```
# Required Libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import Dataset, DataLoader, random_split
import pandas as pd
from PIL import Image
import os
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score


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


image_transforms = {
    'train': transforms.Compose([
        transforms.Resize((299, 299)),  # Resizing images to 299x299 for InceptionV3 compatibility
        transforms.RandomHorizontalFlip(),  # Augmenting by randomly flipping images horizontally
        transforms.ToTensor(),  # Converting images to tensor format
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])  # Normalizing images with pre-defined means and std-dev
    ]),
    'val': transforms.Compose([
        transforms.Resize((299, 299)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}



# Custom Dataset class
class RetinalDataset(Dataset):
    def __init__(self, csv_file, root_dir, transform=None):
        self.annotations = pd.read_csv(csv_file)  # Read the CSV file
        self.root_dir = root_dir  # Root directory for images
        self.transform = transform  # Image transformations to apply

    def __len__(self):
        return len(self.annotations)  # Return the total number of samples

    def __getitem__(self, idx):
        # Get the filename from the 'file' column in the CSV
        filename = self.annotations.iloc[idx]['file']

        # Construct the full image path
        img_name = os.path.join(self.root_dir, filename)

        # Check if the file exists
        if not os.path.exists(img_name):
            raise FileNotFoundError(f"File {img_name} not found")

        # Load the image and convert to RGB
        image = Image.open(img_name).convert("RGB")

        # Get the diabetic retinopathy label from the 'final_icdr' column
        icdr_code = self.annotations.iloc[idx]['final_icdr']

        # Convert the 'final_icdr' code to binary classification: 0 for Not DR, 1 for DR
        label = 0 if icdr_code == 0 else 1

        # Apply transformations, if any
        if self.transform:
            image = self.transform(image)

        return image, label



dataset = RetinalDataset(csv_file='/content/drive/MyDrive/labels_mbrset.csv',
                         root_dir='/content/drive/MyDrive/images',
                         transform=image_transforms['train'])


# Split the dataset into 80% training and 20% testing using random_split
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
```

```
train_dataset, test_dataset = random_split(dataset, [train_size, test_size])


# Create DataLoader for both train and test datasets
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)  # Shuffle=True for random batches in training
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)  # Shuffle=False to preserve order in testing


# Load InceptionV3 pretrained model
model = models.inception_v3(pretrained=True)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/inception_v3_google-0cc3c7bd.pth" to /root/.cache/torch/hub/checkpoints/inception_v3_g
100%|██████████| 104M/104M [00:01<00:00, 84.8MB/s]
```

```
num_ftrs = model.fc.in_features  # Fetching the number of input features for the last layer
model.fc = nn.Linear(num_ftrs, 1)  # Change output layer to output 1 feature (binary classification)



model = model.to(device)


criterion = nn.BCEWithLogitsLoss()



optimizer = optim.Adam(model.parameters(), lr=0.001)


import numpy as np
# Early Stopping and Checkpointing setup
early_stopping_patience = 5  # Stop training if validation loss doesn't improve for 5 epochs
best_val_loss = np.inf
patience_counter = 0
checkpoint_path = 'best_model.pth'  # Path to save the best model


# Training loop
num_epochs = 15
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    # Training phase
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device).float().unsqueeze(1)

        optimizer.zero_grad()
        outputs = model(inputs)

        # Handle both main and auxiliary outputs during training
        if isinstance(outputs, tuple):
            logits = outputs[0]  # Main output is the first element
        else:
            logits = outputs  # If only one output is returned (e.g., in eval mode)

        # Calculate loss based on the main output
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    train_loss = running_loss / len(train_loader)

    # Validation phase
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device).float().unsqueeze(1)
            outputs = model(inputs)
```

```
            # During evaluation, only one output is returned
            logits = outputs  # No auxiliary output during eval mode
            loss = criterion(logits, labels)
            val_loss += loss.item()

    val_loss /= len(test_loader)
    print(f"Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}")

    # Early stopping and checkpointing
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save(model.state_dict(), checkpoint_path)  # Save best model
        patience_counter = 0
        print(f"New best model saved at epoch {epoch+1} with validation loss: {best_val_loss:.4f}")
    else:
        patience_counter += 1
        print(f"No improvement in validation loss. Patience: {patience_counter}/{early_stopping_patience}")

    if patience_counter >= early_stopping_patience:
        print("Early stopping triggered. Training terminated.")
        break
```

```
Epoch [1/15], Train Loss: 0.4653, Val Loss: 0.4387
New best model saved at epoch 1 with validation loss: 0.4387
Epoch [2/15], Train Loss: 0.3875, Val Loss: 0.4669
No improvement in validation loss. Patience: 1/5
Epoch [3/15], Train Loss: 0.3656, Val Loss: 0.3560
New best model saved at epoch 3 with validation loss: 0.3560
Epoch [4/15], Train Loss: 0.3629, Val Loss: 0.3949
No improvement in validation loss. Patience: 1/5
Epoch [5/15], Train Loss: 0.3337, Val Loss: 0.3856
No improvement in validation loss. Patience: 2/5
Epoch [6/15], Train Loss: 0.3172, Val Loss: 0.3588
No improvement in validation loss. Patience: 3/5
Epoch [7/15], Train Loss: 0.2866, Val Loss: 0.3877
No improvement in validation loss. Patience: 4/5
Epoch [8/15], Train Loss: 0.2792, Val Loss: 0.3837
No improvement in validation loss. Patience: 5/5
Early stopping triggered. Training terminated.
```

```
# Load the best model for evaluation
model.load_state_dict(torch.load(checkpoint_path))
model.eval()
```

```
        )
        (branch3x3dbl_2): BasicConv2d(
          (conv): Conv2d(448, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        )
        (branch3x3dbl_3a): BasicConv2d(
          (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
          (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        )
        (branch3x3dbl_3b): BasicConv2d(
          (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
          (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        )
        (branch_pool): BasicConv2d(
          (conv): Conv2d(2048, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
      (dropout): Dropout(p=0.5, inplace=False)
      (fc): Linear(in_features=2048, out_features=1, bias=True)
    )
```

```
# Test model on the test set
all_preds = []
all_labels = []
with torch.no_grad():
    for inputs, labels in test_loader:
        inputs = inputs.to(device)
        outputs = model(inputs)
        preds = torch.sigmoid(outputs).cpu().numpy() > 0.5  # Convert outputs to binary predictions
        all_preds.extend(preds)
        all_labels.extend(labels.numpy())
```

```
# Convert predictions to integer format
all_preds = [int(p[0]) for p in all_preds]
all_labels = [int(l) for l in all_labels]
```

```
# Calculate evaluation metrics
accuracy = accuracy_score(all_labels, all_preds)
precision = precision_score(all_labels, all_preds)
recall = recall_score(all_labels, all_preds)
f1 = f1_score(all_labels, all_preds)

print(f"Test Accuracy: {accuracy * 100:.2f}%")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1:.2f}")
```

```
Test Accuracy: 84.03%
Precision: 0.79
Recall: 0.60
F1 Score: 0.68
```