Group Project Report

Fashion Finder: Classify Clothing Categories with AI

Group-04

Meet Daxini

Deborah Aina

DATS-6303

Deep-Learning

Dr. Amir Jafari

05/03/2024

Contents

1 Introduction **3**

2 Data **4**

2.1 Dataset Used 4

2.2 Adaptation for Multi-label Classification 4

2.3 Splits 4

3 Modeling **5**

3.1 CNN 5

3.2 Resnet 5

3.3 Vision Transformer (ViT) 6

3.4 Inceptionnet 7

3.5 Training 7

3.5.1 Class Weight Calculation 8

3.5.2 Freeze-Unfreeze technique 9

3.5.3 FocalLoss Loss function 10

4 Performance Metrics **10**

5 App **12**

6 Conclusion **14**

7 Future Enhancements **14**

8 References **15**

A Appendix: Source Code **16**

A.1 Dataset Loading and Preprocessing Code 16

A.2 Class weights calculation Code 19

A.3 CNN Model Code 19

A.4 Resnet Model Code 21

A.5 ViT Model Code 21

A.6 Pytorch model training loop 21

1 Introduction

The global fashion industry is a colossal market, valued at approximately $1.7 trillion in 2023. Within this vibrant ecosystem, the United States stands out with a market size of $343.70 billion. On a per capita basis, Americans lead globally, spending an average of $1460 annually on clothing and footwear. This high level of expenditure is indicative of the country's strong consumer culture, particularly among Gen Z, where 36% report purchasing new clothing at least once every month.

Despite the significant advancements in computer vision across various sectors such as healthcare, sports, and automotive, its application in the fashion industry remains relatively under explored.

Our project, "Fashion Finder," aims to bridge this gap by utilizing machine learning models to classify images of people donned in various outfits. These models are designed to recognize and categorize different clothing items and attributes such as t-shirts, pants, dresses, glasses and other four six categories in various poses. By doing so, we aspire to provide a tool that not only enhances consumer experience but also offers valuable insights to retailers and designers by understanding current trends and customer preferences more profoundly.

This initiative is not just about classifying clothing but in future transforming how we perceive and interact with fashion using artificial intelligence.

2 Data

2.1 Dataset Used

For our project, we utilized the dataset from the iMaterialist (Fashion) 2019 at FGVC6 competition, hosted on Kaggle. This dataset was originally designed for a challenge focused on creating bounding boundaries of automatic product detection in fashion images. It comprised approximately 50,000 images of people wearing a variety of clothing types in a variety of poses. Labels were applied by both domain experts and crowd workers. They provided detailed segmentations that cover a standardized taxonomy of 46 apparel items and 92 fine-grained attributes, making it a rich source for deep learning models aimed at understanding complex visual information in the fashion domain.

2.2 Adaptation for Multi-label Classification

Initially the dataset was conceived for image segmentation tasks, the dataset presented a steep learning curve due to our limited experience with image segmentation techniques. To align the dataset with our project's goals and our team's skill set, we transformed it into a format suitable for multi-label classification. This adaptation involved simplifying the existing segmentation labels into classification labels that identify multiple clothing types present in each image. This modification allowed us to focus on developing a model that can recognize and categorize various clothing items from images, leveraging our strengths in handling classification tasks and making the project more feasible within our technical constraints.

2.3 Splits

The dataset was already split in train and test. We just used the train dataset in our training scripts with 80-20 train test split and we did not touch the original test set of kaggle in the training process as it would be used for final evaluation scores. Alternatively

we added a column called Split in the final dataset excel file using 80-20. This makes it easier to process and read in the samples one at a time in the custom data class created.

3 Modeling

3.1 CNN

First, We designed a custom convolutional neural network (CNN) for this task as CNN are most popular for image classification tasks. Our CNN architecture comprises three convolutional layers each followed by batch normalization and a ReLU activation. Each convolutional layer is followed by a max pooling layer to reduce the spatial dimensions of the feature maps. The final feature maps are passed through a global average pooling layer, reducing each feature map to a single value. These values are then fed into a fully connected layer that outputs the predictions for the 46 classes.

3.2 Resnet

Then We then tried pretrained models. First we tried was Resnet, The key innovation of ResNet is the introduction of residual connections, also known as skip connections. These connections allow the network to learn residual functions instead of learning unreferenced functions. The idea is that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. The residual connections enable the gradients to flow directly through the network, mitigating the vanishing gradient problem and allowing for much deeper networks to be trained effectively. We found that Resnet 101 was working best for our dataset based on F1 score. ResNet-101 as the name suggests consists of 101 layers.

3.3 Vision Transformer (ViT)

After experimenting with CNN and ResNet architectures, we explored the Vision Transformer (ViT) model for our multi-label image classification task. ViT is a relatively new architecture that adapts the transformer model, which has been highly successful in natural language processing, to the domain of computer vision. Initially, we attempted to use a pre-trained ViT model called vit-base-patch16-224 from the Hugging Face library. However, we encountered challenges with this model, as the loss remained high during training, and we were unable to resolve the issue. As an alternative, we turned to the ViT implementation provided by PyTorch in the torchvision library. The ViT model we used, specifically the vit_b_16 model from PyTorch, is pre-trained on the ImageNet dataset.

The architecture of ViT differs from traditional CNNs. Instead of using convolutional layers, ViT divides the input image into fixed-size patches of 16x16 pixels. These patches are linearly embedded and combined with positional embeddings to preserve spatial information. The resulting sequence of embedded patches is then fed into a transformer encoder, which learns to attend to different patches and capture their relationships.

The transformer encoder consists of multiple layers of self-attention and feed-forward networks. The self-attention mechanism allows the model to weigh the importance of different patches and learn their dependencies. After passing through the transformer layers, the encoded representation is fed into a linear classification head, which predicts the presence or absence of each class label.

To adapt the pre-trained ViT model to our multi-label classification task, we modified the classification head by replacing it with a linear layer that outputs the desired number of classes (46 in our case). We also applied a sigmoid activation function to the output of the classification head to obtain probability scores for each class.

The ViT model demonstrated impressive performance on our test set, achieving high F1 scores and accuracy. The self-attention mechanism of the transformer architecture allowed the model to effectively capture the relationships between different regions of the image and make accurate predictions for multiple labels simultaneously.

3.4 Inceptionnet

InceptionNet is a convolutional neural network (CNN) architecture that Google developed to improve upon the performance of previous CNNs on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The reason for choosing this pretrained model is because it uses inception modules which is a combination of m * m sized kernels specifically 1 * 1 , 3 * 3 and 5 * 5. The result is that these smaller and varying kernel sizes will learn a combination of local and global features from the input data (images). These feature maps created at different scales are then concatenated together to form a better representation of the input data and this is known as the inception module.

Applying batch normalization and other regularization techniques through layers of convolving and pooling, the pretrained model returns a softmax classification however,the task is at hand is to predict a multilabel classification data, the last layer of the pretrained model is transformed to a linear layer and trained. The InceptionNet model expects the input data(image) to come in as 299 * 299 so the image had to be resized to match this size. Using pytorch torchvision transforms version 2 method, we were able to apply both geometric and functional transformations to the input data(images). Training the model for less than 5 epochs resulted in the loss increasing. Training for longer than 8 epochs however stabilized the model. A figure can be found under the Performance Metrics section

3.5 Training

CNN, Resnet, and Vit were trained for 10 epochs using the same training loop and evaluation metrics. The model's performance is assessed using F1 scores (micro and macro) and accuracy. The best model based on the validation F1 macro score is saved for later use. For image preprocessing, we used the transformation function provided by the pretrained models so that they are in the same shape and size during thier training process but for custom CNN we added Random horizontal flips, rotations and jitters to make model generalize better. We employed techniques such as weighted random sampling to handle class imbalance and used a learning rate scheduler to adjust the

learning rate during training.

3.5.1 Class Weight Calculation

Given the imbalanced nature of our dataset, as discussed in the data section, we implemented a strategy to manage the uneven distribution of classes effectively. This strategy involves calculating class weights to ensure that the model does not become biased toward more frequently occurring classes.

To calculate class weights, we first converted the categorical labels into a binary matrix where each row represents an image and each column represents a class. This matrix was achieved using the Pandas get_dummies function on the Category column of our training data:

```
label_matrix = train_data["Category"].str.get_dummies(",")
```

Next, we computed the frequency of each class across all samples:

```
class_frequencies = label_matrix.sum()
total_samples = class_frequencies.sum()
```

Using these frequencies, we determined the weight for each class by dividing the total number of samples by the product of the frequency of the class and the number of classes, ensuring that under-represented classes are given higher importance during model training:

```
class_weights = total_samples / (class_frequencies * len(class_frequencies))
```

To apply these weights during the training phase, we calculated sample weights for each image in the dataset. As it is Multi label, we were doing the sum of the weights of each class present as the weight of the image. But it was taking a lot of time so we got the same operation done faster multiplying the binary label matrix with the class weight matrix:

```
sample_weights = label_matrix.dot(class_weight_tensor).values
```

We then used these sample weights to create a sampler for the inbuilt DataLoader module in pytorch, ensuring that each batch of data is representative of the overall dataset, despite the class imbalance:

```
sampler = WeightedRandomSampler(
sample_weights, num_samples=len(train_dataset), replacement=True
)
```

Finally, the DataLoader was configured to use this sampler along with a specified batch size and number of worker threads for loading data:

```
train_loader = DataLoader(train_dataset, batch_size=16, sampler=sampler)
```

After using class weights our models fairly improved across all classes, and its ability to generalize also got better which reduces the risk of bias towards more frequent classes.

3.5.2 Freeze-Unfreeze technique

The freeze-unfreeze technique, is a method for accelerating the training of deep neural networks by progressively freezing layers. The motivation behind this technique is that early layers in deep architectures tend to converge to simple configurations (e.g., edge detectors) and may not require as much fine-tuning as later layers, which contain most of the parameters.

In our script, we employed a variant of the freeze unfreeze technique. We start by freezing all layers of the pre-trained models except for the last fully connected layer. During training, we gradually unfreeze more layers as the number of epochs progresses. This allows the model to adapt its weights to our specific task while leveraging the pre-trained features from earlier layers.

The freeze-unfreeze technique helped speed up the training process but it did not improved the F1 scores or Accuracy that much.

3.5.3 FocalLoss Loss function

FocalLoss function is designed to address class imbalance by down-weighting the easy examples or labels such that the contribution of these easy examples to the overall loss value is small.The traditional Cross entropy function applies equal weights to each class i.e. for a given predicted probability p, the loss value calculated will be the same for any class. To solve this problem, if the predicted probability of a class is low, we penalize the loss heavily and if the predicted probability is high, we do not penalize the loss. We introduce two parameters, alpha and gamma. Gamma is the modulating factor, if we increase gamma,it changes the loss function curve and extends our criteria of well-classified examples, consequently extending the range of probabilities where the loss function is low. Gamma reduces the loss contribution from easy examples. Alpha on the other hand, is the weighting factor, (we see this in the class weight calculation). is the inverse class frequency. alpha at t is the alpha for positive class and 1 - alpha for negative class. This helped our model focus on harder examples during training.

$$FL(p) = \begin{cases} -\alpha(1-p)^{\gamma} log(p), & y = 1 \\ -(1-\alpha)p^{\gamma} log(1-p), & otherwise \end{cases}$$

Figure 1: FocalLoss Equation

4 Performance Metrics

To evaluate the effectiveness of our models, we analyzed their performance using a variety of metrics, including Accuracy, F1 Score, Precision, and Recall. These metrics provide insights into how well each model predicts the correct clothing categories.

Model	Accuracy	F1 Score	Precision	Recall
ViT_B_16	15.03%	74.84%	78.50%	71.51%
ResNet_101	7.43%	65.01%	69.05%	61.41%
CNN	4.74%	55.63%	71.75%	45.43%
InceptionNet_V3	3.19%	30%	52%	43.25%

Table 1: Overall performance metrics of the models.

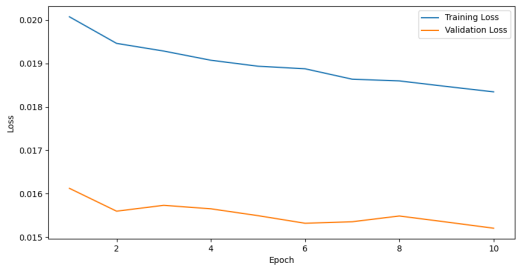

Figure 2: Training and Validation Loss Trends for CNN Model Across Epochs

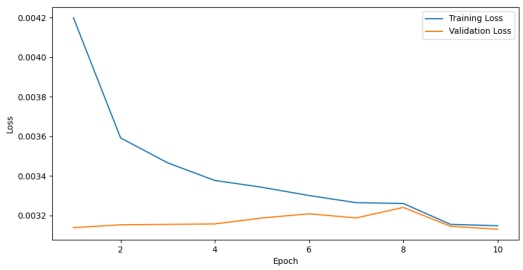

Figure 3: Training and Validation Loss Trends for ViT Model Across Epochs

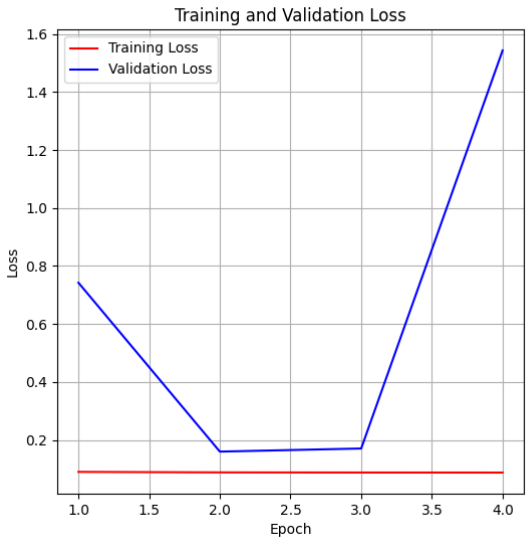

Figure 4: Training and Validation Loss Trends for InceptionNet for Epochs less than 5

5 App

To see our trained models in live action, we developed an interactive tool using Streamlit, an open-source app framework that is particularly well-suited for machine learning and data science projects. This application allows users to engage with our models in a real-world setting, providing a hands-on experience with the models we built.

Users have the option to select from multiple classification models that we have trained. This feature provides flexibility and allows users to compare performance across different models, catering to various needs and scenarios. Before making a selection, users can view comprehensive performance metrics for each model. We display both overall metrics and per-class metrics.

The application supports the upload of images of any size and shape. Once an image is uploaded, it undergoes automatic preprocessing to format it correctly for the model selected. This preprocessing includes resizing and normalizing the image to fit the input requirements of the model. After processing the image, the selected model classifies the content and outputs the probabilities for each class detected in the image. Results are displayed in a table, highlighting the probability percentage for each class that exceeds a user-defined threshold. This visualization not only shows which items are present in the

image but also indicates the model's confidence level for each prediction.

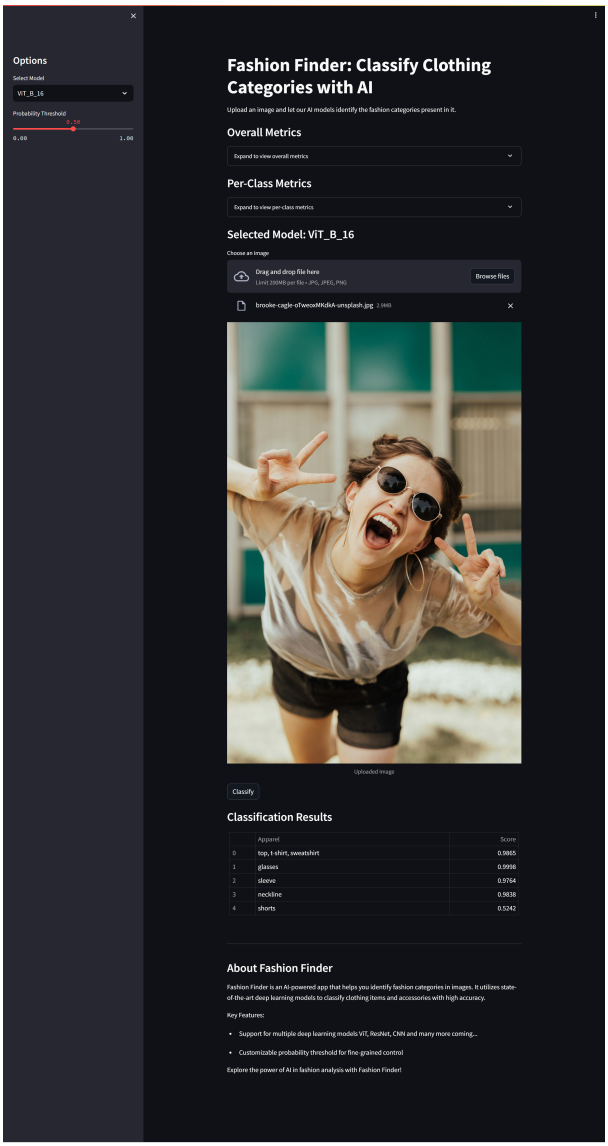

Figure 5: Streamlit APP

6 Conclusion

Our project successfully tackled the complex task of multi-label classification of fashion apparel using state-of-the-art machine learning models such as CNNs, ResNets, InceptionNet and transformers. Among these, the Vision Transformer (ViT) model stood out, demonstrating superior accuracy and generalization capabilities across various clothing categories.

Furthermore, the integration of these models into a Streamlit-based application provided a seamless and user-friendly platform for real-time fashion classification. This application allows users to effortlessly upload images and receive immediate, reliable classification results, making it an excellent tool for both fashion enthusiasts and industry professionals.

7 Future Enhancements

Our initial goal was to develop a model that could provide personalized fashion recommendations based on both text and image inputs. However, due to technical and time constraints, we focused on multi-label classification of fashion images using CNN, ResNet, and transformer-based models. In the future, we aim to:

1. Implement unsupervised learning techniques, such as K-means clustering, to prescreen uploaded images and identify non-apparel images. This will improve the efficiency and accuracy of our application by preventing the processing of irrelevant images.

2. Gain a deeper understanding of each machine learning models used in our project. This knowledge will enable us to conduct more effective post-training analysis and optimize our models for better accuracy in future iterations.

3. Revisit our initial goal of providing personalized fashion recommendations by incorporating text/image inputs and developing a more comprehensive system that closely aligns with real world use cases.

8 References

1. Pytorch documentation.
2. google/vit-base-patch16-224
3. Fine-Tuning Vision Transformer with Hugging Face and PyTorch
4. Training data-efficient image transformers & distillation through attention
5. FREEZEOUT: ACCELERATE TRAINING BY PROGRESSIVELY FREEZING LAYERS.
6. Pytorch Multi Label Classifier github example
7. Enhancing Multi-Class Classification with Focal Loss in PyTorch
8. Focal Loss for Dense Object Detection

A Appendix: Source Code

A.1 Dataset Loading and Preprocessing Code

```
# %%
import pandas as pd
from torch import nn
from torchvision import models, transforms
import torch
from torch.utils.data import Dataset
import cv2
import os
from sklearn.model_selection import train_test_split
from PIL import Image


# %%
PATH = "/home/ubuntu/Final-Project-Group4"
EXCEL_PATH = PATH + os.path.sep + "dataset" + os.path.sep + "final_dataset.xlsx"
DATA_DIR = PATH + os.path.sep + "dataset" + os.path.sep + "train" + os.path.sep

# %%
df = pd.read_excel(EXCEL_PATH)
one_hot_encoded = df["Category"].str.get_dummies(sep=",")
df["target_class"] = one_hot_encoded.apply(lambda x: ",".join(x.astype(str)), axis=1)
train_data, test_data = train_test_split(df, test_size=0.20, random_state=42)
# %%


class MultiLabelImageDataset(Dataset):
    def __init__(self, list_IDs, type_data, transform=None):
        self.type_data = type_data
        self.list_IDs = list_IDs
```

```
        self.transform = transform

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

    def __getitem__(self, index):
        ID = self.list_IDs[index]

        if self.type_data == "train":
            y = train_data.target_class.get(ID)
            file = DATA_DIR + train_data.ImageId.get(ID)
        else:
            y = test_data.target_class.get(ID)
            file = DATA_DIR + test_data.ImageId.get(ID)

        y = y.split(",")
        labels_ohe = [int(e) for e in y]
        y = torch.FloatTensor(labels_ohe)

        img = cv2.imread(file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = Image.fromarray(img)  # Convert numpy array to PIL Image

        if self.transform:
            img = self.transform(img)

        return img, y

train_transform = transforms.Compose(
    [
        transforms.Resize((224, 224)),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation
    =0.1, hue=0.1),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
     0.225]),
    ]
```

```
)

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

train_dataset = MultiLabelImageDataset(
    list_IDs=train_data.index,
    type_data="train",
    transform=train_transform,
)
test_dataset = MultiLabelImageDataset(
    list_IDs=test_data.index,
    type_data="test",
    transform=test_transform,
)
```

Listing 1: Dataset Loader

A.2 Class weights calculation Code

```
label_matrix = train_data["Category"].str.get_dummies(",")
class_frequencies = label_matrix.sum()
total_samples = class_frequencies.sum()
class_weights = total_samples / (class_frequencies * len(class_frequencies))
class_weight_tensor = torch.tensor(class_weights.values, dtype=torch.float)
sample_weights = label_matrix.dot(class_weight_tensor).values

# %%
sampler = WeightedRandomSampler(
    sample_weights, num_samples=len(train_dataset), replacement=True
)

train_loader = DataLoader(train_dataset, batch_size=16, sampler=sampler, num_workers=4)
validation_loader = DataLoader(test_dataset, batch_size=16, num_workers=4)
```

Listing 2: Class weights for imabalnced dataset

A.3 CNN Model Code

```
from torch import nn

class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(128, num_classes)
```

```
15          self.act = nn.ReLU(inplace=True)
16
17      def forward(self, x):
18          x = self.pool(self.act(self.bn1(self.conv1(x))))
19          x = self.pool(self.act(self.bn2(self.conv2(x))))
20          x = self.pool(self.act(self.bn3(self.conv3(x))))
21          x = self.global_avg_pool(x)
22          x = x.view(x.size(0), -1)
23          x = self.fc(x)
24          return x
```

Listing 3: CNN model for image classification

A.4 Resnet Model Code

```
from torch import nn
from torchvision import models

model = models.resnet101(weights=models.ResNet101_Weights.DEFAULT)
num_classes = 46


model.fc = nn.Linear(model.fc.in_features, num_classes)
model.fc.sigmoid = nn.Sigmoid()
model.fc.requires_grad = True
```

Listing 4: Resnet model for image classification

A.5 ViT Model Code

```
from torch import nn
from torchvision import models

model = models.vit_b_16(weights=models.ViT_B_16_Weights.
    IMAGENET1K_SWAG_E2E_V1)
num_classes = 46


model.heads.head = nn.Linear(model.heads.head.in_features, num_classes)
model.heads.head.sigmoid = nn.Sigmoid()
model.heads.head.requires_grad = True
```

Listing 5: ViT model for image classification

A.6 Pytorch model training loop

```

criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.heads.head.parameters(), lr=0.001)

```

```
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode="max", factor=0.1, patience=2, verbose=True
)

num_epochs = 10
model.to(device)
best_val_f1_macro = 0

train_losses = []
val_losses = []
val_f1_micros = []
val_f1_macros = []


for epoch in range(num_epochs):
    model.train()
    train_loss = 0.0
    steps_train = 0.0
    with tqdm(total=len(train_loader), desc="Epoch {}".format(epoch)) as pbar:
        for images, labels in train_loader:
            images = images.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            steps_train += 1
            train_loss += loss.item()  # * images.size(0)
            pbar.update(1)
            pbar.set_postfix_str("Train Loss: {:.5f}".format(train_loss / steps_train))
    train_loss /= len(train_dataset)
    train_losses.append(train_loss)

    model.eval()
    val_loss = 0.0
```

```
    steps_test = 0.0
    predictions = []
    true_labels = []

    with torch.no_grad():
        with tqdm(total=len(validation_loader), desc="Epoch {}".format(
epoch)) as pbar:
            for images, labels in validation_loader:
                images = images.to(device)
                labels = labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()  # * images.size(0)
                predictions.extend(outputs.sigmoid().cpu().numpy() >= 0.5)
                true_labels.extend(labels.cpu().numpy())
                steps_test += 1
                pbar.update(1)
                pbar.set_postfix_str("Test Loss: {:.5f}".format(val_loss /
steps_test))

    val_loss /= len(test_dataset)
    val_losses.append(val_loss)

    f1_micro = f1_score(true_labels, predictions, average="micro")
    f1_macro = f1_score(true_labels, predictions, average="macro")
    f1_scores = f1_score(true_labels, predictions, average=None)
    val_accuracy_score = accuracy_score(true_labels, predictions)
    val_f1_micros.append(f1_micro)
    val_f1_macros.append(f1_macro)

    print(
        f"Epoch [{epoch+1}/{num_epochs}], "
        f"Train Loss: {train_loss:.4f}, "
        f"Val Loss: {val_loss:.4f}, "
        f"Val F1 Micro: {f1_micro:.4f}, "
        f"Val F1 Macro: {f1_macro:.4f}, "
        f"Val Accuracy: {val_accuracy_score:.4f}"
    )
    print("Class-wise F1 Scores:")
```

```
    for i, score in enumerate(f1_scores):
        print(f"Class {i+1}: {score:.4f}")

    if f1_macro > best_val_f1_macro:
        best_val_f1_macro = f1_macro
        best_epoch = epoch + 1
        torch.save(
            model.state_dict(),
            f"model_vit_b_16.pt",
        )

    print(classification_report(true_labels, predictions))
    scheduler.step(f1_macro)
```

Listing 6: Pytorch model training loop