * len(scores_positive)) # 1 for similar

# Forward pass for anchor-negative pairs
negative_embeddings = model(negative_images)
scores_negative = calculate_similarity(anchor_embeddings,
negative_embeddings)
similarity_scores.extend(scores_negative)
true_labels.extend([0] * len(scores_negative)) # 0 for not similar

```

تابع `calculate_similarity` امتیازهای شباهت بین دو `embedding` داده شده را با استفاده از شباهت `Euclidian` محاسبه می‌کند، یک معیار رایج برای اندازه‌گیری شباهت بین بردارها. در کد بعدی، امتیاز شباهت و برچسب‌های واقعی برای تجزیه و تحلیل ویژگی‌های عملیاتی گیرنده `ROC` آماده شده است. در یک حلقه اعتبارسنجی `valid_loader`، برای هر دسته، تصاویر `anchor`، مثبت و منفی از طریق مدل پردازش می‌شوند تا جاسازی‌های مربوطه را به دست آورند. سپس نمرات شباهت کسینوس برای جفت‌های `anchor` مثبت و `anchor` منفی محاسبه می‌شود. این امتیازها، همراه با برچسب‌های واقعی مربوطه (۱ برای جفت‌های مشابه و ۰ برای جفت‌های غیرمشابه)، جمع‌آوری می‌شوند تا مجموعه داده‌هایی را برای تجزیه و تحلیل `ROC` تشکیل دهند. فهرست نهایی «نمرات_شباهت» حاوی امتیازهای شباهت کسینوس محاسبه شده است، و فهرست «برچسب‌های_واقعی» حاوی برچسب‌های حقیقت زمینی مربوطه است که ارزیابی عملکرد مدل را در تمایز بین جفت‌های مشابه و غیرمشابه بر اساس جاسازی‌های به دست آمده امکان‌پذیر می‌سازد.

در نهایت به کمک کد زیر `ROC` ره به کمک کتابخانه‌های آماده محاسبه می‌کنیم.

```

from sklearn.metrics import roc_curve, auc
# Compute ROC curve and ROC area
fpr, tpr, _ = roc_curve(true_labels, similarity_scores)
roc_auc = auc(fpr, tpr)

```

و در نهایت نیز میتوان به راحتی `ROC` را رسم کرد.

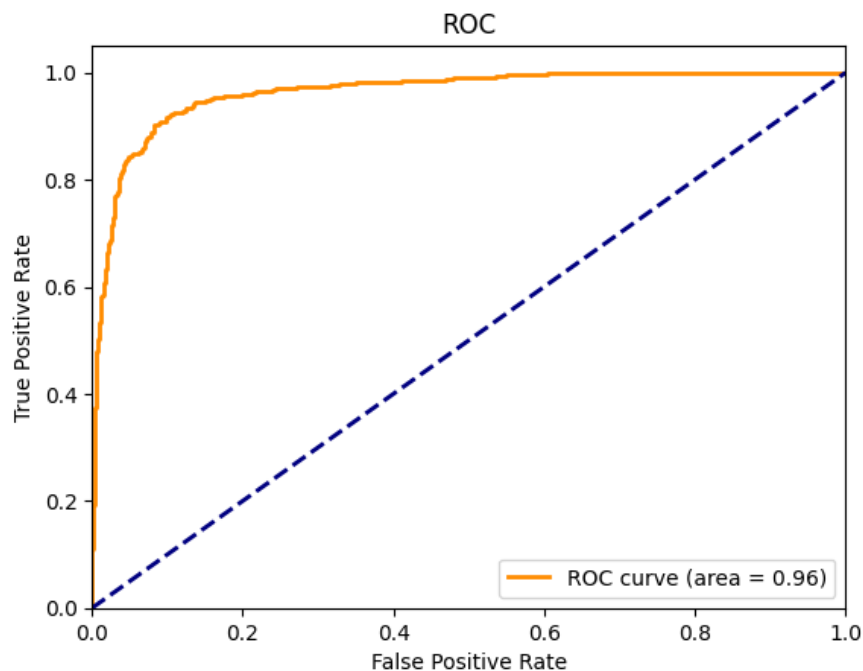
```

# Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' %
roc_auc)
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('ROC')
plt.legend(loc="lower right")
plt.show()
```

منحنی ROC برای triplet loss به صورت زیر می باشد. برای داده های validation



شکل ۸: منحنی ROC برای triplet loss

همانطور که مشاهده میشود مدل عملکرد مناسب و خوبی دارد که از top 10 آن هم مشخص بود. (مساحت زیر منحنی ۹۶ میباشد)

از آنجایی که در صورت سوال از ما مقادیر TPR به ازای FPR های متفاوت را خواسته است به همین دلیل از کد زیر برای رسم ROC استفاده میشود که این نقاط هم بر روی آن مقادیرشان مشخص باشد.

```
# Function to find closest index in an array to a given value
def find_nearest(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return idx

# Desired FPR values
desired_fprs = [0.5, 0.1, 0.01, 0.001]
```

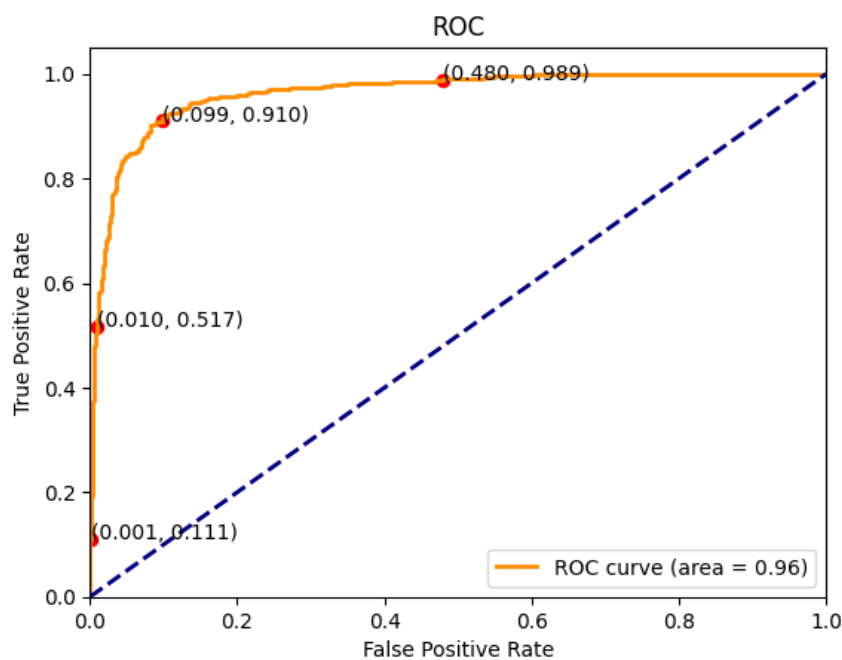


```

# Find nearest FPRs and corresponding TPRs
nearest_fprs = [fpr[find_nearest(fpr, desired_fpr)] for desired_fpr in
desired_fprs]
nearest_tprs = [tpr[find_nearest(fpr, desired_fpr)] for desired_fpr in
desired_fprs]
# Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' %
roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
# Annotate desired points
for fpr_point, tpr_point in zip(nearest_fprs, nearest_tprs):
    plt.scatter(fpr_point, tpr_point, color='red')
    plt.text(fpr_point, tpr_point, f'({fpr_point:.3f}, {tpr_point:.3f})')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC')
plt.legend(loc="lower right")
plt.show()

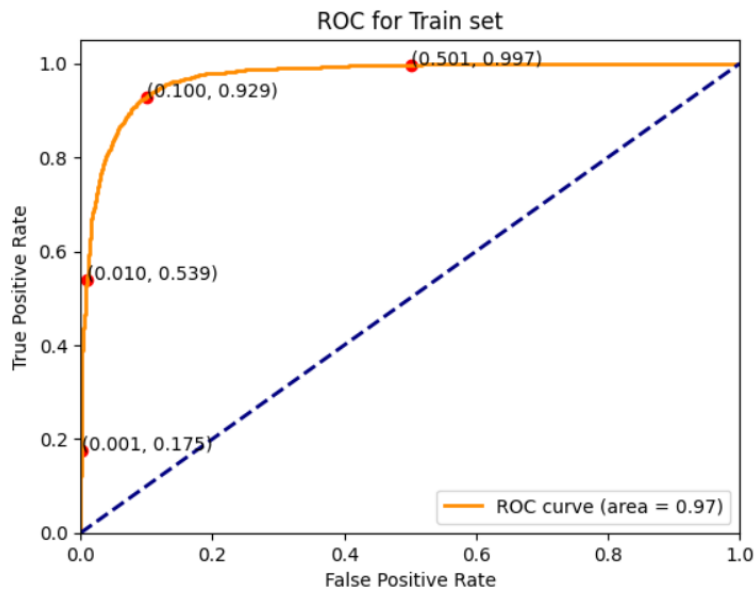
```

خروجی نهایی به صورت زیر میشود. برای داده های validation



شکل ۹: منحنی ROC برای triplet loss به همراه مقادیر مختلف TPR

برای داده های Train به شکل زیر است:



شکل ۱۰ - منحنی ROC برای triplet loss به همراه مقادیر مختلف TPR

TPR مختلف در قسمت ج آورده شده است.

CONTRASTIVE LOSS

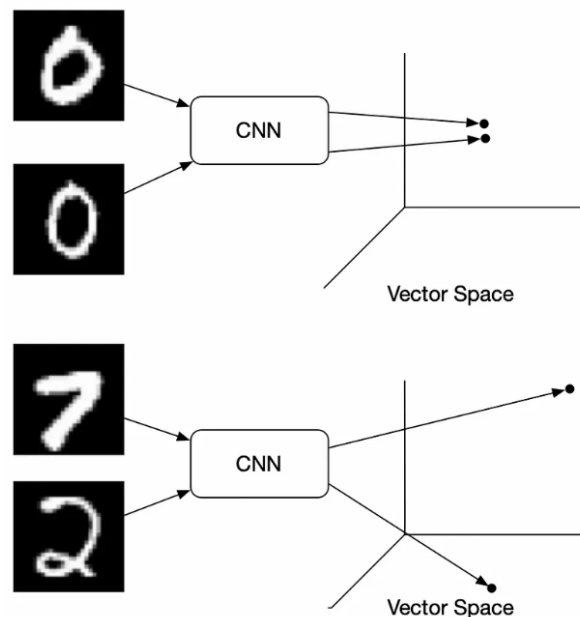
تعریف

یک تابع از **loss** است که معمولاً در شبکه های سیامی و سه گانه برای کارهایی مانند تشخیص چهره، یادگیری شباهت تصویر و تجزیه و تحلیل شباهت متن استفاده می شود. این تابع از **loss** مدل را تشویق می کند تا نمایش ویژگی های متمایز را برای کلاس ها یا جفت نقاط داده مختلف بیاموزد. ایده اصلی این است که فاصله بین جفت های مشابه یا مثبت را به حداقل برسانیم در حالی که فاصله بین جفت های نامشابه یا منفی را در فضای ویژگی به حداکثر برسانیم. **contrastive loss** به صورت زیر تعریف می شود:

$$L(Y, D) = (1 - Y) * (1/2) * D^2 + Y * (1/2) * \max(0, \text{margin} - D)^2$$

در اینجا، Y یک برچسب باینری را نشان می دهد (۱ برای جفت های مشابه، ۰ برای جفت های غیرمشابه)، D فاصله اقلیدسی یا متریک فاصله مناسب دیگری بین نمایش ویژگی های دو نقطه داده است، و '**margin**' یک فراپارامتر است که حداقل مورد نظر را تعیین می کند. جدایی بین جفت های مشابه و غیر مشابه از **contrastive loss** مدل را ارتقا می دهد تا ورودی های مشابه را نزدیک به هم و ورودی های غیرمشابه را با فاصله دور در فضای ویژگی ترسیم کند، و آن را برای یادگیری جاسازی هایی که روابط شباهت ذاتی

درون داده‌ها را به تصویر می‌کشد، ارزشمند می‌سازد، که می‌تواند برای وظایف مختلف مبتنی بر شباهت اعمال شود.



شکل ۱۱ - ساز و کار **contrastive loss**

پیاده سازی CONTRASTIVE LOSS

جهت پیاده سازی به راحتی از فرمول فوق استفاده میکنیم.

کلاس **ContrastiveLoss** یک تابع از **CONTRASTIVE loss** را تعریف می‌کند که برای آموزش شبکه‌های سیامی یا معماری‌های مشابه با هدف یادگیری جاسازی‌هایی استفاده می‌شود که در آن موارد مشابه در فضای جاسازی نزدیک‌تر هستند و نمونه‌های غیرمشابه با یک حاشیه از هم جدا می‌شوند. با توجه به دو مجموعه جاسازی «خروجی ۱» و «خروجی ۲» و یک برچسب دودویی که نشان می‌دهد نمونه‌ها مشابه هستند («برچسب = ۱») یا غیرمشابه («برچسب = ۰»)، تلفات بر اساس فاصله اقلیدسی بین محاسبه می‌شود. تعبیه‌ها اصطلاح **loss** از دو جزء تشکیل شده است: یکی برای جفت‌های مشابه، جریمه کردن مدل در صورتی که فاصله آنها از حاشیه بیشتر شود، و دیگری برای جفت‌های غیرمشابه، که مدل را تشویق می‌کند تا فاصله آنها را فراتر از حاشیه برود. تلفات نهایی میانگین این تلفات محاسبه شده است که شبکه را برای یادگیری جاسازی‌هایی که به طور مناسب شباهت یا عدم شباهت جفت‌های ورودی را مطابق با حاشیه مشخص شده منعکس می‌کند، ارتقا می‌دهد.

```

# Contrastive Loss Function
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=1):
        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,
2) +
                                     (label) * torch.pow(torch.clamp(self.margin
- euclidean_distance, min=0.0), 2))
        return loss_contrastive

```

آموزش مدل با CONTRASTIVE

برای آموزش مدل ابتدا پارامترهای آموزش را به صورت زیر تعریف میکنیم.

```

# Loss and Optimizer
criterion = ContrastiveLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Learning Rate Scheduler
scheduler = StepLR(optimizer, step_size=2, gamma=0.1)

# Device configuration
device = torch.device('cuda:1' if torch.cuda.is_available() else 'cpu')

```

از کد زیر برای آموزش استفاده میشود.

```

# Validation function
def validate(model, device, val_loader, criterion):
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for ((data1, data2), labels) in val_loader:
            data1, data2, labels = data1.to(device), data2.to(device),
labels.to(device)
            output1 = model(data1)
            output2 = model(data2)
            loss = criterion(output1, output2, labels)
            val_loss += loss.item()
    return val_loss / len(val_loader)

```

```

def train(model, device, train_loader, val_loader, optimizer, criterion,
scheduler, num_epochs, save_path='best_model.pth'):
    train_losses = []
    val_losses = []
    best_val_loss = float('inf')

    for epoch in range(num_epochs):
        model.train()
        train_loss = 0.0
        for batch_idx, ((data1, data2), labels) in tqdm(enumerate(train_loader),
total=len(train_loader), desc=f"Epoch {epoch+1}/{num_epochs}"):
            data1, data2, labels = data1.to(device), data2.to(device),
labels.to(device)
            optimizer.zero_grad()
            output1 = model(data1)
            output2 = model(data2)
            loss = criterion(output1, output2, labels)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        scheduler.step() # Update learning rate
        avg_train_loss = train_loss / len(train_loader)
        avg_val_loss = validate(model, device, val_loader, criterion)

        train_losses.append(avg_train_loss)
        val_losses.append(avg_val_loss)

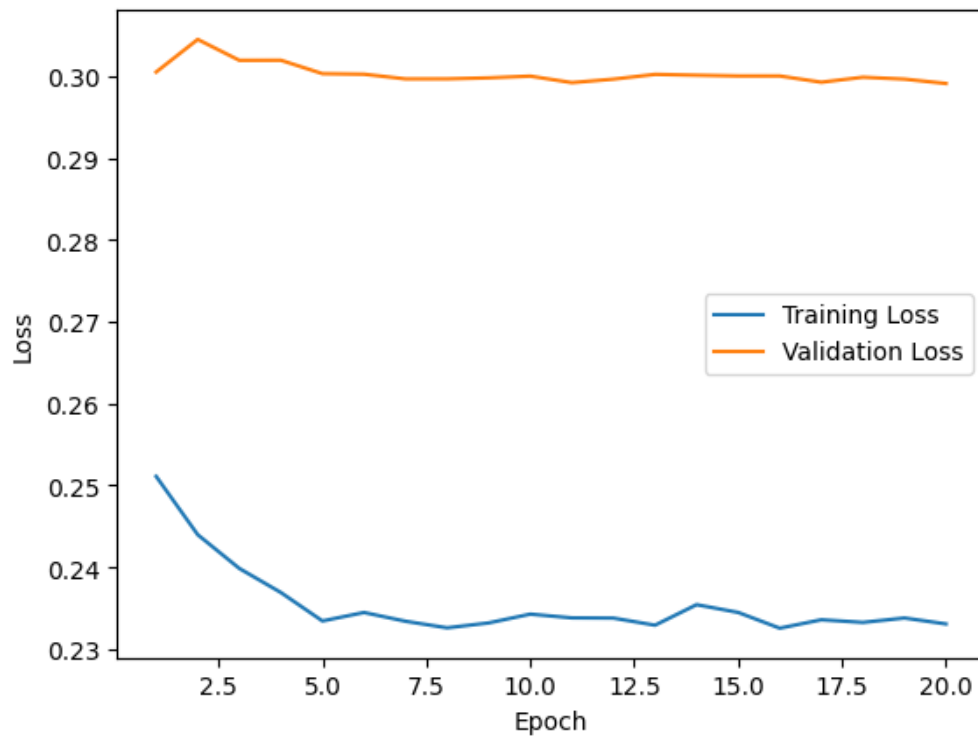
        print(f'Epoch {epoch+1}, Training Loss: {avg_train_loss}, Validation
Loss: {avg_val_loss}')

        # Save the model if the validation loss has decreased
        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            torch.save(model.state_dict(), save_path)
            print(f'Saving model with the best validation loss: {best_val_loss}')

    # Plotting
    epochs = range(1, num_epochs + 1)
    plt.plot(epochs, train_losses, label='Training Loss')
    plt.plot(epochs, val_losses, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

```

منحنی **loss** برای این حالت به صورت زیر میشود.

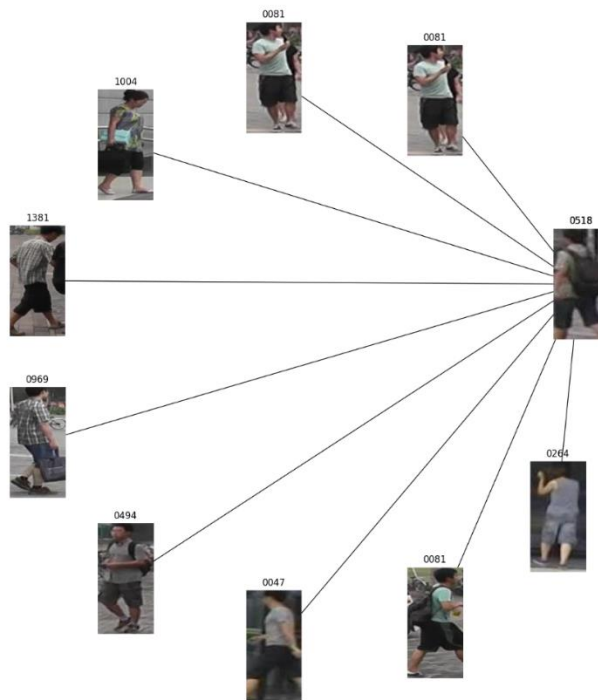


شکل ۱۲ : منحنی آموزش و ارزیابی پس از آموزش مدل با **contrastive loss**

همانطور که مشاهده میشود نزول انچنانی ای در **val loss** نداریم و تقریباً روی یک عدد ثابتی میماند (با **lr** های متفاوت هم به همین صورت میشود و نشان میدهد این **loss** برای این دیتاست خیلی مناسب نمیباشد).

پیدا کردن TOP 10

همانطور که در بخش قبل کد های آن آمده است بر اساس فاصله میتوان ۱۰ تا از نزدیک ترین به **anchor** را رسم کرد



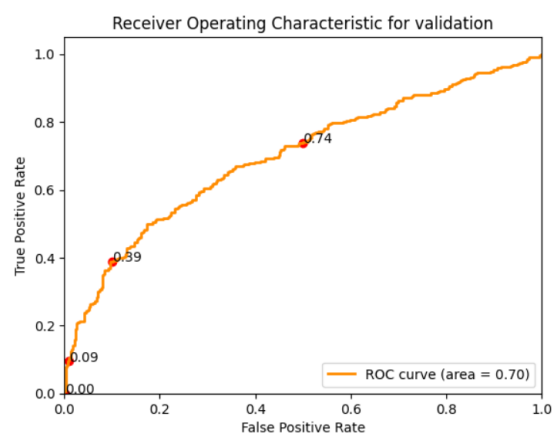
شکل ۱۳: top 10 برای contrastive loss

همانطور که مشاهده میشود این loss دقت مناسبی ندارد و عملکرد آن نیز در ۱۰ تای اول خیلی مناسب نمیباشد.

محاسبه ROC

منحنی ROC برای contrastive به صورت زیر میشود (همان کد های بخش triplet loss میباشد)

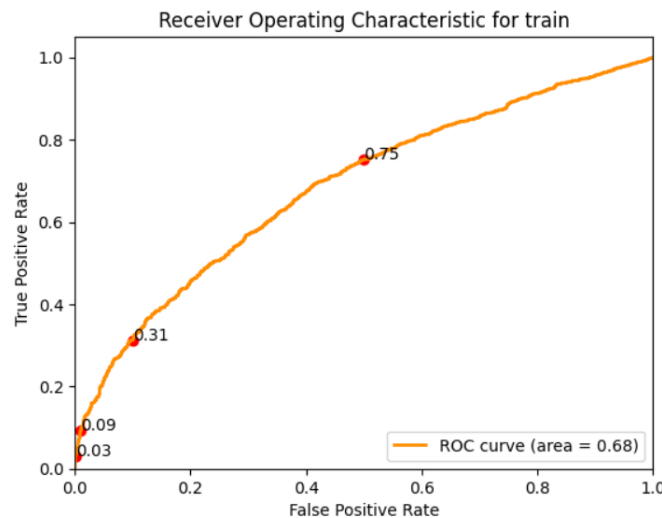
برای validation:



شکل ۱۴: منحنی ROC برای contrastive loss

همانطور که مشاهده میشود مساحت زیر منحنی ۷۶ میباشد که نشان از عملکرد نامناسب $loss$ برای مدل دارد.

برای train:



شکل ۱۰ - منحنی ROC برای $contrastive loss$

TPR مختلف در قسمت ج آورده شده است.

SENSIVITY OVER MARGIN برای ۲ بخش قبل

در قسمت تعریف $loss$ که در ۲ قسمت داشتیم یک قسمت $margin$ دارد که آن را یک در نظر گرفتیم. در این قسمت مدل را برای $margin$ های ۰,۱ و ۱۰ تریم میکنیم و خروجی ها را مشاهده میکنیم. کد این قسمت همانند قسمت قبل است با این تفاوت که عدد $margin$ را تغییر می دهیم. از ما خواسته شده است که برای ۳ $margin$ مقایسه را انجام دهیم، برای سومین $margin$ که عدد ۱ هست، نمودار های آن در قسمت های قبل آمده است و فقط در اینجا آن را با بقیه مقایسه می کنیم.

- Triplet:

```



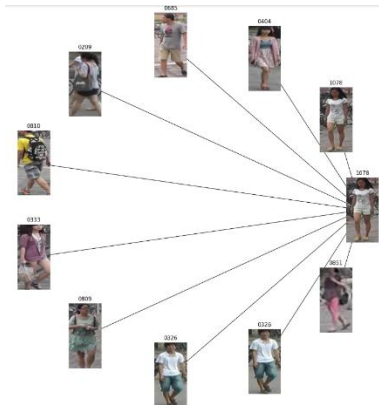
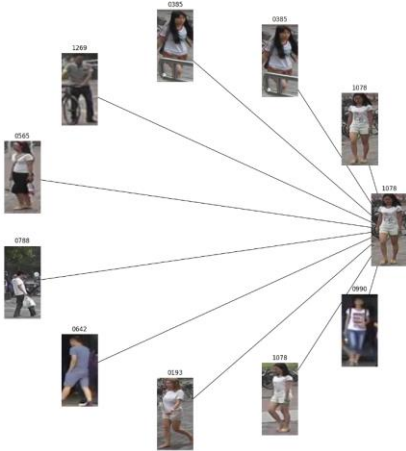
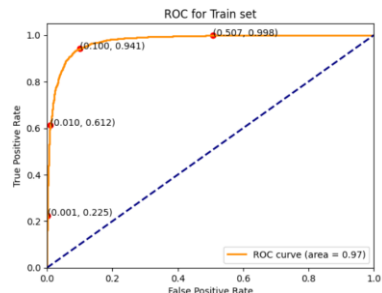
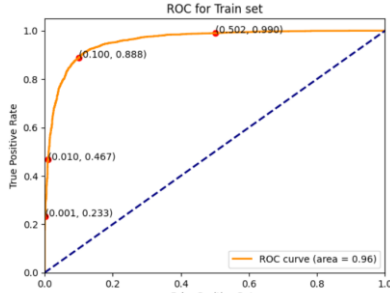
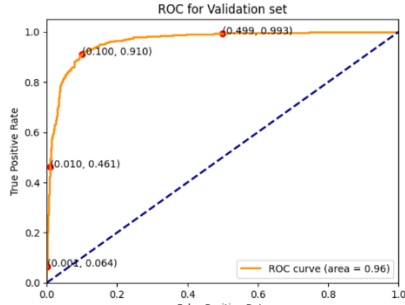
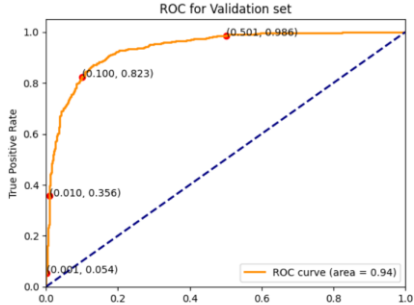
criterion = CustomTripletMarginLoss(margin=0.1)
optimizer = torch.optim.Adam(model.parameters(), lr= 0.001)

criterion = CustomTripletMarginLoss(margin=10)
optimizer = torch.optim.Adam(model.parameters(), lr= 0.001)

```


خروجی ها به شکل زیر خواهد بود:

جدول ۱ - margin های مختلف برای triplet

	margin=0.1	margin=10
loss		
Top 10		
ROC Train		
ROC Valid		
Similarity Accuracy	97.12%	95.38%

Similarity Accuracy برای margin=1 برابر ۹۸,۰۰٪ است.

برای بدست آوردن دقت شباهت سنجی از کد زیر استفاده می کنیم:

```
import torch
from tqdm import tqdm

device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
# Initialize counters for correct and total pairs
correct_pairs = 0
total_pairs = 0
model.eval()
with torch.no_grad():
    for A, P, N in tqdm(valid_loader):
        pass
        A, P, N = A.to(device), P.to(device), N.to(device)

        # Forward pass to get embeddings
        emb_A = model(A)
        emb_P = model(P)
        emb_N = model(N)
        # Compute Euclidean distances
        dist_pos = torch.norm(emb_A - emb_P, dim=1)
        dist_neg = torch.norm(emb_A - emb_N, dim=1)
        # Check if the distance of the positive pair is smaller than the distance
of the negative pair
        correct_pairs += torch.sum(dist_pos < dist_neg).item()
        total_pairs += A.size(0) # Batch size
# Calculate similarity accuracy
similarity_accuracy = correct_pairs / total_pairs
print(f"Similarity Accuracy: {0.98 * 100:.2f}%")
```

در این قطعه کد، یک مدل آموزش دیده بر روی یک مجموعه داده اعتبارسنجی **valid_loader** ارزیابی می شود تا عملکرد آن در تمایز بین جفت های مشابه و غیر مشابه بر اساس جاسازی های آموخته شده ارزیابی شود. مدل روی حالت ارزیابی (**model.eval()**) تنظیم می شود، جفت های **A**، **P** و **N** از طریق مدل پردازش می شوند تا **embedding** مربوطه خود را به دست آورند. سپس فواصل اقلیدسی بین تعبیه های جفت **anchor** مثبت و **anchor** منفی محاسبه می شود. تعداد جفت های صحیح، که در آن فاصله جفت مثبت کوچکتر از جفت منفی است، به همراه تعداد کل جفت ها در دسته جمع می شود. در نهایت دقت شباهت به عنوان نسبت جفت های صحیح به جفت کل محاسبه می شود و نتیجه چاپ می شود. دقت شباهت نشان دهنده نسبت جفت های مشابه است که به درستی شناسایی شده اند و بینشی را در مورد توانایی مدل برای تمایز بین نمونه های مثبت و منفی ارائه می دهد.

حاشیه در از triplet loss نقش مهمی در تعریف جدایی بین جفت مثبت و منفی در فضای تعبیه دارد. حاشیه یک فرارامتر است که حداقل فاصله قابل قبول بین جاسازی‌های یک جفت anchor مثبت و یک جفت anchor منفی را تعیین می‌کند. انتخاب اندازه حاشیه می‌تواند تاثیر قابل توجهی بر عملکرد و رفتار مدل داشته باشد:

۱. حاشیه بزرگتر: حاشیه بزرگتر جداسازی دقیق تری را بین جفت های مثبت و منفی اعمال می‌کند. این می‌تواند منجر به خوشه های متمایزتر و به خوبی جدا شده در فضای جاسازی شود و نمایش های آموخته شده را متمایزتر کند. با این حال، تعیین حاشیه بسیار بزرگ ممکن است منجر به ضرری شود که بهینه سازی آن بسیار دشوار می‌شود و منجر به همگرایی آهسته یا هم‌گرایی به راه‌حل‌های غیربهینه می‌شود.

۲. حاشیه کوچکتر: حاشیه کوچکتر امکان انعطاف پذیری بیشتری را در قرار دادن جفت های مثبت و منفی در فضای تعبیه می‌دهد. این می‌تواند منجر به همگرایی سریع‌تر در طول آموزش شود، اما ممکن است منجر به تعبیه‌هایی شود که کمتر تبعیض‌آمیز هستند و مستعد همپوشانی بین خوشه‌های مثبت و منفی هستند.

۳. قانون تعادل: حاشیه باید با دقت انتخاب شود تا تعادلی بین تشویق جداسازی جفت‌های متفاوت و اجتناب از دشواری بیش از حد در train ایجاد شود. این بستگی به ویژگی های خاص مجموعه داده و وظیفه در دست دارد. معمولاً توصیه می‌شود در طول توسعه مدل، اندازه‌های حاشیه‌های مختلف را آزمایش کنید تا مقداری را که بهترین عملکرد را در مجموعه اعتبارسنجی دارد، بیابید.

۴. حساسیت کار: تاثیر حاشیه ممکن است بسته به کار شباهت خاص متفاوت باشد. در برخی موارد، یک حاشیه کوچک ممکن است کافی باشد، در حالی که در موارد دیگر، ممکن است حاشیه بزرگتر برای دستیابی به تبعیض معنادار لازم باشد.

TPR به ازای margin‌های مختلف برای داده‌های train-validation به شرح زیر است:

جدول ۲ - TPR به ازای margin‌های مختلف برای داده های validation

False Positive Rate	Margin=0.1	Margin=1	Margin=10
0.5	۰/۹۹۳	۰/۹۸۹	۰/۹۸۶
0.1	۰/۹۱	۰/۹۱	۰/۸۲۳
0.01	۰/۴۶۱	۰/۵۱۷	۰/۳۵۶
0.001	۰/۰۶۴	۰/۱۱۱	۰/۰۵۶

جدول ۳ - TPR به ازای margin های مختلف برای داده های train

False Positive Rate	Margin=0.1	Margin=1	Margin=10
0.5	0.998	0.997	0.99
0.1	0.941	0.929	0.888
0.01	0.612	0.539	0.467
0.001	0.25	0.179	0.233

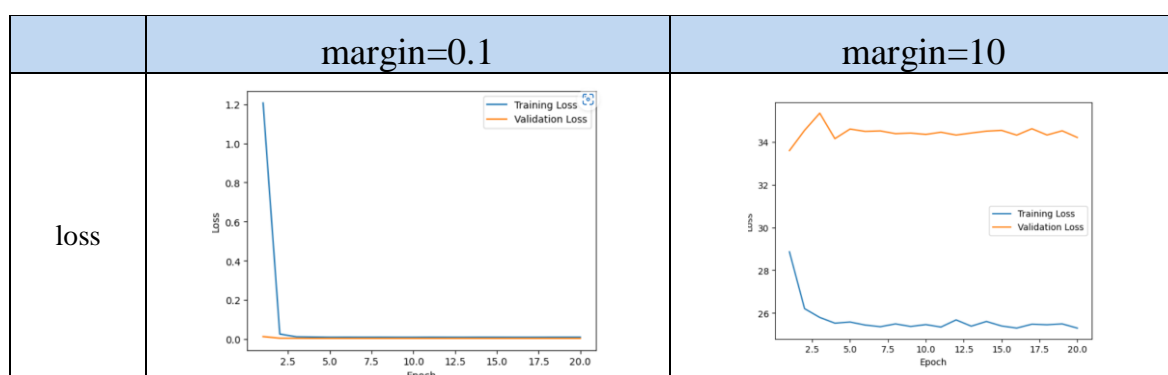
از نتایج بالا مشاهده میکنیم که این لاس برای margin هایی که تعریف کردیم، زیاد حساس نیست ولی اگر margin های را خیلی بزرگ و یا خیلی کوچک میکردیم در عملکرد آن تغییر ایجاد میشد. ما باید با آزمایش بهترین margin را انتخاب کنیم که در اینجا عدد ۱ بهترین margin است و در بقیه margin ها مشاهده می کنیم که دچار افت دقت شده ایم.



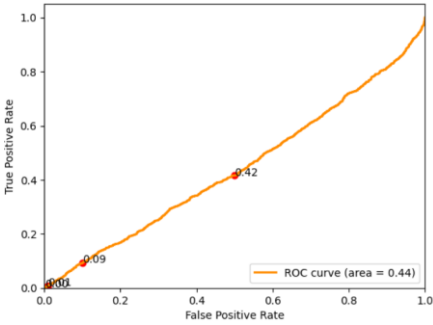
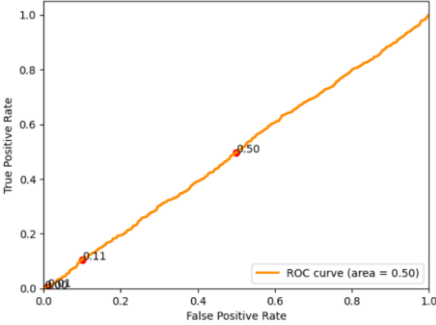
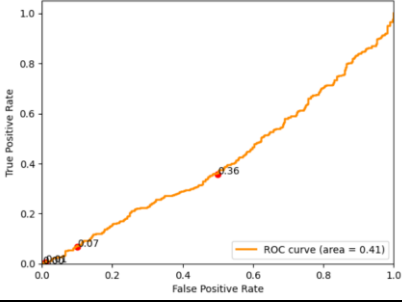
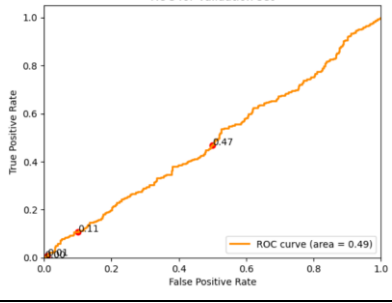
اندازه حاشیه بهینه معمولاً مجموعه داده ها و وظایف خاص است. اغلب برای یافتن تعادل مناسب نیاز به آزمایش یا تنظیم فرایارامتر دارد. حاشیه بسیار کوچک یا خیلی بزرگ می تواند بر عملکرد مدل تأثیر منفی بگذارد. نکته کلیدی یافتن اندازه حاشیه است که مدل را تشویق می کند تا ویژگی های مفید و متمایز را بیاموزد بدون اینکه کار یادگیری را بی دلیل سخت کند.

- contrastive:

همانند قسمت قبل برای margin های ۰,۱ و ۱۰ مدل را ترین میکنیم.

جدول ۴ - margin های مختلف برای contrastive



Top 10		
ROC Train	 <p>ROC for Train set</p> <p>True Positive Rate</p> <p>False Positive Rate</p> <p>ROC curve (area = 0.44)</p> <p>0.09, 0.42</p>	 <p>ROC for Train set</p> <p>True Positive Rate</p> <p>False Positive Rate</p> <p>ROC curve (area = 0.50)</p> <p>0.11, 0.50</p>
ROC Valid	 <p>ROC for Validation set</p> <p>True Positive Rate</p> <p>False Positive Rate</p> <p>ROC curve (area = 0.41)</p> <p>0.07, 0.36</p>	 <p>ROC for Validation set</p> <p>True Positive Rate</p> <p>False Positive Rate</p> <p>ROC curve (area = 0.49)</p> <p>0.11, 0.47</p>
Similarity Accuracy	49.50%	46.00%

برای $\text{margin}=1$ دقت شباهت سنجی: ۴۸,۵۰٪

مشاهده می کنیم که در این حالت این لاس به شدت به مقدار margin وابسته است نسبت به triplet و تغییر زیادی را در مقدار دقت شباهت سنجی و مقادیر ROC می بینیم.

TPR به ازای margin‌های مختلف برای داده های train-validation به شرح زیر است:

جدول ۵ - TPR به ازای margin‌های مختلف برای داده های validation

False Positive Rate	Margin=0.1	Margin=1	Margin=10
0.5	0.36	0.74	0.47
0.1	0.07	0.39	0.11
0.01	0.01	0.09	0.01
0.001	0	0	0

جدول ۶ - TPR به ازای margin‌های مختلف برای داده های train

False Positive Rate	Margin=0.1	Margin=1	Margin=10
0.5	0.42	0.75	0.5
0.1	0.09	0.31	0.11
0.01	0.01	0.09	0.01
0.001	0	0.03	0

اندازه حاشیه در از contrastive loss ، شبیه به triplet loss ، بر نحوه یادگیری embedding توسط مدل تأثیر می‌گذارد. از loss معمولاً در معماری شبکه های سیامی برای کارهایی مانند شباهت تصویر استفاده می‌شود. حاشیه یک آستانه تعیین می‌کند و حداقل فاصله ای را که جفت های مشابه باید داشته باشند و حداکثر فاصله را برای جفت های غیرمشابه در فضای جاسازی تعیین می‌کند.

۱. حاشیه بزرگتر: حاشیه بزرگتر، جداسازی دقیق تری را بین جفت های مثبت و منفی اعمال می‌کند. این می‌تواند منجر به خوشه های متمایزتر و به خوبی جدا شده در فضای جاسازی شود و نمایش های آموخته شده را متمایزتر کند. با این حال، تعیین حاشیه بسیار زیاد ممکن است منجر به ضرری شود که بهینه‌سازی آن بسیار دشوار می‌شود، روند آموزش را کند می‌کند یا باعث مشکلات همگرایی می‌شود.

۲. حاشیه کوچکتر: حاشیه کوچکتر امکان انعطاف پذیری بیشتری را در قرار دادن جفت های مثبت و منفی در فضای تعبیه می‌دهد. این می‌تواند منجر به همگرایی سریع‌تر در طول آموزش شود، اما ممکن است منجر به تعبیه‌هایی شود که کمتر تبعیض‌آمیز هستند و مستعد همپوشانی بین خوشه‌های مثبت و منفی هستند.

۳. قانون متعادل سازی: حاشیه باید با دقت انتخاب شود تا تعادل بین تشویق جدایی جفت های متفاوت و اجتناب از دشواری بیش از حد در **train** ایجاد شود. این بستگی به ویژگی های خاص مجموعه داده و وظیفه در دست دارد. معمولاً توصیه می شود در طول توسعه مدل، اندازه های حاشیه های مختلف را آزمایش کنید تا مقداری را که بهترین عملکرد را در مجموعه اعتبارسنجی دارد، بیابید.

۴. حساسیت کار: تأثیر حاشیه ممکن است بسته به کار شباهت خاص متفاوت باشد. در برخی موارد، یک حاشیه کوچک ممکن است کافی باشد، در حالی که در موارد دیگر، ممکن است حاشیه بزرگتر برای دستیابی به تبعیض معنادار لازم باشد.

از نتایج دو **loss** میتوانیم نتیجه بگیریم که **triplet loss** بهتر است و حساسیت کمتری نسبت به **margin** دارد.

حساسیت این دو لاس به اندازه حاشیه می تواند به عوامل متعددی از جمله ویژگی های خاص مجموعه داده و ماهیت کار شباهت بستگی داشته باشد. با این حال، به طور کلی، **triplet loss** اغلب به انتخاب حاشیه در مقایسه با **loss** کنتراست حساس تر در نظر گرفته می شود. در اینجا دلیل آن است:

۱. سه گانه ها به محدودیت های متعادل نیاز دارند: در این لاس، هر سه گانه **train** از یک **anchor**، یک مثال مثبت (همان کلاس **anchor**) و یک مثال منفی (کلاس مختلف) تشکیل شده است. حاشیه در **loss** حداقل تفکیک مطلوب بین جفت مثبت و منفی را تعیین می کند. اگر حاشیه خیلی کوچک باشد، ممکن است منجر به تبعیض ناکافی شود، زیرا ممکن است مدل به طور موثر یاد نگیرد که جفت های منفی را به اندازه کافی از **anchor** دور کند. برعکس، اگر حاشیه خیلی زیاد باشد، می تواند آموزش را بیش از حد چالش برانگیز کند و ممکن است منجر به همگرایی کندتر شود.

۲. **contrastive loss** امکان مقایسه زوجی بیشتر را فراهم می کند: از طرف دیگر **contrastive loss** مستقیماً جفت ها (مثبت و منفی) را مقایسه می کند و نیازی به سه قلو ندارد. حاشیه در **loss** آستانه شباهت یا عدم تشابه بین جفت ها را مشخص می کند. در حالی که مارجین هنوز در **contrastive loss** مهم است، ممکن است حساسیت کمتری در نظر گرفته شود زیرا عمل متعادل کننده مورد نیاز در **loss** **triplet** را شامل نمی شود.

۳. **triplet loss** بر فواصل نسبی تأکید می کند: **triplet loss** به صراحت بر فواصل نسبی بین جفت های **anchor** مثبت و **anchor** منفی تأکید می کند. حاشیه به طور مستقیم بر میزان جریمه شدن مدل برای

عدم رعایت تفکیک مورد نظر تأثیر می گذارد. در مقابل، **contrastive loss** بر روی مشابه یا غیرمشابه بودن جفت ها تمرکز می کند و حاشیه آستانه این تعیین را تعیین می کند.

در عمل، حساسیت این لاس ها به نوع دیتاست برمیگردد که در اینجا دیدیم که **triplet** حساسیت کمتری دارد.

FISHER DISCRIMINANT CONTRASTIVE LOSS

- تعریف

(FDC) یک رویکرد جدید برای آموزش شبکه های سیامی است که هدف آن استفاده از اصول تحلیل تشخیصی فیشر (FDA) است. مفهوم اصلی این است که جداسازی بین طبقات مختلف (پراکندگی بین طبقاتی) را به حداکثر برسانیم و در عین حال فاصله بین نمونه های یک کلاس را به حداقل برسانیم (پراکندگی درون کلاسی). این امر با متضاد جفت نقاط داده (contrastive) ، کنار هم قرار دادن آنها را از یک کلاس و جدا کردن نقاط داده از کلاسهای مختلف به دست می آید. هدف افزایش قدرت تمایز ویژگی های آموخته شده است و در نتیجه عملکرد وظایف طبقه بندی را بهبود می بخشد. FDC loss به ویژه در سناریوهایی مفید است که استخراج ویژگی های قوی و یادگیری متریک بسیار مهم است، مانند تشخیص تصویر یا تجزیه و تحلیل هیستوپاتولوژی.

فرمول این loss به صورت زیر می باشد.

$$l_{fdc} = (2 - \lambda)tr(U^T \widetilde{S_W} U) + [-\lambda tr(U^T \widetilde{S_B} U) + \alpha]_+$$

- پیاده سازی FDC

به راحتی بر اساس anchor و pos و neg ماتریس های within و between را ساخته و محاسبات فرمول بالا را انجام می دهیم.

```
class FisherContrastiveLoss(nn.Module):
    def __init__(self, margin_in_loss=1.0, epsilon_Sw=0.0001, epsilon_Sb=0.0001,
lambda_=0.01):
        super(FisherContrastiveLoss, self).__init__()
        self.margin_in_loss = margin_in_loss
        self.epsilon_Sw = epsilon_Sw
        self.epsilon_Sb = epsilon_Sb
```



```

self.lambda_ = lambda_

def forward(self, anchor_embeddings, positive_embeddings,
negative_embeddings, weights_lastLayer):
    # Calculation of within scatter (anchor-positive)
    temp1 = anchor_embeddings - positive_embeddings
    S_within = torch.matmul(temp1.t(), temp1)

    # Calculation of between scatter (anchor-negative)
    temp2 = anchor_embeddings - negative_embeddings
    S_between = torch.matmul(temp2.t(), temp2)

    # Strengthen main diagonal of S_within and S_between
    I_matrix = torch.eye(S_within.shape[0], device=anchor_embeddings.device)
    I_matrix_Sw = self.epsilon_Sw * I_matrix
    I_matrix_Sb = self.epsilon_Sb * I_matrix
    S_within = S_within + I_matrix_Sw
    S_between = S_between + I_matrix_Sb

    # Calculation of variance of projection considering within scatter
    temp3 = torch.matmul(torch.matmul(weights_lastLayer.t(), S_within),
weights_lastLayer)
    within_scatter_term = torch.trace(temp3)

    # Calculation of variance of projection considering between scatter
    temp4 = torch.matmul(torch.matmul(weights_lastLayer.t(), S_between),
weights_lastLayer)
    between_scatter_term = torch.trace(temp4)

    # Calculation of loss
    loss = ((2 - self.lambda_) * within_scatter_term) +
torch.max(torch.tensor(0.0).to(anchor_embeddings.device), self.margin_in_loss -
(self.lambda_ * between_scatter_term))

    return loss

```

کلاس **FisherContrastiveLoss** یک تابع **loss** را تعریف می‌کند که اطلاعات فیشر را برای تقویت یادگیری ویژگی‌های متمایز در شبکه‌های عصبی ترکیب می‌کند. **loss** بر اساس ماتریس‌های پراکندگی درون کلاسی **S_within** و پراکندگی بین کلاسی **S_between** embedding های به دست آمده از جفت های **anchor** مثبت و **anchor** منفی محاسبه می‌شود. این ماتریس‌های پراکندگی واریانس درون و بین کلاس‌ها را به ترتیب کمیت می‌کنند. موزن اصلی **S_within** و **S_between** برای جلوگیری از تکینگی تقویت شده است. **loss** شامل عباراتی است که واریانس پیش‌بینی را با در نظر گرفتن درون و بین پراکندگی‌ها نشان می‌دهد. به علاوه، یک اصطلاح منظم‌سازی، که توسط فرآپارامتر **lambda**

کنترل می‌شود، برای متعادل کردن مشارکت‌های پراکنده معرفی شده است. از دست دادن مدل را تشویق می‌کند تا واریانس پیش‌بینی‌ها را در همان کلاس به حداکثر برساند در حالی که واریانس بین کلاس‌های مختلف را به حداقل برساند. پارامتر `margin_in_loss` برای تنظیم حداقل تفکیک مطلوب بین جفت‌های مثبت و منفی استفاده می‌شود و `loss` نهایی به عنوان ترکیبی از درون و بین عبارت‌های پراکنده با منظم‌سازی محاسبه می‌شود.

- دیتالودر

برای این بخش از دیتالودرهای بخش `triplet` استفاده می‌شود.

```
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, dataframe, transform=None):
        self.data = dataframe
        self.transform = transform

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

    def __getitem__(self, idx):
        anchor_img = Image.open('Homework5_dataset/' + self.data.iloc[idx, 0])
        positive_img = Image.open('Homework5_dataset/' + self.data.iloc[idx, 2])
        negative_img = Image.open('Homework5_dataset/' + self.data.iloc[idx,
1]) # Load negative image

        if self.transform:
            anchor_img = self.transform(anchor_img)
            positive_img = self.transform(positive_img)
            negative_img = self.transform(negative_img) # Transform negative
image

        return anchor_img, positive_img, negative_img # Return all three images

# Define data transforms
transform = transforms.Compose([
    transforms.Resize((224, 224)), # Assuming EfficientNet B0 input size
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225]) # Imagenet normalization
])

# Create a custom dataset
custom_dataset = CustomDataset(df, transform=transform)
```

```

# Split dataset into training and validation sets (80% train, 20% val)
train_size = int(0.8 * len(custom_dataset))
val_size = len(custom_dataset) - train_size
train_dataset, val_dataset = random_split(custom_dataset, [train_size, val_size])

# Create data loaders
batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)

```

- آموزش مدل

به کمک کد زیر لوپ ترین را مینویسیم.

```

# Training loop
num_epochs = 20
best_val_loss = float('inf')
early_stopping_patience = 5
no_improvement_count = 0

# Lists to store the losses
train_losses = []
val_losses = []

# Training loop
for epoch in range(num_epochs):
    model.train()
    total_loss = 0.0
    progress_bar = tqdm(train_loader, desc=f'Epoch {epoch+1}/{num_epochs}',
leave=False)

    for batch in progress_bar:
        anchor_images, positive_images, negative_images = batch
        anchor_images, positive_images, negative_images =
anchor_images.to(device), positive_images.to(device), negative_images.to(device)
        # Forward pass
        anchor_embeddings = model(anchor_images) # Embeddings for anchor
        positive_embeddings = model(positive_images) # Embeddings for positive
        negative_embeddings = model(negative_images) # Embeddings for negative

        # Compute Fisher Contrastive loss
        loss = contrastive_loss(anchor_embeddings, positive_embeddings,
negative_embeddings, model.efficientnet.classifier[-1].weight)

        # Backpropagation

```

```

optimizer.zero_grad()
loss.backward()
optimizer.step()

total_loss += loss.item()
progress_bar.set_postfix({'Train Loss': total_loss / len(progress_bar)})

# Calculate average training loss for this epoch
avg_train_loss = total_loss / len(train_loader)
train_losses.append(avg_train_loss)

# Validation loop
model.eval()
val_loss = 0.0
with torch.no_grad():
    for batch in progress_bar:
        anchor_images, positive_images, negative_images = batch
        anchor_embeddings, positive_embeddings, negative_embeddings =
anchor_images.to(device), positive_images.to(device), negative_images.to(device)

        # Forward pass
        anchor_embeddings = model(anchor_images) # Embeddings for anchor
        positive_embeddings = model(positive_images) # Embeddings for
positive
        negative_embeddings = model(negative_images) # Embeddings for
negative

        # Compute Fisher Contrastive loss
        loss = contrastive_loss(anchor_embeddings, positive_embeddings,
negative_embeddings, model.efficientnet.classifier[-1].weight)

        val_loss += loss.item()

# Calculate average validation loss for this epoch
avg_val_loss = val_loss / len(val_loader)
val_losses.append(avg_val_loss)

print(f'Epoch [{epoch+1}/{num_epochs}] | Train Loss: {avg_train_loss:.4f} |
Val Loss: {avg_val_loss:.4f}')

if avg_val_loss < best_val_loss:
    best_val_loss = avg_val_loss
    no_improvement_count = 0
else:
    no_improvement_count += 1

if no_improvement_count >= early_stopping_patience:

```

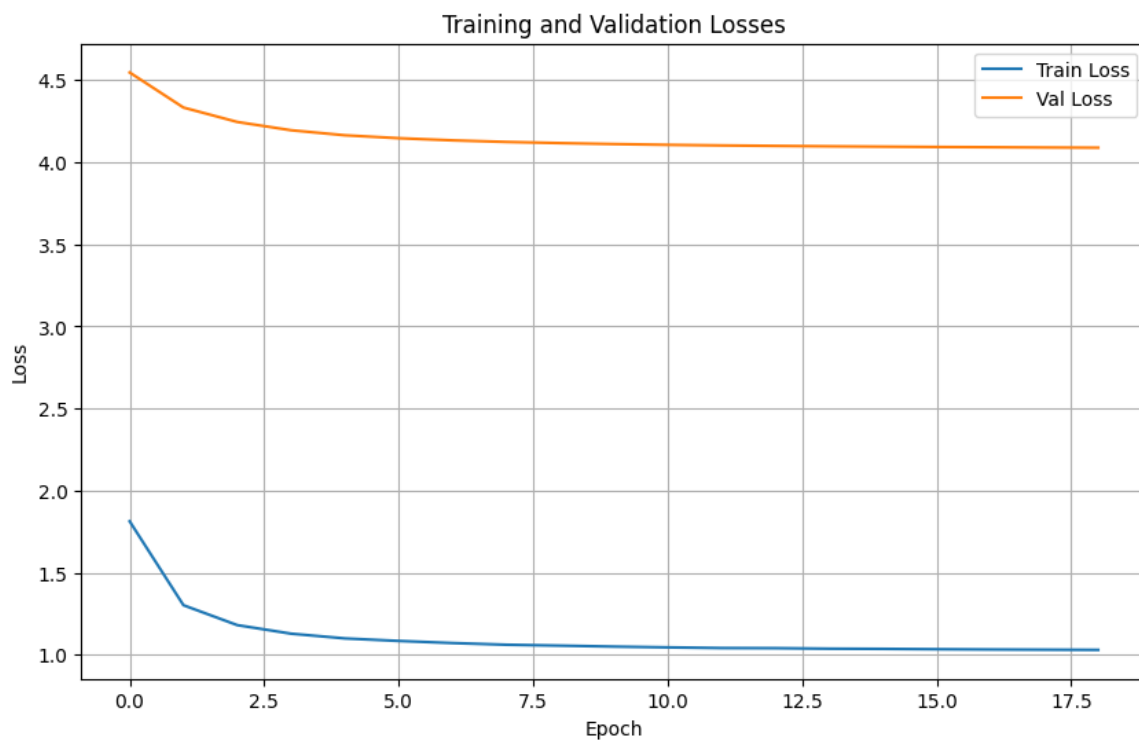
```

print("Early stopping triggered")
break

# Save the trained model
torch.save(model.state_dict(), 'model_loss_FDA_constrative.pth')

```

منحنی loss در این حالت به صورت زیر می باشد.

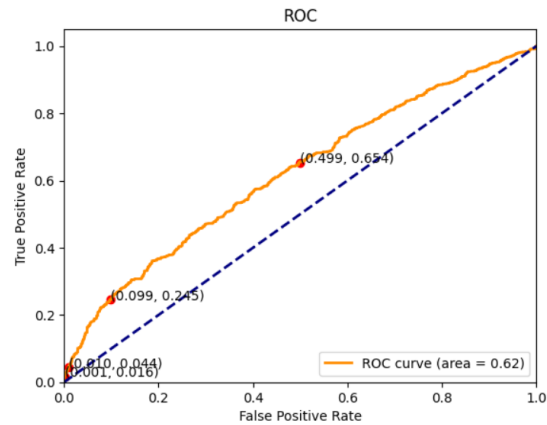


شکل ۱۶: منحنی loss برای FDC

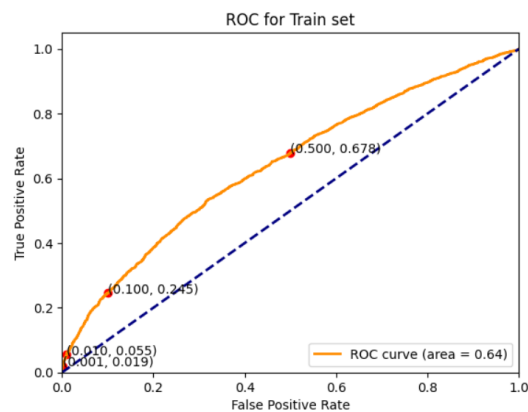
همانطور که مشاهده میشود سیر نزولی ای در loss مشاهده میشود.

- منحنی ROC برای FDC

منحنی زیر منحنی ROC برای FDC Loss را نشان میدهد. که مساحت زیر آن ۶۲ بوده و عملکرد بهتری نسبت به **contrastive loss** عادی دارد ولی همچنان با اختلاف عملکرد **triplet loss** بهتر میباشد.



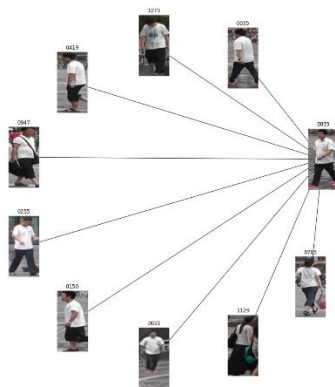
شکل ۱۷: منحنی ROC برای FCD loss برای داده های valid



شکل ۱۸ - منحنی ROC برای FCD loss برای داده های train

- TOP 10 برای FDC

خروجی یک نمونه برای این حالت به صورت زیر می باشد. که نشان از عملکرد به مراتب بهتر این حالت نسبت به **contrastive loss** دارد.



شکل ۱۹: خروجی Top 10 برای FDC loss

FISHER DISCRIMINANT TRIPLET LOSS

تعریف

(FDT) یک تابع $loss$ است که برای آموزش شبکه‌های سیامی طراحی شده است و از تجزیه و تحلیل تشخیصی فیشر (FDA) الهام می‌گیرد. FDT نمونه‌های $anchor$ ، مثبت (همسایه) و منفی (دور) را برای به حداکثر رساندن قابلیت تفکیک کلاس‌های مختلف در نظر می‌گیرد. بر خلاف $loss\ triplet$ ، که بر معیارهای فاصله تکیه می‌کند، تلفات FDT از معیار فیشر برای بهینه‌سازی مستقیم قابلیت تفکیک کلاس‌ها استفاده می‌کند. به طور خاص، هدف FDT افزایش پراکندگی بین طبقاتی (تغییرپذیری بین کلاس‌های مختلف) و در عین حال کاهش پراکندگی درون کلاسی (تغییرپذیری در همان کلاس)، با هدف بهبود توانایی تمایز فضای $embedding$ توسط شبکه است. این امر در تابع $loss$ با ردیابی ماتریس‌های پراکندگی از داده‌های پیش‌بینی شده، تنظیم تعادل بین پراکندگی بین کلاسی و درون کلاسی با یک فرایارامتر λ و اعمال حاشیه با پارامتر α رسمیت می‌یابد. نشان داده شده است که از دست دادن FDT در آزمایش‌های مختلف، به ویژه در استخراج ویژگی و وظایف یادگیری متریک، مانند تشخیص تصویر، که در ایجاد جاسازی‌های متمایزتر از $triplet\ loss$ سنتی بهتر عمل می‌کند، کارآمد است.

فرمول FDT به صورت زیر می‌باشد.

$$l_{fdt} = [(2 - \lambda)tr(U^T S_W U) - \lambda tr(U^T S_B U) + \alpha]_+$$

نکته حایز اهمیت در فرمول بندی FDT فرم $triplet$ بودن و بردن همش در بخش مثبت گیری میباشد اما در مورد دیتاست پیش رو با $margin\ 1$ این لاس با FCD خیلی فرقی ندارد و همانطور که در ادامه خواهید دید $result$ های FCD و FDT تقریباً یکسان میباشد.

پیاده سازی FDT

به راحتی با توجه به فرمول فوق FDT را پیاده سازی میکنیم (در کد کامنت گذاشته شده است)

منظور از `secondToLast` همان `anchor` و منظور از ۲ تای دیگر همان `positive` و `negative` ها میباشد.

```
class FisherTripletLoss(nn.Module):
    def __init__(self, emb_size=512, margin_in_loss=1.0):
        super(FisherTripletLoss, self).__init__()
        self.margin_in_loss = margin_in_loss
```

```

self.epsilon_Sw, self.epsilon_Sb = 0.0001, 0.0001
self.lambda_ = 0.01

def forward(self, o1_secondToLast, o2_secondToLast, o3_secondToLast,
weights_lastLayer):
    # Calculation of within scatter:
    temp1 = o1_secondToLast - o2_secondToLast
    S_within = torch.matmul(temp1.t(), temp1)

    # Calculation of between scatter:
    temp2 = o1_secondToLast - o3_secondToLast
    S_between = torch.matmul(temp2.t(), temp2)

    # Strengthen main diagonal of S_within and S_between:
    I_matrix = torch.eye(S_within.shape[0], device=o1_secondToLast.device)
    I_matrix_Sw = self.epsilon_Sw * I_matrix
    I_matrix_Sb = self.epsilon_Sb * I_matrix
    S_within = S_within + I_matrix_Sw
    S_between = S_between + I_matrix_Sb

    # Calculation of variance of projection considering within scatter:
    temp3 = torch.matmul(torch.matmul(weights_lastLayer.t(), S_within),
weights_lastLayer)
    within_scatter_term = torch.trace(temp3)

    # Calculation of variance of projection considering between scatter:
    temp4 = torch.matmul(torch.matmul(weights_lastLayer.t(), S_between),
weights_lastLayer)
    between_scatter_term = torch.trace(temp4)

    # Calculation of loss:
    loss_ = self.margin_in_loss + ((2 - self.lambda_) * within_scatter_term)
- (self.lambda_ * between_scatter_term)
    loss = torch.max(torch.tensor(0.0).to(o1_secondToLast.device), loss_)
    return loss

```

- دیتالودر

برای این بخش از همان دیتالودر آماده شده برای بخش FDC استفاده میشود.

- حلقه آموزش

به کمک کد زیر مدل را آموزش میدهیم.

```

# Training loop
num_epochs = 20

```



```

best_val_loss = float('inf')
early_stopping_patience = 5
no_improvement_count = 0

# Lists to store the losses
train_losses = []
val_losses = []

# Training loop
for epoch in range(num_epochs):
    model.train()
    total_loss = 0.0
    progress_bar = tqdm(train_loader, desc=f'Epoch {epoch+1}/{num_epochs}',
leave=False)

    for batch in progress_bar:
        anchor_images, positive_images, negative_images = batch
        anchor_images, positive_images, negative_images =
anchor_images.to(device), positive_images.to(device), negative_images.to(device)

        # Forward pass
        anchor_embeddings = model(anchor_images)
        positive_embeddings = model(positive_images)
        negative_embeddings = model(negative_images)

        # Convert embeddings to the same device as the model's weight
        anchor_embeddings = anchor_embeddings.to(device)
        positive_embeddings = positive_embeddings.to(device)
        negative_embeddings = negative_embeddings.to(device)

        # Compute Fisher triplet loss
        loss = triplet_loss(anchor_embeddings, positive_embeddings,
negative_embeddings, model.efficientnet.classifier[-1].weight)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        progress_bar.set_postfix({'Train Loss': total_loss / len(progress_bar)})

# Calculate average training loss for this epoch
avg_train_loss = total_loss / len(train_loader)
train_losses.append(avg_train_loss)

```

```

# Validation loop
model.eval()
val_loss = 0.0
with torch.no_grad():
    for batch in progress_bar:
        anchor_images, positive_images, negative_images = batch
        anchor_embeddings, positive_embeddings, negative_embeddings =
anchor_images.to(device), positive_images.to(device), negative_images.to(device)

        # Forward pass
        anchor_embeddings = model(anchor_images)
        positive_embeddings = model(positive_images)
        negative_embeddings = model(negative_images)

        # Convert embeddings to the same device as the model's weight
        anchor_embeddings = anchor_embeddings.to(device)
        positive_embeddings = positive_embeddings.to(device)
        negative_embeddings = negative_embeddings.to(device)

        # Compute Fisher triplet loss
        loss = triplet_loss(anchor_embeddings, positive_embeddings,
negative_embeddings, model.efficientnet.classifier[-1].weight)

        val_loss += loss.item()

# Calculate average validation loss for this epoch
avg_val_loss = val_loss / len(val_loader)
val_losses.append(avg_val_loss)

print(f'Epoch [{epoch+1}/{num_epochs}] | Train Loss: {avg_train_loss:.4f} |
Val Loss: {avg_val_loss:.4f}')

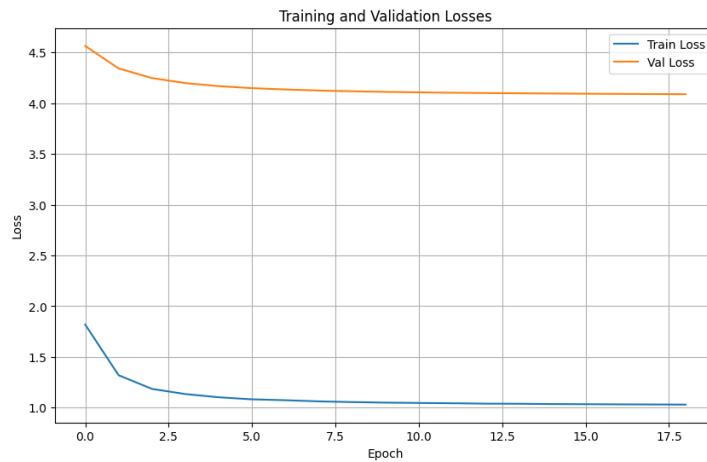
if avg_val_loss < best_val_loss:
    best_val_loss = avg_val_loss
    no_improvement_count = 0
else:
    no_improvement_count += 1

if no_improvement_count >= early_stopping_patience:
    print("Early stopping triggered")
    break

# Save the trained model
torch.save(model.state_dict(), 'model_loss_FDA_triplet.pth')

```

منحنی loss به صورت زیر می باشد.



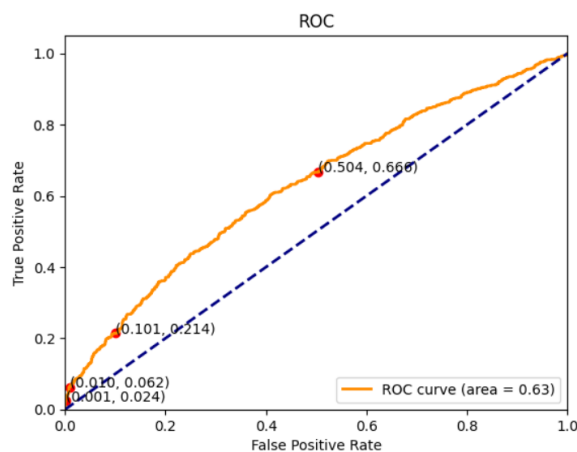
شکل ۲۰: منحنی loss برای FDT

همانطور که مشاهده میشود loss سیر نزولی دارد ولی به حد triplet نمی رسد و همانطور که در مقدمه این بخش توضیح داده شد تقریباً با loss FDC برابر می باشد.

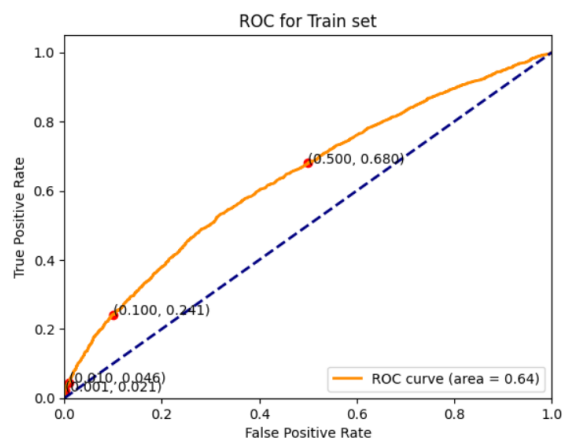
به صورت نسبی نتایج اندکی از FDC بهتر می باشد.

- منحنی ROC برای FDT

منحنی ROC برای FDT به صورت زیر می باشد (همانطور که انتظار میرفت شبیه FDC می باشد و فقط ۱ درصد بهتر می باشد)



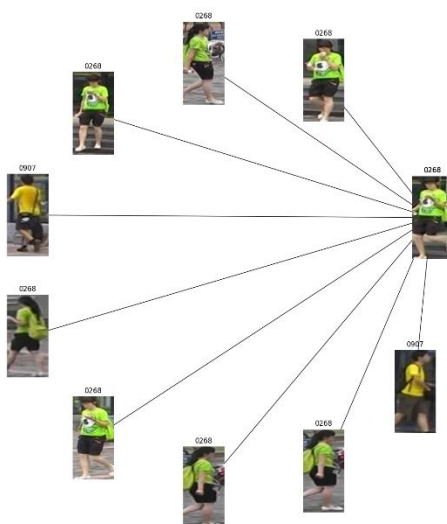
شکل ۲۱: منحنی ROC برای FCD loss برای داده های valid



شکل ۲۲ - منحنی ROC برای FCD loss برای داده های train

- TOP 10 برای FDT

تصویر یک نمونه از top 10 را برای FDT مشخص میکند که تقریباً عملکردی شبیه به FDC دارد.



شکل ۲۳ : top 10 برای FDT

آنالیز و نتیجه گیری

TPR به ازای marginهای مختلف برای داده های train-validation به شرح زیر است:

جدول ۷ - TPR به ازای lossهای مختلف برای داده های validation

False Positive Rate	Triplet loss	contrastive loss	FDC	FDT
0.5	۰/۹۸۹	0.74	0.654	0.666
0.1	۰/۹۱	0.39	0.245	0.214
0.01	۰/۵۱۷	0.09	0.044	0.062
0.001	۰/۱۱۱	0	0.016	0.024

جدول ۸ - TPR به ازای lossهای مختلف برای داده های train

False Positive Rate	Triplet loss	contrastive loss	FDC	FDT
0.5	0.997	0.75	0.678	0.68
0.1	0.929	0.31	0.245	0.241
0.01	0.539	0.09	0.055	0.046
0.001	0.179	0.03	0.019	0.021

درک زمینه و کاربرد خاصی که برای آن از این loss استفاده می کنید، مهم است. هر کدام نقاط قوت و ضعف خاص خود را دارند و اثربخشی آنها بسته به ماهیت مشکل و داده ها می تواند متفاوت باشد.

- Fisher Discriminant Contrastive (FDC) و Fisher Discriminant Triplet (FDT)

اینها به ترتیب انواع خاصی از توابع از triplet loss و contrastive هستند که اصولی را از تجزیه و تحلیل تفکیک فیشر ترکیب می کنند. هدف در اینجا نه تنها یادگیری تعبیه‌هایی است که از اهداف triplet یا contrastive استاندارد پیروی می کنند، بلکه همچنین به حداکثر رساندن واریانس بین طبقاتی در حالی که واریانس درون کلاسی را به حداقل می‌رسانند، یک اصل کلیدی در تحلیل تفکیک‌کننده فیشر است. این می تواند در برخی موارد به ویژگی های تبعیض آمیز بیشتری منجر شود.

- Contrastive loss

توابع **contrastive loss** معمولاً در کارهای طبقه بندی باینری، به ویژه در شبکه های سیامی، برای یادگیری **embedding** ها استفاده می شود. با کاهش فاصله بین جفت های مشابه، تلفات کاهش می یابد و با کاهش فاصله بین جفت های غیرمشابه، تا یک حاشیه مشخص افزایش می یابد. به طور کلی ساده تر است و ممکن است در برخی سناریوها سریع تر همگرا شود، اما به صراحت روابط بین چندین مثال (بیش از دو) را در نظر نمی گیرد.

- Triplet loss

Triplet loss از نمونه ها (**anchor**، مثبت، منفی) را در نظر می گیرد و پیچیده تر است. باعث ایجاد حاشیه بین جفت های مثبت و منفی نسبت به **anchor** می شود. این می تواند منجر به یادگیری قوی تر **embedding** ها شود، زیرا فواصل نسبی را در **triplet** به جای مجزا در نظر می گیرد.

- حساسیت به حاشیه

هر دو تابع **contrastive** و **triplet** نسبت به انتخاب حاشیه حساس هستند، اما حساسیت آنها ممکن است به دلیل تفاوت های ساختاری آنها متفاوت باشد:

در **triplet loss**، حاشیه تعیین می کند که مثال منفی در مقایسه با مثال مثبت چقدر باید از **anchor** فاصله داشته باشد.

در **contrastive loss**، حاشیه معمولاً نقطه برشی را تعیین می کند که فراتر از آن جفت های غیرمشابه به **loss** کمک نمی کنند.

❖ کدام بهتر است؟

وابسته به موضوع: اثربخشی این توابع از **loss** به شدت به برنامه خاص بستگی دارد. برای برخی از کارها، سادگی از دست دادن کنتراست ممکن است کافی باشد، در حالی که برای برخی دیگر، ماهیت نسبی از **triplet loss** یا قدرت تمایز اضافی **FDT/FDC** ممکن است سودمندتر باشد.

ویژگی های داده: ماهیت و مقدار داده های موجود نیز می تواند بر انتخاب تأثیر بگذارد. **Triplet** و انواع آن می توانند قدرتمندتر باشند، اما ممکن است برای یادگیری موثر روابط پیچیده به داده های بیشتری نیاز داشته باشند.

راندمان محاسباتی: تلفات **contrastive** معمولاً از نظر محاسباتی کمتر از **triplet loss** یا انواع فیشر آن است، زیرا شامل مقایسه‌های کمتری در هر دسته است.

سهولت در **train**: **triplet loss** و انواع آن ممکن است سخت‌تر باشد، زیرا برای اطمینان از یادگیری موثر، نیاز به انتخاب دقیق **triplets** دارند.

با توجه به موارد بالا در **triplet** و **FDT** بهترین نتایج را نسبت به **FDC** و **contrastive** گرفته ایم