EuroSAT Classification with Swin Transformer and Advanced Techniques (Version 1)

Documented Code

February 14, 2025

Section 1 of this document provides an annotated listing of the code used for optimized Swin Transformer for EuroSAT Classification with advanced techniques such as Mixup, CosineAnnealingWarmRestarts, GradScaler for mixed precision, and a custom parameter scheduling function. The code is designed to run within a Python environment (e.g., Kaggle notebooks, Google Colab) that supports PyTorch, timm, and scikit-learn.

Section 2 provides the code to train each of the 5 ViT architecture models, namely DeIT, YOLO, Swin, PVT and MAE. This was used to select the baseline model used in Section 1 above.

1 Optimized Swin Transformer

Below is a comprehensive code listing for training a *Swin Transformer* on the EuroSAT dataset using advanced techniques:

- **Data Augmentation** via `RandAugment`, random flips, random erasing, etc.
- **Mixup** to enhance robustness.
- **Layer-wise LR** scheduling for finer control over learning rates in deeper blocks.
- **CosineAnnealingWarmRestarts** for cyclical LR patterns.
- **SWA (Stochastic Weight Averaging)** to stabilize and improve final performance.
- **Fine-Tuning** stage that freezes earlier layers and adjusts LR for the final classification head.

Listing 1: EuroSAT + Swin Base Patch4 Window7 224 with RandAugment, Mixup, SWA, and Fine-Tuning

```python
import os
import glob
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
```

```python
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix

import seaborn as sns
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image

# Additional transforms
from torchvision.transforms import RandAugment, RandomErasing

# timm (SOTA models) + Mixup
from timm import create_model
from timm.data.mixup import Mixup

# PyTorch optimization
from torch.optim import AdamW
from torch.optim.lr_scheduler import CosineAnnealingWarmRestarts

# AMP (mixed precision) and GradScaler
from torch.cuda.amp import autocast, GradScaler
# SWA from PyTorch
from torch.optim.swa_utils import AveragedModel, SWALR

# Seeds for reproducibility
torch.manual_seed(42)
np.random.seed(42)

####################################################################
# 1. DATASET PREPARATION
####################################################################
data_dir = '/kaggle/input/eurosat10-classes/EuroSAT_RGB/'
# 'data_dir' has subfolders named after land-cover classes (e.g., "AnnualCrop", "Forest",
    etc.)

# Gather all .jpg paths
image_paths = glob.glob(os.path.join(data_dir, '*', '*.jpg'))
labels = [os.path.basename(os.path.dirname(path)) for path in image_paths]

# Make a DataFrame
df = pd.DataFrame({'image_path': image_paths, 'label': labels})

# Encode label strings into integers
le = LabelEncoder()
df['label_enc'] = le.fit_transform(df['label'])

# Stratified split: 80% train, 20% validation
train_df, val_df = train_test_split(
    df,
```

```
    test_size=0.2,
    stratify=df['label_enc'],
    random_state=42
)

print(f"Total images: {len(df)}")
print(f"Training images: {len(train_df)}")
print(f"Validation images: {len(val_df)}")

####################################################################
# 2. IMAGE TRANSFORMS
####################################################################
# Mean/Std for EuroSAT (approx)
mean = [0.3444, 0.3809, 0.4082]
std = [0.1829, 0.1603, 0.1321]

train_transforms = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    RandAugment(num_ops=2, magnitude=7), # apply RandAugment
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std),
    RandomErasing(p=0.5, scale=(0.02, 0.2)) # random erasing
])

val_transforms = transforms.Compose([
    transforms.Resize(224),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std),
])

####################################################################
# 3. CUSTOM DATASET
####################################################################
class EuroSATDataset(Dataset):
    """
    A simple Dataset class that:
    - Uses a DataFrame with 'image_path' and 'label_enc'
    - Loads images from disk, applies transforms
    - Returns (image_tensor, label_index)
    """
    def __init__(self, df, transforms):
        self.df = df.reset_index(drop=True)
        self.transforms = transforms

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

    def __getitem__(self, idx):
        image_path = self.df.loc[idx, 'image_path']
        label_enc = self.df.loc[idx, 'label_enc']
```

```python
        image = Image.open(image_path).convert('RGB')
        image = self.transforms(image)
        return image, label_enc

# Build Dataset + Dataloader
train_dataset = EuroSATDataset(train_df, train_transforms)
val_dataset = EuroSATDataset(val_df, val_transforms)

batch_size = 64
train_loader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=4,
    pin_memory=True
)
val_loader = DataLoader(
    val_dataset,
    batch_size=batch_size,
    shuffle=False,
    num_workers=4,
    pin_memory=True
)

####################################################################
# 4. CREATE SWIN MODEL
####################################################################
model = create_model(
    'swin_base_patch4_window7_224',
    pretrained=True,
    num_classes=10,
    drop_rate=0.0,
    drop_path_rate=0.1
)
# Cross-entropy with label smoothing
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)

####################################################################
# 5. PARAM GROUPS + COSINE RESTARTS
####################################################################
def get_param_groups(model, base_lr, weight_decay):
    """
    Build parameter groups for layer-wise LR scheduling.
    We'll downscale LR for deeper blocks by 0.95^(layer_index).
    """
    no_weight_decay = model.no_weight_decay()
    param_groups = {}

    for name, param in model.named_parameters():
        if not param.requires_grad:
            continue
        group_name = 'layer_0'
        if 'blocks' in name:
            block_num = int(name.split('.')[1])
```

```
            group_name = f'layer_{block_num + 1}'
        elif 'cls_token' in name or 'pos_embed' in name:
            group_name = 'layer_0'
        else:
            group_name = 'layer_0'

        if group_name not in param_groups:
            param_groups[group_name] = {
                'params': [],
                'weight_decay': weight_decay,
                'lr': base_lr
            }
        param_groups[group_name]['params'].append(param)

    param_groups_list = []
    num_layers = len(param_groups)
    for i, (group_name, group) in enumerate(
        sorted(param_groups.items(), key=lambda x: x[0])
    ):
        group['lr'] = base_lr * (0.95 ** (num_layers - i - 1))
        param_groups_list.append(group)

    return param_groups_list

base_lr = 3e-5
weight_decay = 0.01
optimizer = AdamW(get_param_groups(model, base_lr, weight_decay))

scheduler = CosineAnnealingWarmRestarts(optimizer, T_0=10, T_mult=1)

##################################################################
# 6. MIXUP
##################################################################
mixup_fn = Mixup(
    mixup_alpha=0.8,
    cutmix_alpha=1.0,
    prob=1.0,
    switch_prob=0.5,
    mode='batch',
    label_smoothing=0.1,
    num_classes=10
)

##################################################################
# 7. AMP / GRAD SCALER
##################################################################
scaler = GradScaler()

# Move model to GPU
model = model.cuda()

##################################################################
# 8. SWA SETUP
##################################################################
```

```python
swa_model = AveragedModel(model)
swa_start_epoch = 25
swa_scheduler = SWALR(
    optimizer,
    swa_lr=base_lr,
    anneal_epochs=5,
    anneal_strategy="cos"
)

#####################################################################
# 9. TRAINING (MIXUP, AMP, SWA)
#####################################################################
def train_one_epoch(epoch, model, dataloader, optimizer, criterion, scheduler, mixup_fn):
    model.train()
    running_loss = 0.0
    total_samples = 0

    for images, labels in dataloader:
        images = images.cuda(non_blocking=True)
        labels = labels.cuda(non_blocking=True)

        # Apply mixup
        images, labels = mixup_fn(images, labels)

        optimizer.zero_grad()
        with autocast(device_type='cuda', dtype=torch.float16):
            outputs = model(images)
            loss = criterion(outputs, labels)

        scaler.scale(loss).backward()
        nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)
        scaler.step(optimizer)
        scaler.update()

        running_loss += loss.item() * images.size(0)
        total_samples += images.size(0)

    epoch_loss = running_loss / total_samples

    if epoch < swa_start_epoch:
        scheduler.step()

    print(f"[Epoch {epoch} | Train] Loss: {epoch_loss:.4f}")
    return epoch_loss

def validate(model, dataloader, criterion):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    all_preds = []
    all_labels = []

    with torch.no_grad():
```

```python
        for images, labels in dataloader:
            images = images.cuda(non_blocking=True)
            labels = labels.cuda(non_blocking=True)

            with autocast(device_type='cuda', dtype=torch.float16):
                outputs = model(images)
                loss = criterion(outputs, labels)

            running_loss += loss.item() * images.size(0)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    epoch_loss = running_loss / total
    epoch_acc = correct / total

    print("Classification Report:")
    print(classification_report(all_labels, all_preds, target_names=le.classes_, digits=4))

    cm = confusion_matrix(all_labels, all_preds)
    print(f"[Validation] Loss: {epoch_loss:.4f}, Accuracy: {epoch_acc:.4f}")
    return epoch_loss, epoch_acc

##################################################################
# 10. MAIN TRAIN LOOP
##################################################################
num_epochs = 30
best_acc = 0.0

for epoch in range(1, num_epochs + 1):
    print(f"Epoch {epoch}/{num_epochs}")

    train_loss = train_one_epoch(
        epoch, model, train_loader, optimizer, criterion, scheduler, mixup_fn
    )

    # SWA updates after 'swa_start_epoch'
    if epoch >= swa_start_epoch:
        swa_model.update_parameters(model)
        swa_scheduler.step()

    val_loss, val_acc = validate(model, val_loader, criterion)

    if val_acc > best_acc:
        best_acc = val_acc
        torch.save(model.state_dict(), 'best_vit_model.pth')
        print("Saved Best Model!")

    print('-' * 40)

```

```python
# Update BN for SWA model
torch.optim.swa_utils.update_bn(train_loader, swa_model, device='cuda')

# Save SWA model
torch.save(swa_model.state_dict(), 'swa_vit_model.pth')
print("SWA Model Saved!")

###################################################################
# 11. FINE-TUNING PHASE
###################################################################
import torch
import torch.nn as nn
from torch.cuda.amp import autocast, GradScaler
from torch.optim import AdamW
from torch.optim.lr_scheduler import CosineAnnealingWarmRestarts
from timm import create_model
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Re-create the same model structure, but we'll load our best or SWA weights
model_finetune = create_model(
    'swin_base_patch4_window7_224',
    pretrained=False,
    num_classes=10
)
model_finetune.load_state_dict(torch.load('best_vit_model.pth'))
model_finetune = model_finetune.cuda()

# Optionally freeze layers except final head
for name, param in model_finetune.named_parameters():
    if 'head' not in name:
        param.requires_grad = False

finetune_lr = 1e-5
finetune_weight_decay = 1e-4
finetune_optimizer = AdamW(
    filter(lambda p: p.requires_grad, model_finetune.parameters()),
    lr=finetune_lr,
    weight_decay=finetune_weight_decay
)
finetune_scheduler = CosineAnnealingWarmRestarts(finetune_optimizer, T_0=5, T_mult=1)

finetune_criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
finetune_scaler = GradScaler()

def finetune_one_epoch(epoch, model, dataloader, optimizer, criterion, scheduler):
    model.train()
    running_loss = 0.0
    total_samples = 0

    for images, labels in dataloader:
        images = images.cuda(non_blocking=True)
        labels = labels.cuda(non_blocking=True)
```

```python

        optimizer.zero_grad()
        with autocast(device_type='cuda', dtype=torch.float16):
            outputs = model(images)
            loss = criterion(outputs, labels)

        finetune_scaler.scale(loss).backward()
        nn.utils.clip_grad_norm_(model.parameters(), 5.0)
        finetune_scaler.step(optimizer)
        finetune_scaler.update()

        running_loss += loss.item() * images.size(0)
        total_samples += images.size(0)

    epoch_loss = running_loss / total_samples
    scheduler.step()
    print(f"[Fine-Tune Epoch {epoch}] Loss: {epoch_loss:.4f}")
    return epoch_loss

def validate_finetune(model, dataloader, criterion):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for images, labels in dataloader:
            images = images.cuda(non_blocking=True)
            labels = labels.cuda(non_blocking=True)

            with autocast(device_type='cuda', dtype=torch.float16):
                outputs = model(images)
                loss = criterion(outputs, labels)

            running_loss += loss.item() * images.size(0)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    epoch_loss = running_loss / total
    epoch_acc = correct / total
    print(f"[Fine-Tune Validation] Loss: {epoch_loss:.4f}, Accuracy: {epoch_acc:.4f}")

    final_report = classification_report(all_labels, all_preds, target_names=le.classes_, digits=4)
    print("\nFinal Classification Report (Fine-Tuned Model):")
    print(final_report)

    cm = confusion_matrix(all_labels, all_preds)
```

```
    plt.figure(figsize=(8,6))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
                xticklabels=le.classes_, yticklabels=le.classes_)
    plt.title("Confusion Matrix (Fine-Tuned Model)")
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.show()

    return epoch_loss, epoch_acc

finetune_epochs = 10
best_acc_ft = 0.0

for ep in range(1, finetune_epochs+1):
    train_loss_ft = finetune_one_epoch(
        ep, model_finetune, train_loader, finetune_optimizer, finetune_criterion,
            finetune_scheduler
    )
    val_loss_ft, val_acc_ft = validate_finetune(model_finetune, val_loader,
        finetune_criterion)

    if val_acc_ft > best_acc_ft:
        best_acc_ft = val_acc_ft
        torch.save(model_finetune.state_dict(), "best_vit_model_finetuned.pth")
        print("Saved Best Fine-Tuned Model!\n")
```

2 Training 5 different ViT models

Listing 2: Downloading and Preparing EuroSAT Data

```
!wget http://madm.dfki.de/files/sentinel/EuroSAT.zip
!unzip EuroSAT.zip -d EuroSAT/

# Explanation:
# 1) We retrieve the EuroSAT dataset from dfki.de.
# 2) We unzip the dataset contents into a directory named 'EuroSAT/'.
# This yields subfolders (e.g., "AnnualCrop", "Forest", etc.) each containing images.
```

Listing 3: Custom Collate Function for DataLoader

```
from torch.utils.data import DataLoader
import torch

def collate_fn(batch):
    # 'batch' is a list of tuples: (PIL_image, label_index).
    images, labels = zip(*batch)
    # Convert 'labels' (ints) to a single Tensor.
    return list(images), torch.tensor(labels)
```

Listing 4: Dataset Splitting and Dataloader Setup

```
import torch
```

```python
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, SubsetRandomSampler
from torchvision import datasets
import numpy as np
import os
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
    classification_report

from transformers import (
    AutoImageProcessor,
    AutoModelForImageClassification,
    AutoModelForPreTraining,
    AutoModel,
    VitMatteForImageMatting
)

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

data_dir = "/content/EuroSAT/2750"

# Create ImageFolder dataset from the unzipped directory
dataset = datasets.ImageFolder(root=data_dir)
classes = dataset.classes
num_classes = len(classes)
print("Classes:", classes)

# We'll define a validation split ratio of 0.2 (i.e., 20% for validation)
valid_size = 0.2
batch_size = 32

num_data = len(dataset)
indices = list(range(num_data))
np.random.shuffle(indices) # shuffle indices for random splitting
split = int(np.floor(valid_size * num_data))

# Partition dataset indices
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# We define train_loader and valid_loader with collate_fn
train_loader = DataLoader(
    dataset,
    batch_size=batch_size,
    sampler=train_sampler,
    collate_fn=collate_fn
)

valid_loader = DataLoader(
    dataset,
    batch_size=batch_size,
```

```python
    sampler=valid_sampler,
    collate_fn=collate_fn
)
```

Listing 5: Training, Validation, and Evaluation Routines

```python
def train_one_epoch(model, processor, dataloader, optimizer):
    """
    Train the model for one epoch.

    Args:
      model: HF model for image classification
      processor: HF image processor to transform PIL images to tensors
      dataloader: DataLoader yielding (list_of_PILs, label_tensor)
      optimizer: e.g. AdamW
    """
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in dataloader:
        labels = labels.to(device)
        inputs = processor(images, return_tensors="pt").to(device)

        outputs = model(**inputs, labels=labels)
        loss = outputs["loss"]
        logits = outputs["logits"]

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

        running_loss += loss.item() * labels.size(0)

        _, predicted = logits.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    epoch_loss = running_loss / total
    epoch_acc = 100.0 * correct / total
    return epoch_loss, epoch_acc

def validate(model, processor, dataloader):
    """
    Validate model performance on a held-out set.
    """
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in dataloader:
```

```python
            labels = labels.to(device)
            inputs = processor(images, return_tensors="pt").to(device)

            outputs = model(**inputs, labels=labels)
            loss = outputs["loss"]
            logits = outputs["logits"]

            val_loss += loss.item() * labels.size(0)

            _, predicted = logits.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

    val_loss /= total
    val_acc = 100.0 * correct / total
    return val_loss, val_acc

def evaluate_model(model, processor, dataloader):
    """
    Obtain all predictions and ground truths to compute classification metrics.
    """
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for images, labels in dataloader:
            labels = labels.to(device)
            inputs = processor(images, return_tensors="pt").to(device)
            outputs = model(**inputs)
            logits = outputs["logits"]

            _, predicted = logits.max(1)
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    return all_labels, all_preds

def plot_confusion_matrix(all_labels, all_preds, classes):
    """
    Plot a confusion matrix using sklearn's ConfusionMatrixDisplay.
    """
    cm = confusion_matrix(all_labels, all_preds)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes)
    disp.plot(cmap=plt.cm.Blues, xticks_rotation='vertical', values_format='d')
    plt.title("Confusion Matrix on Validation Set")
    plt.show()

def print_classification_report(all_labels, all_preds, classes):
    """
    Print a detailed classification report with precision, recall, F1.
    """
    report = classification_report(all_labels, all_preds, target_names=classes)
    print("Classification Report:\n", report)
```

Listing 6: High-Level Training and Evaluation Loop

```python
def train_and_evaluate(model, processor, train_loader, valid_loader, epochs=35, lr=1e-3,
    wd=1e-4):
    """
    Train the model for several epochs, track best val accuracy, and then
    produce final plots and classification metrics.

    Args:
      model: HF model for classification
      processor: HF image processor
      train_loader, valid_loader: DataLoaders
      epochs: number of epochs
      lr: learning rate
      wd: weight decay
    """
    optimizer = optim.AdamW(model.parameters(), lr=lr, weight_decay=wd)
    scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10)

    best_val_acc = 0.0
    train_losses, train_accuracies = [], []
    val_losses, val_accuracies = [], []

    best_model_state = None

    for epoch in range(epochs):
        train_loss, train_acc = train_one_epoch(model, processor, train_loader, optimizer)
        val_loss, val_acc = validate(model, processor, valid_loader)

        scheduler.step()

        train_losses.append(train_loss)
        train_accuracies.append(train_acc)
        val_losses.append(val_loss)
        val_accuracies.append(val_acc)

        print(f"Epoch [{epoch+1}/{epochs}]")
        print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%")
        print(f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.2f}%\n")

        # Track the best model
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            best_model_state = model.state_dict()

    if best_model_state is not None:
        model.load_state_dict(best_model_state)

    # Plot training vs. validation loss
    plt.figure(figsize=(10,5))
    plt.title("Training and Validation Loss")
    plt.plot(train_losses, label="Train Loss")
    plt.plot(val_losses, label="Validation Loss")
```

```
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()

    # Plot training vs. validation accuracy
    plt.figure(figsize=(10,5))
    plt.title("Training and Validation Accuracy")
    plt.plot(train_accuracies, label="Train Accuracy")
    plt.plot(val_accuracies, label="Validation Accuracy")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy (%)")
    plt.legend()
    plt.show()

    # Final evaluation on validation set
    all_labels, all_preds = evaluate_model(model, processor, valid_loader)
    plot_confusion_matrix(all_labels, all_preds, classes)
    print_classification_report(all_labels, all_preds, classes)

    return best_val_acc
```

Listing 7: Loading and Training the DeiT Model

```
def get_deit_model(num_classes):
    """
    Load the 'facebook/deit-small-patch16-224' model & processor for classification.
    """
    from transformers import AutoImageProcessor, AutoModelForImageClassification
    processor = AutoImageProcessor.from_pretrained(
        "facebook/deit-small-patch16-224",
        use_fast=True
    )
    model = AutoModelForImageClassification.from_pretrained(
        "facebook/deit-small-patch16-224",
        num_labels=num_classes,
        ignore_mismatched_sizes=True
    )
    model.to(device)
    return model, processor

print("=== DeiT ===")
deit_model, deit_processor = get_deit_model(num_classes)
deit_acc = train_and_evaluate(deit_model, deit_processor, train_loader, valid_loader)
print(f"DeiT Val Acc: {deit_acc:.2f}%")
```

Listing 8: Loading and Training the Swin Transformer Model

```
def get_swin_model(num_classes):
    """
    Load the 'microsoft/swin-tiny-patch4-window7-224' Swin Transformer for classification
        .
    """
    from transformers import AutoImageProcessor, AutoModelForImageClassification
```

```python
    processor = AutoImageProcessor.from_pretrained(
        "microsoft/swin-tiny-patch4-window7-224",
        use_fast=True
    )
    model = AutoModelForImageClassification.from_pretrained(
        "microsoft/swin-tiny-patch4-window7-224",
        num_labels=num_classes,
        ignore_mismatched_sizes=True
    )
    model.to(device)
    return model, processor

print("=== Swin Transformer ===")
swin_model, swin_processor = get_swin_model(num_classes)
swin_acc = train_and_evaluate(swin_model, swin_processor, train_loader, valid_loader)
print(f"Swin Transformer Val Acc: {swin_acc:.2f}%")
```

Listing 9: Loading and Training the MAE-based Model

```python
def get_mae_model(num_classes):
    """
    Load a pretrained MAE (Masked Autoencoder) model from 'facebook/vit-mae-large'
    and adapt it for classification by adding a custom head.
    """
    from transformers import AutoImageProcessor, AutoModelForPreTraining
    processor = AutoImageProcessor.from_pretrained("facebook/vit-mae-large", use_fast=
        True)
    pretrained_model = AutoModelForPreTraining.from_pretrained("facebook/vit-mae-large")
    base_model = pretrained_model.vit

    classification_head = nn.Linear(base_model.config.hidden_size, num_classes)

    class MAEForClassification(nn.Module):
        def __init__(self, base_model, classification_head):
            super().__init__()
            self.base_model = base_model
            self.classifier = classification_head

        def forward(self, pixel_values, labels=None):
            outputs = self.base_model(pixel_values=pixel_values)
            pooled_output = outputs.last_hidden_state[:, 0, :]
            logits = self.classifier(pooled_output)
            loss = None
            if labels is not None:
                loss_fn = nn.CrossEntropyLoss()
                loss = loss_fn(logits, labels)
            return {"loss": loss, "logits": logits}

    model = MAEForClassification(base_model, classification_head).to(device)
    return model, processor

print("=== MAE ===")
mae_model, mae_processor = get_mae_model(num_classes)
mae_acc = train_and_evaluate(mae_model, mae_processor, train_loader, valid_loader)
```

```
print(f"MAE Val Acc: {mae_acc:.2f}%")
```

Listing 10: Loading and Training the PVT (Pyramid Vision Transformer) Model

```
def get_pvt_model(num_classes):
    """
    Load 'Zetatech/pvt-tiny-224' for classification, specifying 'num_labels=num_classes'.
    """
    import torch
    import torch.nn as nn
    from transformers import AutoImageProcessor, PvtForImageClassification

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    processor = AutoImageProcessor.from_pretrained("Zetatech/pvt-tiny-224")
    model = PvtForImageClassification.from_pretrained(
        "Zetatech/pvt-tiny-224",
        num_labels=num_classes,
        problem_type="single_label_classification",
        ignore_mismatched_sizes=True
    )
    model.to(device)
    return model, processor

print("=== PVT ===")
pvt_model, pvt_processor = get_pvt_model(num_classes)
pvt_acc = train_and_evaluate(pvt_model, pvt_processor, train_loader, valid_loader)
print(f\"PVT Validation Accuracy: {pvt_acc:.2f}%\")
```

Listing 11: Loading and Training the YOLOS Model

```
def get_yolos_small_for_classification(num_classes):
    """
    Convert the 'hustvl/yolos-small' model (normally for object detection)
    into a classification model by adding a linear head.
    """
    import torch
    import torch.nn as nn
    from transformers import YolosModel, AutoImageProcessor

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    processor = AutoImageProcessor.from_pretrained("hustvl/yolos-small")
    base_model = YolosModel.from_pretrained("hustvl/yolos-small")

    hidden_size = base_model.config.hidden_size
    classification_head = nn.Linear(hidden_size, num_classes)

    class YOLOSSmallForClassification(nn.Module):
        def __init__(self, base_model, classifier):
            super().__init__()
            self.base_model = base_model
            self.classifier = classifier

        def forward(self, pixel_values, labels=None):
            outputs = self.base_model(pixel_values=pixel_values, return_dict=True)
```

```
            pooled_output = outputs.pooler_output # YOLOS model's global feature
            logits = self.classifier(pooled_output)
            loss = None
            if labels is not None:
                loss_fn = nn.CrossEntropyLoss()
                loss = loss_fn(logits, labels)
            return {"loss": loss, "logits": logits}

    model = YOLOSSmallForClassification(base_model, classification_head).to(device)
    return model, processor

print("=== YOLOS ===")
yolos_model, yolos_processor = get_yolos_small_for_classification(num_classes=num_classes)
yolos_acc = train_and_evaluate(yolos_model, yolos_processor, train_loader, valid_loader)
print(f\"YOLOS Validation Accuracy: {yolos_acc:.2f}%\")
```