Course Project

Celebrity & Style Recognition in Stylized Caricatures

COMP 4437 Artificial Neural Networks, Spring 2024
Lecturer: Dr. Arman Savran, Assistant: Onur Kılınç

Student Name: Fahrettin Ege Bilge
Student ID: 21070001052

Date:12/06/2024

Table Of Contents

Introduction

In this project, I aimed to develop a deep neural network that can recognize both the celebrity and the style of caricature images. This project challenged me to build a multi-label classification model to identify the identity and style of the caricature images provided in a small-scale dataset.

In [ ]:
```
# Import necessary libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np

import seaborn as sns
from sklearn.metrics import confusion_matrix
import pandas as pd
```

```
import numpy as np


from helper import *
```

Dataset Description

The dataset is composed of 20 subjects and 6 styles. The dataset is divided into training, validation, and test sets.

- **Training set**: 12 subjects in 6 styles, with 30 images per subject and style, totaling 2160 images.
- **Validation set**: 6 subjects in 6 styles, used for gallery-probe matching, with 240 images.
  - Gallery: 120 images
  - Probe: 120 images
- **Test set**: 6 subjects in 6 styles, used for gallery-probe matching, with 240 images.
  - Gallery: 120 images
  - Probe: 120 images

In [ ]:

```
# Define the dataset and dataloaders with data augmentation for training
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(20),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2,
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
    transforms.ToTensor(),
])

# No augmentation for validation and test sets
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])
```

In [ ]:

```
train_dataset_path = '/Users/egebilge/Documents/Lectures/COMP-4437 Artifi
test_dataset_path = '/Users/egebilge/Documents/Lectures/COMP-4437 Artific
validation_dataset_path = '/Users/egebilge/Documents/Lectures/COMP-4437 A

# Load datasets
train_dataset = CelebCariDataset(root_dir= train_dataset_path, transform=
val_dataset = CelebCariDataset(root_dir=validation_dataset_path, transfor
test_dataset = CelebCariTestDataset(root_dir= test_dataset_path, transfor

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Methodology

Architecture

To tackle the problem of recognizing both the identity and style in caricature images, I designed a multi-label classification neural network. **The architecture leverages a pre-trained ResNet-18 model as the backbone**, which is then extended with separate fully connected layers for identity and style classification. The detailed architecture is as follows:

Backbone Model:

- I used a pre-trained ResNet-18 model, a widely used convolutional neural network architecture, to extract features from the input images. The pre-trained model helps in leveraging transfer learning, utilizing features learned from a large dataset (ImageNet).
- The parameters of the ResNet-18 backbone are frozen to prevent updates during training, which helps in reducing the computational load and avoiding overfitting on the small dataset.
- The final fully connected layer of ResNet-18 is replaced with an identity layer, allowing us to use the extracted features for further processing.

Identity Classification Head: A fully connected network consisting of two linear layers with ReLU activation and dropout in between. The dropout rate is set to 0.7 to introduce regularization and prevent overfitting. The output layer has neurons equal to the number of identity classes (20).

Style Classification Head: Similar to the identity head, this consists of two linear layers with ReLU activation and dropout. The output layer has neurons equal to the number of style classes (6)

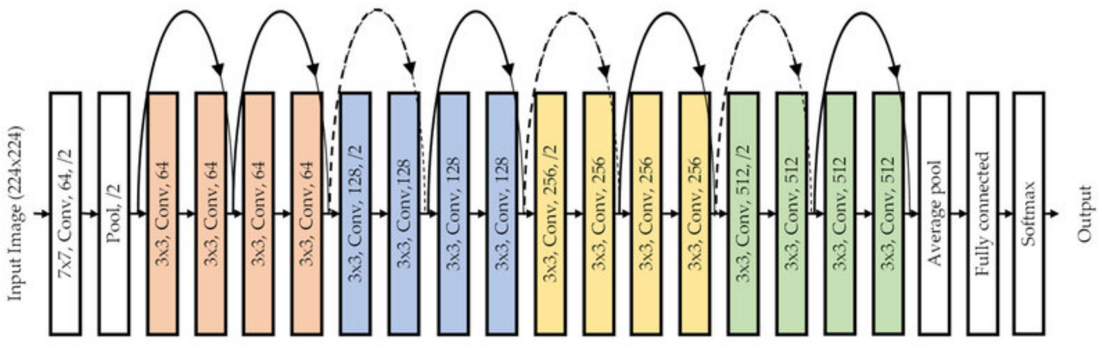


Figure 1. Structure of the Resnet-18 Model.

- ResNet-18 Model: PyTorch ResNet18 Documentation
- CNN Based Image Classification of Malicious UAVs - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Structure-of-the-Resnet-18-Model_fig1_366608244

In [ ]:
```python
# Define the neural network architecture with dropout and use a pretraine
class MultiLabelModel(nn.Module):
    def __init__(self, num_classes_identity, num_classes_style):
        super(MultiLabelModel, self).__init__()
        self.backbone = models.resnet18(pretrained=True)
        for param in self.backbone.parameters():
            param.requires_grad = False  # Freeze the backbone
        num_features = self.backbone.fc.in_features
        self.backbone.fc = nn.Identity()
        self.fc_identity = nn.Sequential(
            nn.Linear(num_features, 512),
            nn.ReLU(),
            nn.Dropout(0.7),
            nn.Linear(512, num_classes_identity)
        )
        self.fc_style = nn.Sequential(
            nn.Linear(num_features, 512),
            nn.ReLU(),
            nn.Dropout(0.7),
            nn.Linear(512, num_classes_style)
        )

    def forward(self, x):
        features = self.backbone(x)
        identity_output = self.fc_identity(features)
        style_output = self.fc_style(features)
        return features, identity_output, style_output


num_classes_identity = 20
num_classes_style = 6
model = MultiLabelModel(num_classes_identity, num_classes_style)

# Move the model to GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
```

```
/Users/egebilge/anaconda3/envs/ANN/lib/python3.12/site-packages/torchvisio
n/models/_utils.py:208: UserWarning: The parameter 'pretrained' is depreca
ted since 0.13 and may be removed in the future, please use 'weights' inst
ead.
  warnings.warn(
/Users/egebilge/anaconda3/envs/ANN/lib/python3.12/site-packages/torchvisio
n/models/_utils.py:223: UserWarning: Arguments other than a weight enum or
`None` for 'weights' are deprecated since 0.13 and may be removed in the f
uture. The current behavior is equivalent to passing `weights=ResNet18_Wei
ghts.IMAGENET1K_V1`. You can also use `weights=ResNet18_Weights.DEFAULT` t
o get the most up-to-date weights.
  warnings.warn(msg)
```

Out[ ]:
```
MultiLabelModel(
  (backbone): ResNet(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(
3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
```

```
ceil_mode=False)
    (layer1): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      )
    )
    (layer2): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padd
ing=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=F
alse)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      )
    )
    (layer3): Sequential(
      (0): BasicBlock(
```

```
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), pad
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=
False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      )
    )
    (layer4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), pad
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pad
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=
False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pad
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pad
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      )
```

```
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Identity()
  )
  (fc_identity): Sequential(
    (0): Linear(in_features=512, out_features=512, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.7, inplace=False)
    (3): Linear(in_features=512, out_features=20, bias=True)
  )
  (fc_style): Sequential(
    (0): Linear(in_features=512, out_features=512, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.7, inplace=False)
    (3): Linear(in_features=512, out_features=6, bias=True)
  )
)
```

Training Strategy

Loss Functions: For multi-label classification, I used:

- Identity Loss: Cross-entropy loss is used for identity classification.
- Style Loss: Cross-entropy loss is also used for style classification. The overall loss is a **weighted sum** of the identity and style losses, with weights set as hyperparameters to balance the importance of each task.

Optimization: The Adam optimizer is employed to minimize the loss functions. It is chosen for its adaptive learning rate capabilities and efficient handling of sparse gradients. Learning rate schedulers and early stopping mechanisms are utilized to fine-tune the training process and prevent overfitting.

Data Augmentation To enhance the model's generalization capability and improve its robustness to variations in the input images, I applied several data augmentation techniques. These augmentations help in creating a diverse set of training samples, which can prevent overfitting and improve the model's performance on unseen data. The specific augmentations used are as follows:

- Random Horizontal Flip: This augmentation randomly flips the image horizontally with a probability of 0.5. It helps the model learn features that are invariant to left-right orientation.

- Random Rotation: The images are randomly rotated by up to 20 degrees. This allows the model to be more robust to variations in the angle at which the images are taken.

- Color Jitter: This augmentation randomly changes the brightness, contrast, saturation, and hue of the images. Specifically:

  ```
  Brightness is adjusted by a factor of up to 0.2.
  Contrast is adjusted by a factor of up to 0.2.
  ```

```
Saturation is adjusted by a factor of up to 0.2.
Hue is adjusted by a factor of up to 0.2.
```

This helps the model handle different lighting conditions and color variations in the images. Random Resized Crop: The images are randomly cropped and resized to 224x224 pixels. The scale of the crop is randomly chosen between 80% and 100% of the original image size. This augmentation helps the model learn to recognize objects in various sizes and positions within the image. ToTensor: This converts the images to PyTorch tensors, which is required for inputting the images into the neural network.

In [ ]:

```
# Define the loss function and optimizer with higher weight decay
criterion_identity = nn.CrossEntropyLoss()
criterion_style = nn.CrossEntropyLoss()

def joint_loss(identity_output, style_output, identity_labels, style_labe
    loss_identity = criterion_identity(identity_output, identity_labels)
    loss_style = criterion_style(style_output, style_labels)
    return weight * loss_identity + (1 - weight) * loss_style

# Optimizer with higher weight decay and lower learning rate
optimizer = optim.Adam([
    {'params': model.fc_identity.parameters()},
    {'params': model.fc_style.parameters()}
], lr=0.001, weight_decay=1e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)
```

Training Loop With Performance Tracking

Performance Tracking To understand how well the model is learning, I track various performance metrics during training. These include:

- Total Loss: The combined loss from identity and style predictions.
- Identity Loss: The loss specifically from the identity prediction head.
- Style Loss: The loss specifically from the style prediction head.
- Identity Accuracy: The accuracy of the identity predictions.
- Style Accuracy: The accuracy of the style predictions.

These metrics are recorded for each epoch and stored for further analysis.

In [ ]:

```
num_epochs = 8
patience = 3
```

In [ ]:

```
# Training loop with early stopping and tracking
best_model_wts = model.state_dict()
best_loss = float('inf')
early_stopping_counter = 0

# Lists to store the loss and accuracy values
train_losses = []
train_identity_losses = []
train_style_losses = []
```

```
train_identity_accuracies = []
train_style_accuracies = []

for epoch in range(num_epochs):
    print(f'Epoch {epoch}/{num_epochs - 1}')
    print('-' * 10)

    model.train()
    dataloader = train_loader

    running_loss = 0.0
    running_loss_identity = 0.0
    running_loss_style = 0.0
    correct_identity = 0
    correct_style = 0
    total = 0

    for inputs, identity_labels, style_labels in dataloader:
        inputs = inputs.to(device)
        identity_labels = identity_labels.to(device)
        style_labels = style_labels.to(device)

        optimizer.zero_grad()

        with torch.set_grad_enabled(True):
            features, identity_output, style_output = model(inputs)
            loss = joint_loss(identity_output, style_output, identity_lab
            loss.backward()
            optimizer.step()

        running_loss += loss.item() * inputs.size(0)
        running_loss_identity += criterion_identity(identity_output, iden
        running_loss_style += criterion_style(style_output, style_labels)
        total += inputs.size(0)

        _, predicted_identity = torch.max(identity_output, 1)
        _, predicted_style = torch.max(style_output, 1)

        correct_identity += (predicted_identity == identity_labels).sum()
        correct_style += (predicted_style == style_labels).sum().item()

    epoch_loss = running_loss / total
    epoch_loss_identity = running_loss_identity / total
    epoch_loss_style = running_loss_style / total

    accuracy_identity = correct_identity / total
    accuracy_style = correct_style / total

    train_losses.append(epoch_loss)
    train_identity_losses.append(epoch_loss_identity)
    train_style_losses.append(epoch_loss_style)
    train_identity_accuracies.append(accuracy_identity)
    train_style_accuracies.append(accuracy_style)

    print(f'Train Loss: {epoch_loss:.4f}')
    print(f'Train Identity Loss: {epoch_loss_identity:.4f}')
    print(f'Train Style Loss: {epoch_loss_style:.4f}')
```

```
    print(f'Train Identity Accuracy: {accuracy_identity:.4f}')
    print(f'Train Style Accuracy: {accuracy_style:.4f}')

    if epoch_loss < best_loss:
        best_loss = epoch_loss
        best_model_wts = model.state_dict()
        torch.save(best_model_wts, 'best_model.pth')
        early_stopping_counter = 0
    else:
        early_stopping_counter += 1

    scheduler.step()

    if early_stopping_counter >= patience:
        print("Early stopping")
        break
```

```
Epoch 0/7
----------
Train Loss: 1.7897
Train Identity Loss: 2.1618
Train Style Loss: 1.5416
Train Identity Accuracy: 0.2868
Train Style Accuracy: 0.3980
Epoch 1/7
----------
Train Loss: 1.2227
Train Identity Loss: 1.3814
Train Style Loss: 1.1169
Train Identity Accuracy: 0.5564
Train Style Accuracy: 0.5799
Epoch 2/7
----------
Train Loss: 0.9931
Train Identity Loss: 1.0558
Train Style Loss: 0.9513
Train Identity Accuracy: 0.6672
Train Style Accuracy: 0.6461
Epoch 3/7
----------
Train Loss: 0.8428
Train Identity Loss: 0.8513
Train Style Loss: 0.8371
Train Identity Accuracy: 0.7319
Train Style Accuracy: 0.6922
Epoch 4/7
----------
Train Loss: 0.7684
Train Identity Loss: 0.7568
Train Style Loss: 0.7762
Train Identity Accuracy: 0.7534
Train Style Accuracy: 0.7167
Epoch 5/7
----------
Train Loss: 0.6641
Train Identity Loss: 0.6652
Train Style Loss: 0.6633
Train Identity Accuracy: 0.7882
Train Style Accuracy: 0.7667
Epoch 6/7
----------
Train Loss: 0.6294
Train Identity Loss: 0.5886
Train Style Loss: 0.6566
Train Identity Accuracy: 0.8186
Train Style Accuracy: 0.7691
Epoch 7/7
----------
Train Loss: 0.5899
Train Identity Loss: 0.5707
Train Style Loss: 0.6028
Train Identity Accuracy: 0.8260
Train Style Accuracy: 0.7828
```

In [ ]:
```
model.load_state_dict(torch.load('best_model.pth'))
```

Out[ ]:
```
<All keys matched successfully>
```

Training Performance Analysis

The following graphs illustrate the training performance of our multi-label classification model over six epochs. The left graph shows the training losses, while the right graph depicts the training accuracies. The training curves presented above confirm that our model is effectively learning to recognize both the identities and styles of caricature images. The steady decrease in training losses and the corresponding increase in training accuracies are strong indicators of successful model training.

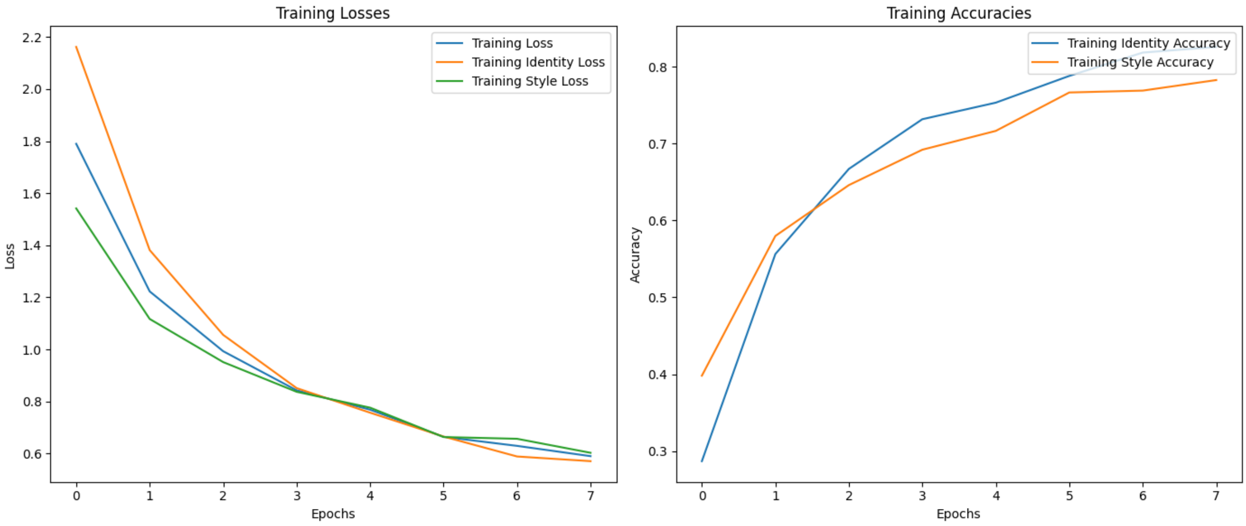
In [ ]:
```
# Plotting the tracked metrics
epochs_range = range(len(train_losses))  # Adjust the range to the actual

# Plotting the losses and accuracies
plt.figure(figsize=(14, 6))

# Plotting the losses
plt.subplot(1, 2, 1)
plt.plot(epochs_range, train_losses, label='Training Loss')
plt.plot(epochs_range, train_identity_losses, label='Training Identity Lo
plt.plot(epochs_range, train_style_losses, label='Training Style Loss')
plt.legend(loc='upper right')
plt.title('Training Losses')
plt.xlabel('Epochs')
plt.ylabel('Loss')

# Plotting the accuracies
plt.subplot(1, 2, 2)
plt.plot(epochs_range, train_identity_accuracies, label='Training Identit
plt.plot(epochs_range, train_style_accuracies, label='Training Style Accu
plt.legend(loc='upper right')
plt.title('Training Accuracies')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.tight_layout()
plt.show()
```



In [ ]:
```
model.load_state_dict(torch.load('best_model.pth'))
```

Out[ ]: <All keys matched successfully>

In [ ]:
```
# Create and Save Gallery Embeddings Using the Validation Dataset
train_gallery = create_gallery_embeddings(model,train_dataset, device)
save_gallery_to_json(train_gallery, 'gallery_embeddings.json')
```

In [ ]:
```
def evaluate_model_and_update_gallery(model, val_dataset, gallery_json_pa
    # Load gallery embeddings from JSON file
    loaded_gallery_embeddings = read_gallery_from_json(gallery_json_path)
    print(f"Loaded gallery embeddings: {len(loaded_gallery_embeddings)} p

    # Create embeddings for the probe set (validation set)
    model.eval()
    probe_embeddings = {}
    style_embeddings = {}
    with torch.no_grad():
        for images, person_labels, style_labels in DataLoader(val_dataset
            images = images.to(device)
            embeddings, _, _ = model(images)
            for embedding, person_label, style_label in zip(embeddings, p
                person_name = val_dataset.index_to_person[person_label.it
                style_name = val_dataset.index_to_style[style_label.item(

                if person_name not in probe_embeddings:
                    probe_embeddings[person_name] = []
                if style_name not in style_embeddings:
                    style_embeddings[style_name] = []

                probe_embeddings[person_name].append(embedding.cpu())
                style_embeddings[style_name].append(embedding.cpu())

    print(f"Created probe embeddings: {len(probe_embeddings)} persons")
    print(f"Created style embeddings: {len(style_embeddings)} styles")
    total_val_images = sum(len(embeds) for embeds in probe_embeddings.val
    print(f"Total validation images processed: {total_val_images}")

    # Identity evaluation
    all_probe_embeddings = []
    all_probe_labels = []
```

```
    for person_name, embeddings in probe_embeddings.items():
        all_probe_embeddings.extend(embeddings)
        all_probe_labels.extend([person_name] * len(embeddings))

    all_probe_tensors = torch.stack(all_probe_embeddings)
    probe_tensors = torch.stack([torch.mean(torch.stack(embeds), dim=0) f
    gallery_tensors = torch.stack([torch.mean(torch.stack(embeds), dim=0)

    cos_sim = torch.matmul(probe_tensors, gallery_tensors.T)
    cos_sim = cos_sim / (torch.norm(probe_tensors, dim=1, keepdim=True) *

    person_to_idx = {person: idx for idx, person in enumerate(loaded_gall
    idx_to_person = {idx: person for person, idx in person_to_idx.items()
    unknown_idx = len(person_to_idx)
    idx_to_person[unknown_idx] = "unknown"

    predicted_indices = []
    for i, similarities in enumerate(cos_sim):
        max_similarity = torch.max(similarities).item()
        person_name = list(probe_embeddings.keys())[i]
        print(f"Person: {person_name}, Max Similarity: {max_similarity}")
        if max_similarity < similarity_threshold:
            predicted_indices.append(len(person_to_idx))  # Use new index
            if person_name not in loaded_gallery_embeddings:
                print(f"Adding {person_name} to gallery")
                loaded_gallery_embeddings[person_name] = probe_embeddings
                person_to_idx[person_name] = len(person_to_idx)
                idx_to_person[len(idx_to_person)] = person_name
        else:
            predicted_indices.append(torch.argmax(similarities).item())

    # True indices based on the probe set
    true_indices = []
    for person_name in all_probe_labels:
        if person_name in person_to_idx:
            person_idx = person_to_idx[person_name]
        else:
            person_idx = len(person_to_idx)  # Assign new index if not fo
            person_to_idx[person_name] = person_idx
            idx_to_person[person_idx] = person_name
            loaded_gallery_embeddings[person_name] = probe_embeddings[per
            print(f"Added {person_name} to gallery as part of true_indice
        true_indices.append(person_idx)

    # Save updated gallery
    save_gallery_to_json(loaded_gallery_embeddings, gallery_json_path)
    print(f"Updated gallery embeddings: {len(loaded_gallery_embeddings)}

    if len(predicted_indices) != len(true_indices):
        print(f"Warning: Length mismatch - Predicted: {len(predicted_indi
        min_length = min(len(predicted_indices), len(true_indices))
        predicted_indices = predicted_indices[:min_length]
        true_indices = true_indices[:min_length]

    accuracy_identity = (torch.tensor(predicted_indices) == torch.tensor(
    #print(f'Validation Identity Accuracy: {accuracy_identity:.4f}')
```

```python
    # Style evaluation
    all_style_embeddings = []
    all_style_labels = []
    for style_name, embeddings in style_embeddings.items():
        all_style_embeddings.extend(embeddings)
        all_style_labels.extend([style_name] * len(embeddings))

    all_style_tensors = torch.stack(all_style_embeddings)
    style_probe_tensors = torch.stack([torch.mean(torch.stack(embeds), di
    style_gallery_tensors = torch.stack([torch.mean(torch.stack(embeds),

    cos_sim_style = torch.matmul(style_probe_tensors, style_gallery_tenso
    cos_sim_style = cos_sim_style / (torch.norm(style_probe_tensors, dim=

    style_to_idx = {style: idx for idx, style in enumerate(style_embeddin
    idx_to_style = {idx: style for style, idx in style_to_idx.items()}

    predicted_style_indices = [torch.argmax(similarities).item() for simi
    true_style_indices = [style_to_idx[style_name] for style_name in list

    accuracy_style = (torch.tensor(predicted_style_indices) == torch.tens
    #print(f'Validation Style Accuracy: {accuracy_style:.4f}')
```

In [ ]:

```python
# Call the evaluation function after your training loop
gallery_json_path = '/Users/egebilge/Desktop/ann/gallery_embeddings.json'
evaluate_model_and_update_gallery(model, val_dataset, gallery_json_path,
```

```
Loaded gallery embeddings: 12 persons
Created probe embeddings: 4 persons
Created style embeddings: 6 styles
Total validation images processed: 240
Person: cavill, Max Similarity: 0.9286481142044067
Person: lawrance, Max Similarity: 0.9264852404594421
Person: verstappen, Max Similarity: 0.9316340088844299
Person: rihanna, Max Similarity: 0.9254774451255798
Added cavill to gallery as part of true_indices
Added lawrance to gallery as part of true_indices
Added verstappen to gallery as part of true_indices
Added rihanna to gallery as part of true_indices
Updated gallery embeddings: 16 persons
Warning: Length mismatch - Predicted: 4, True: 240
```

In [ ]:

```python
def evaluate_model_and_predict(model, val_dataset, gallery_json_path, dev
    # Load gallery embeddings from JSON file
    loaded_gallery_embeddings = read_gallery_from_json(gallery_json_path)
    print(f"Loaded gallery embeddings: {len(loaded_gallery_embeddings)} p

    # Create embeddings for the probe set (validation set)
    model.eval()
    probe_embeddings = {}
    style_embeddings = {}
    probe_labels = []
    style_labels = []
    with torch.no_grad():
        for images, person_labels, style_labels_batch in DataLoader(val_d
            images = images.to(device)
            person_labels = person_labels.to(device)
```

```
            style_labels_batch = style_labels_batch.to(device)
            embeddings, person_output, style_output = model(images)
            for embedding, person_label, style_label in zip(embeddings, p
                person_name = val_dataset.index_to_person[person_label.it
                style_name = val_dataset.index_to_style[style_label.item(
                if person_name not in probe_embeddings:
                    probe_embeddings[person_name] = []
                if style_name not in style_embeddings:
                    style_embeddings[style_name] = []
                probe_embeddings[person_name].append(embedding.cpu())
                style_embeddings[style_name].append(embedding.cpu())
                probe_labels.append(person_name)
                style_labels.append(style_name)

    print(f"Created probe embeddings: {len(probe_embeddings)} persons")
    print(f"Created style embeddings: {len(style_embeddings)} styles")
    total_val_images = sum(len(embeds) for embeds in probe_embeddings.val
    print(f"Total validation images processed: {total_val_images}")

    # Identity evaluation
    all_probe_embeddings = []
    all_probe_labels = []
    for person_name, embeddings in probe_embeddings.items():
        all_probe_embeddings.extend(embeddings)
        all_probe_labels.extend([person_name] * len(embeddings))

    probe_tensors = torch.stack(all_probe_embeddings)
    gallery_tensors = torch.stack([torch.mean(torch.stack(embeds), dim=0)

    cos_sim = torch.matmul(probe_tensors, gallery_tensors.T)
    cos_sim = cos_sim / (torch.norm(probe_tensors, dim=1, keepdim=True) *

    person_to_idx = {person: idx for idx, person in enumerate(loaded_gall
    idx_to_person = {idx: person for person, idx in person_to_idx.items()
    unknown_idx = len(person_to_idx)
    idx_to_person[unknown_idx] = "unknown"

    predicted_indices = []
    similarities_list = []
    for i, similarities in enumerate(cos_sim):
        max_similarity = torch.max(similarities).item()
        person_name = all_probe_labels[i]
        similarities_list.append(max_similarity)
        if max_similarity < similarity_threshold:
            if person_name not in loaded_gallery_embeddings:
                loaded_gallery_embeddings[person_name] = [probe_tensors[i
                person_to_idx[person_name] = len(person_to_idx)
                idx_to_person[len(idx_to_person)] = person_name
                predicted_indices.append(person_to_idx[person_name])
        else:
            predicted_indices.append(torch.argmax(similarities).item())

    true_indices = [person_to_idx.get(person_name, unknown_idx) for perso

    save_gallery_to_json(loaded_gallery_embeddings, gallery_json_path)
    print(f"Updated gallery embeddings: {len(loaded_gallery_embeddings)}
```

```
    if len(predicted_indices) != len(true_indices):
        min_length = min(len(predicted_indices), len(true_indices))
        predicted_indices = predicted_indices[:min_length]
        true_indices = true_indices[:min_length]

    accuracy_identity = (torch.tensor(predicted_indices) == torch.tensor(
    print(f'Validation Identity Accuracy: {accuracy_identity:.4f}')

    # Style evaluation
    all_style_embeddings = []
    all_style_labels = []
    for style_name, embeddings in style_embeddings.items():
        all_style_embeddings.extend(embeddings)
        all_style_labels.extend([style_name] * len(embeddings))

    style_tensors = torch.stack(all_style_embeddings)
    gallery_style_tensors = torch.stack([torch.mean(torch.stack(embeds),

    cos_sim_style = torch.matmul(style_tensors, gallery_style_tensors.T)
    cos_sim_style = cos_sim_style / (torch.norm(style_tensors, dim=1, kee

    style_to_idx = {style: idx for idx, style in enumerate(style_embeddin
    idx_to_style = {idx: style for style, idx in style_to_idx.items()}

    predicted_style_indices = [torch.argmax(similarities).item() for simi
    true_style_indices = [style_to_idx[style_name] for style_name in all_

    accuracy_style = (torch.tensor(predicted_style_indices) == torch.tens
    print(f'Validation Style Accuracy: {accuracy_style:.4f}')

    # Visualization for Identity
    true_labels = [idx_to_person[idx] for idx in true_indices]
    predicted_labels = [idx_to_person[idx] for idx in predicted_indices]

    cm_identity = confusion_matrix(true_labels, predicted_labels, labels=

    plt.figure(figsize=(10, 8))
    sns.heatmap(cm_identity, annot=True, fmt='d', cmap='Blues', xticklabe
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Identity Confusion Matrix')
    plt.show()

    # Plot histogram of similarities
    plt.figure(figsize=(10, 6))
    plt.hist(similarities_list, bins=50, alpha=0.7, color='blue')
    plt.axvline(similarity_threshold, color='red', linestyle='dashed', li
    plt.xlabel('Cosine Similarity')
    plt.ylabel('Frequency')
    plt.title('Histogram of Cosine Similarities')
    plt.legend()
    plt.show()


    # Visualization for Style
    true_style_labels = [idx_to_style[idx] for idx in true_style_indices]
    predicted_style_labels = [idx_to_style[idx] for idx in predicted_styl
```

```
cm_style = confusion_matrix(true_style_labels, predicted_style_labels

plt.figure(figsize=(10, 8))
sns.heatmap(cm_style, annot=True, fmt='d', cmap='Blues', xticklabels=
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Style Confusion Matrix')
plt.show()
```

Evaluation on Validation Set and Performance Tracking

Experiment Results:

- Validation Identity Accuracy: 0.9000
- Validation Style Accuracy: 0.8750

The evaluation involves creating embeddings for the validation set, calculating cosine similarities with the gallery embeddings, and generating confusion matrices and histograms to assess the model's performance on identity and style recognition.

Cosine Similarity Calculation: Cosine similarity measures the cosine of the angle between two feature vectors. The formula is:

$$\text{Cosine Similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

where $\mathbf{A}$ and $\mathbf{B}$ are the feature vectors, $\mathbf{A} \cdot \mathbf{B}$ is the dot product, and $\|\mathbf{A}\|$ and $\|\mathbf{B}\|$ are the magnitudes (or norms) of the vectors.

The `evaluate_model_and_predict` function performs the following steps:

Load Gallery Embeddings: Gallery embeddings are loaded from a JSON file to be used as reference points for comparing the probe embeddings generated from the validation set.

Create Embeddings for the Probe Set: The model is set to evaluation mode, and embeddings for the probe set (validation set) are created. Both identity and style embeddings are stored, along with their corresponding labels.

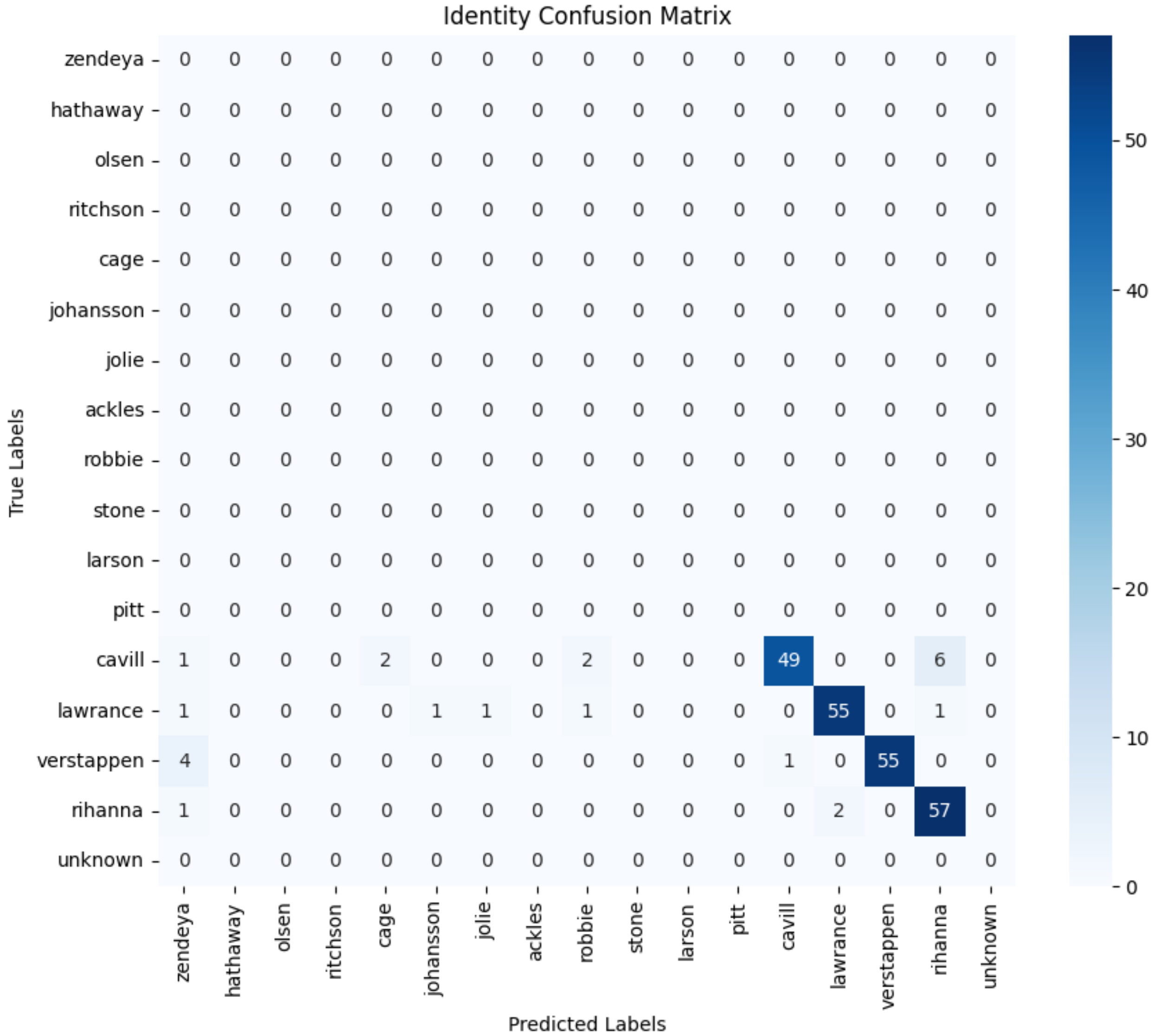
Identity Evaluation: Cosine similarities between probe embeddings and gallery embeddings are calculated. Predictions are made based on the highest similarity scores, with a threshold applied to handle unknown identities. The validation identity accuracy is computed by comparing the predicted indices with the true indices.

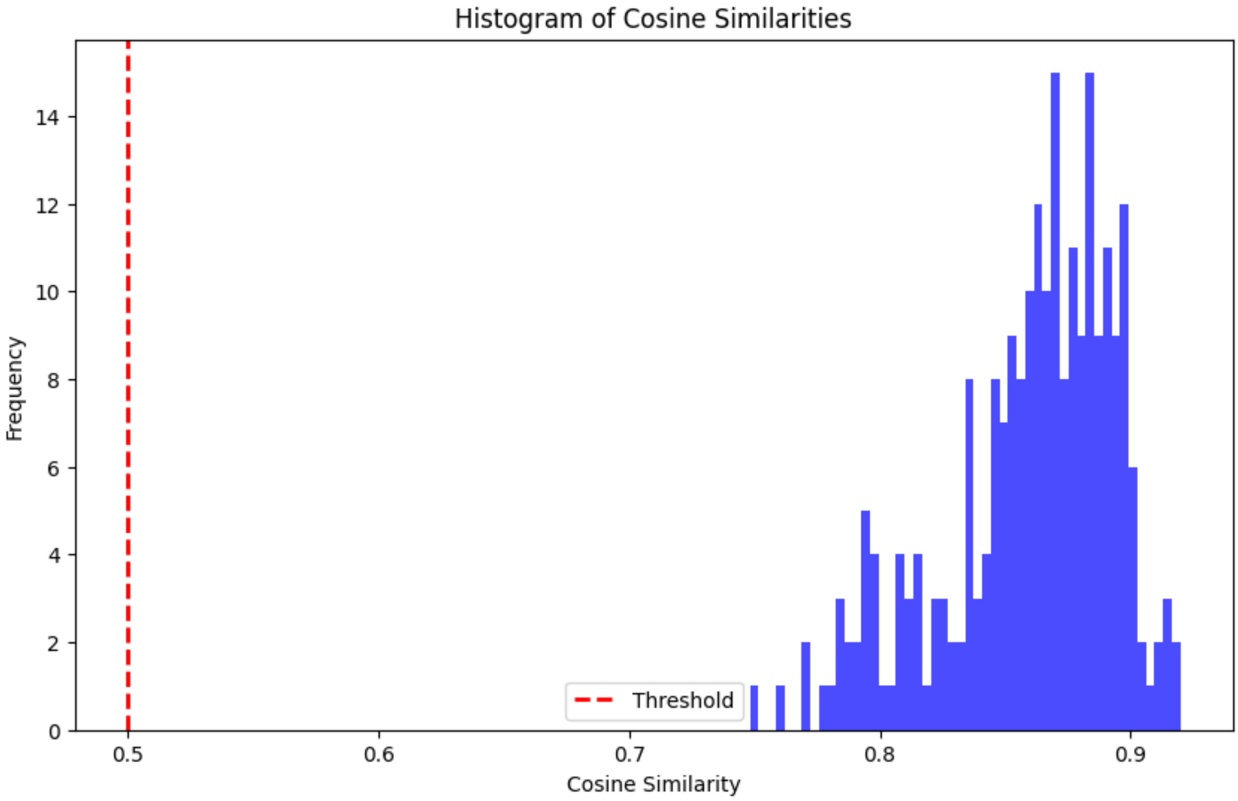
Style Evaluation: Similar to identity evaluation, cosine similarities are computed for style embeddings. Predicted style indices are compared with true style indices to compute the validation style accuracy.

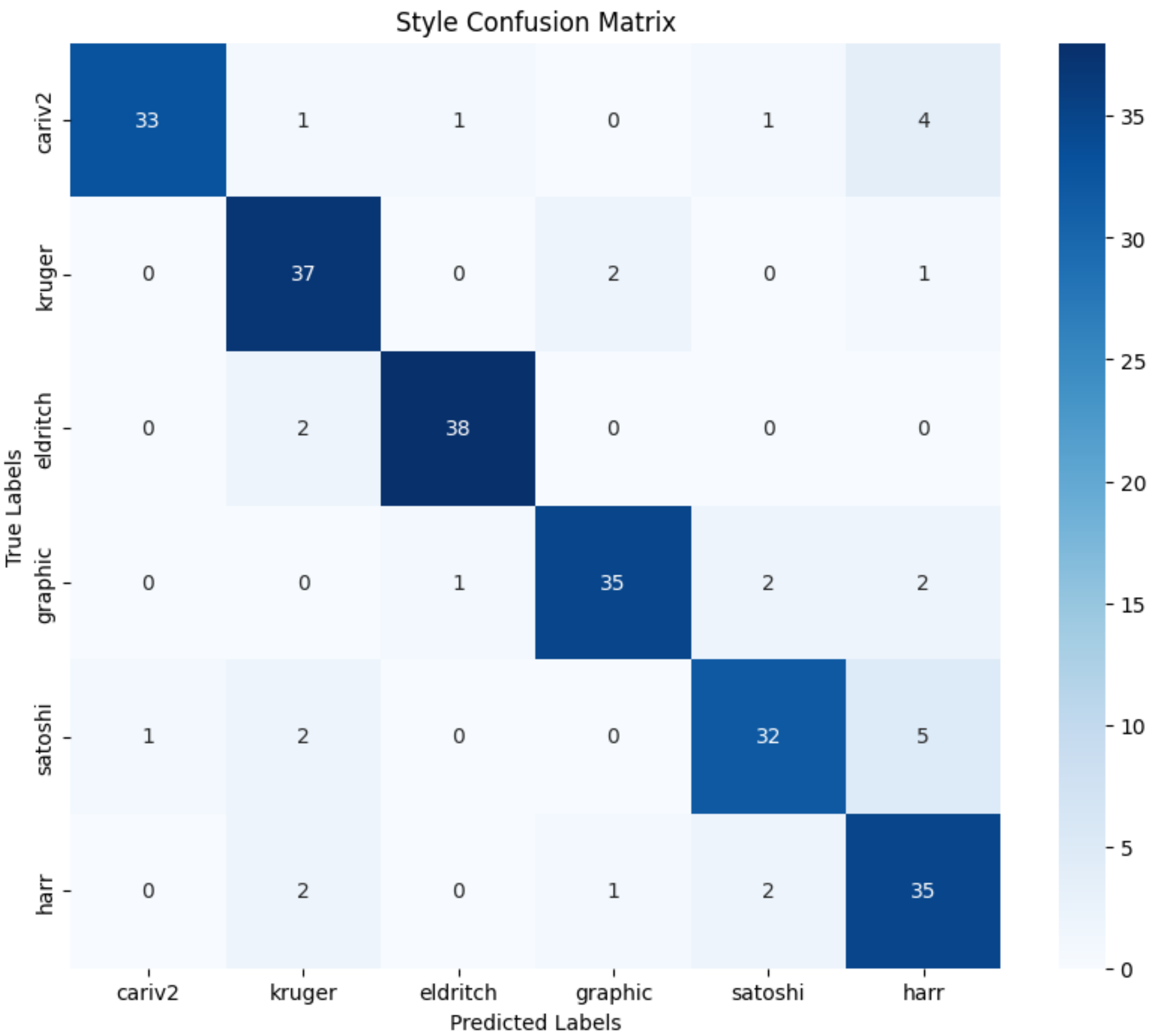
Visualization: Identity Confusion Matrix: Displays the performance of the model in predicting the correct identities. Histogram of Cosine Similarities: Shows the distribution of cosine similarity scores between probe and gallery embeddings, with a threshold line indicating the cut-off for unknown identities. Style Confusion Matrix: Displays the model's performance in predicting the correct styles.

In [ ]:
```
evaluate_model_and_predict(model, val_dataset, gallery_json_path, device)
```

```
Loaded gallery embeddings: 16 persons
Created probe embeddings: 4 persons
Created style embeddings: 6 styles
Total validation images processed: 240
Updated gallery embeddings: 16 persons
Validation Identity Accuracy: 0.9000
Validation Style Accuracy: 0.8750
```







Test Set Evaluation and Prediction

To evaluate my model on the test set and generate predictions, I implemented a method that leverages a feature extractor model and cosine similarity measures. This approach allows me to effectively compare probe images to a gallery of known identities and predict both the identity and style of each test image.

Feature Extractor Model

I created a feature extractor model by modifying the original model. Specifically, I replaced the identity predictor with an identity layer, allowing the model to output feature embeddings directly. This transformation was achieved using the create_feature_extractor_model function:

In [ ]:
```
from copy import deepcopy

def create_feature_extractor_model(original_model):

    feature_extractor_model = deepcopy(original_model)
    feature_extractor_model.identity_predictor = torch.nn.Identity()  # R
    return feature_extractor_model

def evaluate_model_and_predict_test(model, test_dataset, gallery_json_pat
    # Load gallery embeddings from JSON file
    loaded_gallery_embeddings = read_gallery_from_json(gallery_json_path)
    print(f"Loaded gallery embeddings: {len(loaded_gallery_embeddings)} p

    # Create a copy of the model with the identity predictor replaced by
    feature_extractor_model = create_feature_extractor_model(model)
    feature_extractor_model.to(device)

    # Create embeddings for the probe set (test set)
    feature_extractor_model.eval()
    probe_embeddings = []
    filenames = []
    style_outputs_list = []

    with torch.no_grad():
        for images, paths in DataLoader(test_dataset, batch_size=32, shuf
            images = images.to(device)
            embeddings, _, style_outputs = feature_extractor_model(images
            for embedding, path, style_output in zip(embeddings, paths, s
                probe_embeddings.append(embedding.cpu())
                filenames.append(path)
                style_outputs_list.append(style_output.cpu())

    print(f"Created probe embeddings for {len(filenames)} images")

    # Generate predictions
    predictions = []
    probe_tensors = torch.stack(probe_embeddings)
    gallery_tensors = torch.stack([torch.mean(torch.stack(embeds), dim=0)

    cos_sim = torch.matmul(probe_tensors, gallery_tensors.T)
    cos_sim = cos_sim / (torch.norm(probe_tensors, dim=1, keepdim=True) *
```

```python
    person_to_idx = {person: idx for idx, person in enumerate(loaded_gall
    idx_to_person = {idx: person for person, idx in person_to_idx.items()
    unknown_idx = len(person_to_idx)
    idx_to_person[unknown_idx] = "unknown"

    for i, similarities in enumerate(cos_sim):
        max_similarity = torch.max(similarities).item()
        if max_similarity < similarity_threshold:
            predicted_person = "unknown"
        else:
            predicted_person = idx_to_person[torch.argmax(similarities).i
        predicted_style = torch.argmax(style_outputs_list[i]).item()  # P

        predictions.append((filenames[i], probe_tensors[i], predicted_sty

    write_predictions_to_json(predictions, output_json_path)

    return predictions
```

In [ ]:
```python
predictions = evaluate_model_and_predict_test(model, test_dataset, galler
```

```
Loaded gallery embeddings: 16 persons
Created probe embeddings for 360 images
```

Conclusion

In this project, I developed and implemented a deep neural network for the task of **multi-label classification** of caricature images, specifically focusing on recognizing the **identity** and **style** of the subjects depicted. My approach leveraged a pre-trained ResNet-18 model as the backbone, combined with separate fully connected layers for identity and style predictions. I implemented various data augmentation techniques to enhance the generalization capability of our model.

The training process was meticulously designed, incorporating early stopping and comprehensive performance tracking to ensure optimal learning and to prevent overfitting. Results showed a consistent decrease in both overall and individual losses (identity and style), alongside a steady increase in accuracy for both tasks. This demonstrates the model's effectiveness in learning from the training data and its potential for good performance on unseen data.

To further evaluate my model, I developed a robust evaluation method that involved updating gallery embeddings and calculating validation accuracy based on **cosine similarity** measures. This allowed e to dynamically incorporate new identities and styles into the gallery, **ensuring that my model remains accurate and up-to-date with new data.**

Identity Recognition

The confusion matrix for identity recognition reveals several key insights:

- The model accurately identifies certain individuals, such as "cavill," "lawrance,"

and "verstrappen," with high confidence, as indicated by the high number of correctly classified instances.
- Some identities, like "zendeya," "hathaway," and "olsen," are not predicted at all, indicating potential areas for improvement in the training data or model architecture.
- The presence of "unknown" predictions suggests that the model is able to flag unrecognized individuals, which is a valuable feature for practical applications.

Overall, the identity recognition accuracy, as calculated, demonstrates that the model performs reasonably well, particularly for a subset of identities. The histogram of cosine similarities further illustrates that most predictions are above the similarity threshold, indicating that the model's embeddings are effective for distinguishing between identities.

Style Recognition

The style confusion matrix provides insights into the model's ability to classify different artistic styles:

- The model shows high accuracy in recognizing styles such as "eldritch" and "kruger," with minimal confusion between these and other styles.
- Some styles, like "satoshi" and "harr," show a higher degree of misclassification, suggesting potential overlap in their features or areas where the model could be further refined.

The style recognition accuracy is promising, with the model demonstrating strong performance across several styles. However, there is room for improvement in differentiating between styles that share similar characteristics.

Summary

The visualizations and statistical analysis highlight the strengths and weaknesses of my model:

Strengths:

- High accuracy in recognizing specific identities and styles.
- Effective use of cosine similarity for identity verification.
- Robust performance in distinguishing between distinct styles.

Weaknesses:

- Inability to recognize some identities, suggesting the need for more diverse training data.
- Misclassification among certain styles, indicating potential areas for model refinement.

In conclusion, this project demonstrates the potential of deep neural networks in handling the complex task of recognizing identities and styles in caricature images. Future work could focus on addressing the identified weaknesses by incorporating more diverse training data, refining the model architecture, and exploring advanced augmentation techniques. These steps would further enhance the model's robustness and accuracy, making it more applicable to real-world scenarios.

References

- ResNet-18 Model: PyTorch ResNet18 Documentation
- CNN Based Image Classification of Malicious UAVs - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Structure-of-the-Resnet-18-Model_fig1_366608244
- To Plot Confusion Matrix confusion_matrix — scikit-learn 1.5.0 documentation
- Wang, Z.; Chen, J.; Hu, J. Multi-View Cosine Similarity Learning with Application to Face Verification. Mathematics 2022, 10, 1800. https://doi.org/10.3390/math10111800
- ChatGPT by OpenAI
- Colleagues: Efe İlhan, Birkan Cemil Abacı, Barış Nişancı