Early Detection of Diabetic Retinopathy: A Deep Learning Approach Using ResNet-50 and DenseNet

DEVELOPED BY: SUBHAM PATNAIK

1. Abstract

Diabetic Retinopathy (DR) is a severe complication of diabetes and stands as one of the most common causes of blindness globally. The early detection and automated grading of DR from retinal fundus images is a well-studied problem in the medical imaging field.

This project aims to develop a robust deep learning model for the automated classification of Diabetic Retinopathy severity. The model is trained on the APTOS 2019 Blindness Detection dataset and is designed to classify retinal images into five distinct severity grades. To achieve high accuracy, this project implements a custom hybrid Convolutional Neural Network (CNN) that combines the architectural strengths of both the ResNet-50 and DenseNet models.

Using **Convolutional Neural Networks (CNNs),** the system will learn to identify key pathological features, or "signatures", of the disease. In its early layers, the model learns to find simple features like edges and small dots. In deeper layers, it combines these to recognize complex patterns such as microaneurysms , hemorrhages , hard exudates , and the abnormal growth of new blood vessels.

1.1 ResNet-50

The key innovation of ResNet (Residual Network) is its use of "skip connections" or "residual blocks".

Core Idea: In traditional deep networks, the model has to learn a complete, complex transformation from one layer to the next. ResNet changes this: it assumes the easiest thing to learn is nothing (an identity). The network's layers are only asked to learn the difference, or **residual**, between the input and the desired output.

How it Works: The skip connection takes the input to a block of layers and adds it directly to the output of that block. This creates a "highway" for the gradient to flow backward during training, which solves the "vanishing gradient" problem that plagues very deep networks.

The "Bottleneck" Block: Your file specifically mentions ResNet-50 uses "bottleneck" residual blocks. This is a key optimization. Instead of using two large 3 X 3 convolutions, the bottleneck block does this:

- A **1 X 1 convolution** to reduce the number of feature channels (e.g., from 256 to 64).
- A **3 X 3 convolution** to perform the main spatial feature extraction on this smaller set of channels.
- A **1 X 1 convolution** to restore the number of channels (e.g., from 64 back to 256).

This "bottleneck" design provides the same deep representation with significantly fewer parameters and less computation.

1.2 DenseNet

DenseNet (Densely Connected Convolutional Network) takes the idea of "skip connections" to an extreme.

Core Idea: Instead of just connecting the input of a block to its output, DenseNet connects every layer to every other layer in a "dense block".

How it Works: This is the most important difference from ResNet. DenseNet does **not** add the feature maps. It **concatenates** them.

- Layer 1's output is fed to Layer 2.
- Layer 2's output is concatenated with Layer 1's output, and that is fed to Layer 3.
- Layer 3's output is concatenated with Layer 2's and Layer 1's output, and so on.

Key Concepts:

- **Feature Reuse:** This design encourages massive feature reuse. Early layers' simple features (like edges) are directly available to much deeper layers, allowing the network to learn more complex and efficient representations.
- **Growth Rate (K):** Because so many features are being concatenated, each individual layer only needs to produce a very small number of new feature maps. This hyperparameter is called the "growth rate" (e.g., k = 12or k = 32). This is why DenseNet is extremely parameter-efficient.
- **Transition Layers:** As described in your file, blocks of densely connected layers are separated by "transition layers." These layers (usually a $1 \times 1$ convolution followed by a 2 x 2 pooling layer) are responsible for downsampling the feature maps and reducing the channel count, keeping the model manageable.

1.3 Combination of ResNet-50 and DenseNet for a Custom CNN

step-by-step explanation of exactly how your CombinedNet class works:

1. The Setup (in __init__)

First, the two models are set up to act as independent feature extractors:

- Load Models: A pre-trained resnet50 and a pre-trained densenet121 are loaded from torchvision.models.
- Freeze Weights: All original layers in both models are "frozen" by setting param.requires_grad = False. This is crucial as it preserves the powerful, general-purpose features learned from the ImageNet dataset.
- Replace Classifiers: This is the primary customization.

1. The resnet.fc layer is removed and replaced with a new, trainable nn.Linear layer that outputs 512 features.
2. The densenet.classifier layer is similarly removed and replaced with a new, trainable nn.Linear layer that also outputs 512 features.

- Create a New "Head": A final, new classifier block (self.fc) is defined. Its first layer, nn.Linear(1024, 256), is the key. This 1024 input is specifically designed to accept the 512 features from ResNet plus the 512 features from DenseNet (512 + 512 = 1024).

2. The Execution (in forward)

The forward function defines what happens every time an image passes through the model:

1. **Parallel Extraction:** The input image (x) is fed into both modified backbones simultaneously.

- x1 = self.resnet(x): The image goes through the modified ResNet-50, which outputs a 512-feature vector.
- x2 = self.densenet(x): The same image goes through the modified DenseNet-121, outputting its own 512-feature vector.

2. **Combination (Concatenation):**

- x = torch.cat((x1, x2), dim=1): This is the exact line where the combination happens. torch.cat is used to concatenate (join) the two vectors. This stacks the 512 features from ResNet and the 512 features from DenseNet side-by-side, creating one single, larger 1024-feature vector.

3. **Final Classification:**

- x = self.fc(x): This new, combined 1024-feature vector is fed into the final classifier (self.fc). This trainable head processes this rich, combined vector and outputs the final 5 class scores for the diabetic retinopathy grades.

2. Understanding Diabetic Retinopathy

Diabetic Retinopathy (DR) is a significant medical condition and a primary cause of blindness. It is a well-studied problem in the medical imaging field, where the goal is to identify damage to the retina caused by diabetes.

To understand the disease, it's helpful to first know what a Normal Retina looks like. Your project file identifies key components of a healthy retina, including the Fovea , Macula , Optic Disc , and the network of Retinal Arteries and Veins.

In contrast, a retina with Diabetic Retinopathy is characterized by the appearance of various lesions and abnormalities. These are the visual "signatures" that your deep learning model will be trained to detect.

2.1 Key Pathological Features

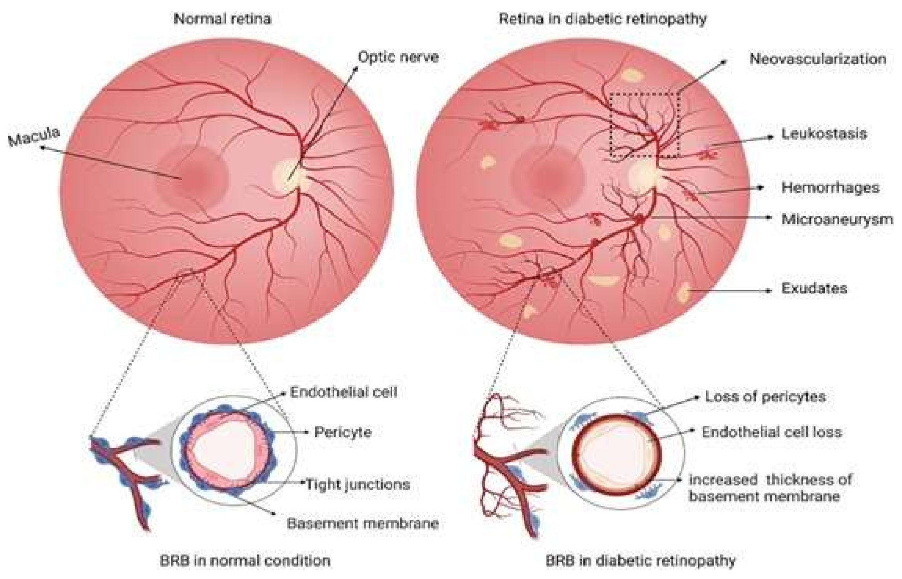
several common features that indicate the presence and severity of Diabetic Retinopathy. The model learns to identify these features, from simple dots to complex patterns:

- **Microaneurysms:** These are described as small red dots , also pictured as Aneurysms. They are caused by swellings in the blood vessels and are considered an early indicator of DR.
- **Hemorrhages:** These are larger bleeding spots in the retina.
- **Exudates (Hard Exudates):** These appear as white or yellowish deposits of lipids and proteins.
- **Cotton Wool Spots:** These are white, fluffy lesions caused by an infarction (loss of blood supply) in the nerve fiber layer.
- **Neovascularization (Abnormal Growth of Blood Vessels):** This is the formation of new, abnormal blood vessels. This specific feature is the hallmark of the most severe stage, Proliferative DR.
- **Macular Edema:** This is a swelling that occurs near the macula , the part of the retina responsible for sharp, central vision.
- **General Abnormalities:** The model also learns to see more general changes, such as abnormalities in the shape, caliber, or tortuosity of blood vessels and variations in the retina's overall texture and color.

2.2 How These Features Relate to the Severity Scale

Your project's goal is to classify images based on a standard 0-to-4 severity scale. The presence, number, and type of the lesions described above determine an image's grade:

- **0 - No DR:** A healthy retina, free of these lesions.
- **1 - Mild:** Typically characterized by the appearance of early-stage features like microaneurysms.
- **2 - Moderate:** A progression from the mild stage, with more numerous lesions.
- **3 - Severe:** Characterized by a significant increase in features like hemorrhages, exudates, and cotton wool spots.
- **4 - Proliferative DR:** The most advanced stage, defined by the presence of neovascularization (new abnormal blood vessel growth).



3. Dataset and Preprocessing

3.1. Dataset

The project utilizes the APTOS 2019 Blindness Detection Dataset. This dataset consists of a large set of retinal fundus images.

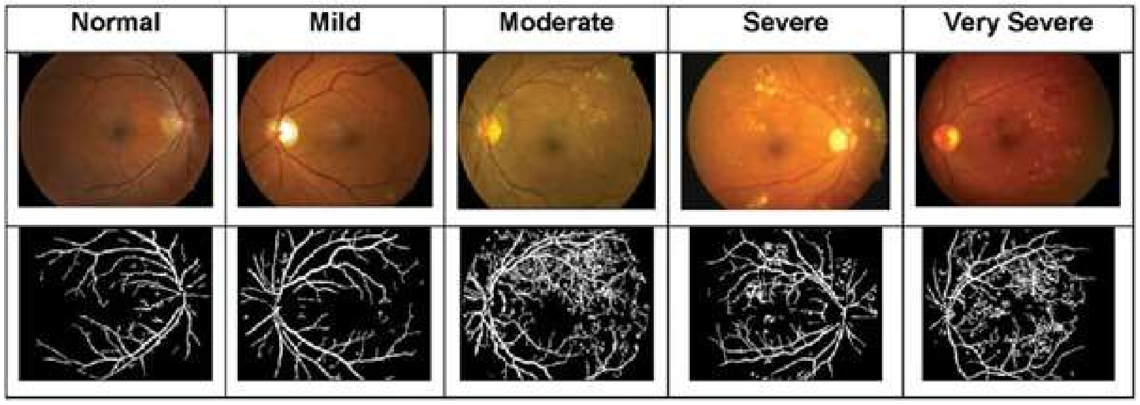
- Training Set Size: ~3,662 images
- Test Set Size: ~1,928 images

The provided train.csv file is split into training (80%) and validation (20%) sets. This split is stratified to ensure that the distribution of all severity classes is maintained in both sets.

3.2. Severity Grading

Each image in the dataset is rated for the severity of diabetic retinopathy on a 5-point scale from 0 to 4:

- 0: No DR
- 1: Mild
- 2: Moderate
- 3: Severe
- 4: Proliferative DR



3.3. Image Preprocessing and Augmentation

A custom APTOSDataset class is used to load and transform the images. Two distinct transformation pipelines are created for training and validation:

Training Transforms:

- Resize: All images are resized to 224 X 224 pixels.
- RandomHorizontalFlip: Randomly flips images horizontally to augment the dataset.
- RandomRotation: Randomly rotates images by up to 15 degrees.
- ColorJitter: Randomly adjusts the brightness, contrast, and saturation of the images.
- ToTensor: Converts the PIL Image to a PyTorch Tensor.
- Normalize: Normalizes the tensor with standard ImageNet mean and standard deviation.

Validation Transforms:

- Resize: All images are resized to 224x224 pixels.
- ToTensor: Converts the PIL Image to a PyTorch Tensor.
- Normalize: Normalizes the tensor.

4. Model Architecture

This project employs a custom hybrid architecture, CombinedNet, which utilizes a parallel feature fusion approach. The core idea is to leverage the unique strengths of two powerful, pre-trained CNNs: ResNet-50 and DenseNet.

4.1. Base Model 1: ResNet-50

ResNet-50 is a 50-layer deep convolutional neural network. Its key innovation is the "skip connection," also known as a residual block.

- **Core Concept:** These connections allow the model to learn a "residual" correction on top of the input, which helps solve the vanishing gradient problem in very deep networks.
- **Optimization:** The architecture uses "bottleneck" residual blocks to reduce computational cost.

4.2. Base Model 2: DenseNet

DenseNet (Densely Connected Convolutional Network) uses a different approach where each layer is connected to every other layer within a "dense block".

- **Core Concept:** It works by **concatenating** the feature maps from all preceding layers.
- **Benefits:** This design encourages maximum feature reuse, improves gradient flow, and typically requires fewer parameters than ResNet.

4.3. The CombinedNet Hybrid Architecture

The custom CombinedNet model is implemented as follows:

1. **Load Backbones:** A pre-trained resnet50 and a pre-trained densenet121 are loaded from torchvision.models.
2. **Freeze Weights:** All parameters in both pre-trained models are frozen (requires_grad = False) so they act as fixed feature extractors.
3. **Replace Classifiers:** The final classification layer of each model is removed.
   - The resnet.fc layer is replaced with a new nn.Linear layer that outputs **512 features**.
   - The densenet.classifier layer is replaced with a new nn.Linear layer that also outputs **512 features**.
4. **Feature Fusion (Forward Pass):**
   - An input image is passed through both the ResNet and DenseNet backbones in parallel.
   - This generates two separate 512-feature vectors: x1 (from ResNet) and x2 (from DenseNet).
   - These two vectors are **concatenated** using torch.cat to create a single, combined **1024-feature vector**.
5. **New Classification Head:** This 1024-feature vector is fed into a new, trainable classifier head (self.fc), which consists of:
   - A Linear layer (1024 →256)
   - A ReLU activation function

- A Dropout layer (with a 0.4 probability) to prevent overfitting
- A final Linear layer (256 → 5) to output the raw scores for the 5 DR classes.

5. Implementation and Training

The model is trained using the PyTorch framework.

- **Loss Function:** nn.CrossEntropyLoss is used, as it is well-suited for multi-class classification problems.
- **Optimizer:** The Adam optimizer is employed with an initial learning rate of 1e-4 to train the new classification head.
- **Learning Rate Scheduler:** A ReduceLROnPlateau scheduler monitors the validation loss. It will decrease the learning rate if the model's performance on the validation set stops improving.
- **Training Loop:** The model is trained for 15 epochs.
- **Evaluation:** After each epoch, the model is evaluated on the validation set using **Accuracy** and **macro-averaged F1 Score**.
- **Checkpointing:** The best-performing model (based on the highest validation accuracy) is saved as "best_combined_model.pth".

5.1 Source Code

5.1.1. Importing Required Libraries

```
import os

import numpy as np

import pandas as pd

from tqdm import tqdm

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, f1_score

from PIL import Image
```

5.1.2 import torch

```
import torch.nn as nn
import torch.optim as optim

from torch.optim import lr_scheduler

from torch.utils.data import Dataset, DataLoader

import torchvision.transforms as transforms

import torchvision.models as models
```

5.1.3. Dataset Setup and Splitting

```
DATA_DIR = "aptos2019"

TRAIN_CSV = "train.csv"

TRAIN_IMG_DIR = os.path.join(DATA_DIR, "train_images")

df = pd.read_csv(TRAIN_CSV)

train_df, val_df = train_test_split(df, test_size=0.2, stratify=df['diagnosis'], random_state=42)
```

5.1.4. Custom PyTorch Dataset Class

```
class APTOSDataset(Dataset):

  def _init_(self, df, img_dir, transform=None):

    self.df = df

    self.img_dir = img_dir

    self.transform = transform


  def _len_(self):

    return len(self.df)


  def _getitem_(self, idx):

    row = self.df.iloc[idx]

    img_path = os.path.join(self.img_dir, row['id_code'] + ".png")

    image = Image.open(img_path).convert("RGB")

    label = torch.tensor(int(row['diagnosis']))
```

```
        if self.transform:
            image = self.transform(image)
        return image, label
```

5.1.5. Image Transformations

```
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])
val_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])
```

5.1.6. Dataloader Creation

```
train_dataset = APTOSDataset(train_df, TRAIN_IMG_DIR, transform=train_transform)
val_dataset = APTOSDataset(val_df, TRAIN_IMG_DIR, transform=val_transform)


train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)
```

5.1.7. Combined Deep Learning Model (ResNet + DenseNet)

```
class CombinedNet(nn.Module):
  def _init_(self):
    super(CombinedNet, self)._init_()
    self.resnet = models.resnet50(pretrained=True)
    self.densenet = models.densenet121(pretrained=True)

    # Freeze pretrained weights
    for param in self.resnet.parameters():
      param.requires_grad = False
    for param in self.densenet.parameters():
      param.requires_grad = False

    # Replace final layers
    self.resnet.fc = nn.Linear(self.resnet.fc.in_features, 512)
    self.densenet.classifier = nn.Linear(self.densenet.classifier.in_features, 512)

    # Combined fully connected classifier
    self.fc = nn.Sequential(
      nn.Linear(1024, 256),
      nn.ReLU(),
      nn.Dropout(0.4),
      nn.Linear(256, 5)
    )

  def forward(self, x):
    x1 = self.resnet(x)
```

```
        x2 = self.densenet(x)

        x = torch.cat((x1, x2), dim=1)

        x = self.fc(x)

        return x
```

5.1.8. Training Setup

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

model = CombinedNet().to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=1e-4)

scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', patience=3, factor=0.3)

best_acc = 0
```

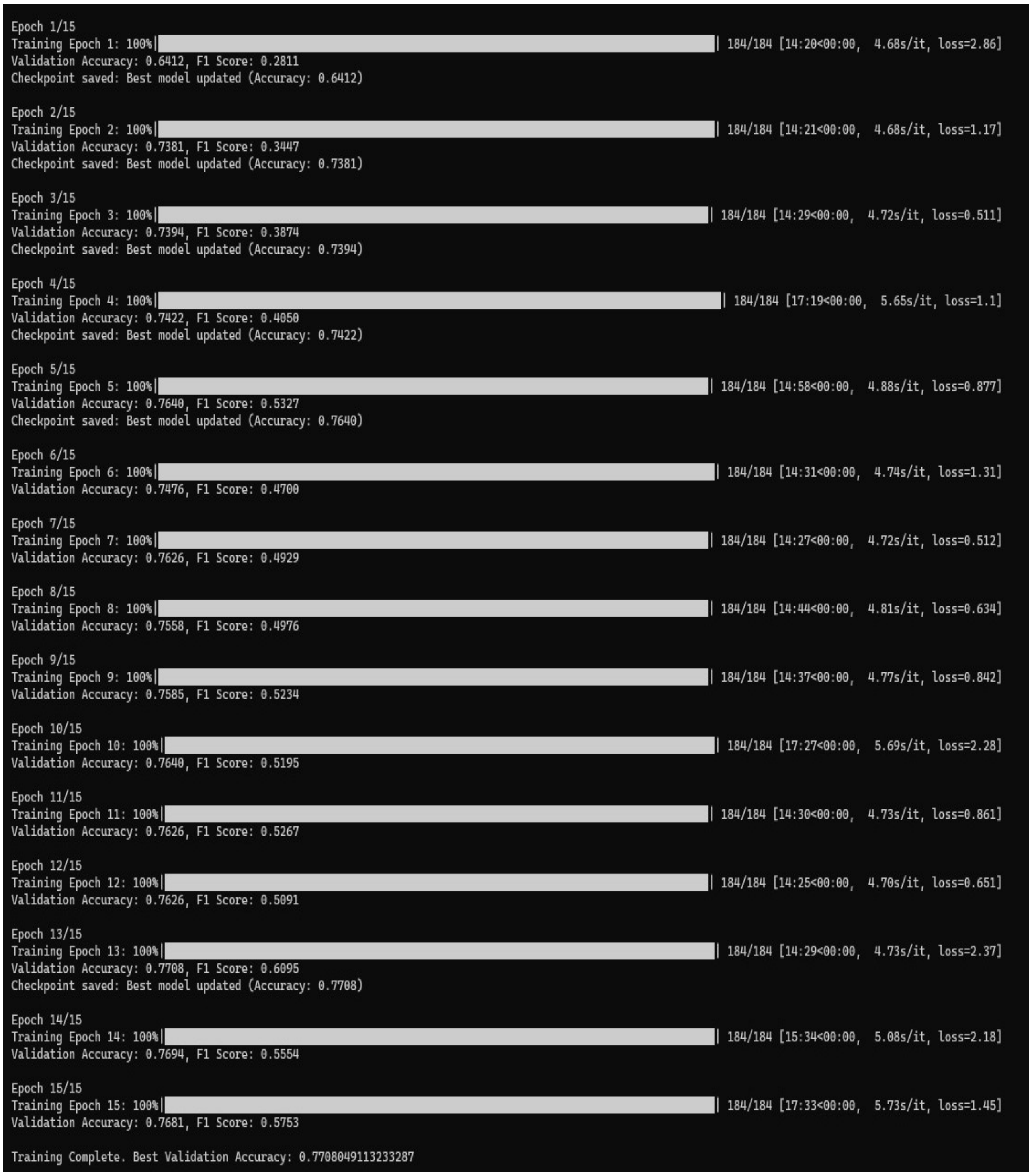
5.1.9. Training & Validation Loop

```
for epoch in range(15):

    print(f"\nEpoch {epoch + 1}/15")

    model.train()

    running_loss = 0.0


    with tqdm(train_loader, desc=f"Training Epoch {epoch+1}", leave=True) as t_bar:

        for images, labels in t_bar:

            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()

            outputs = model(images)

            loss = criterion(outputs, labels)

            loss.backward()

            optimizer.step()

            running_loss += loss.item()
        t_bar.set_postfix(loss=loss.item())


    model.eval()
    preds, truths = [], []
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            preds.extend(predicted.cpu().numpy())
            truths.extend(labels.cpu().numpy())


    acc = accuracy_score(truths, preds)
    f1 = f1_score(truths, preds, average='macro')
    scheduler.step(running_loss)
    print(f"Validation Accuracy: {acc:.4f}, F1 Score: {f1:.4f}")
```

5.1.10. Checkpoint Saving

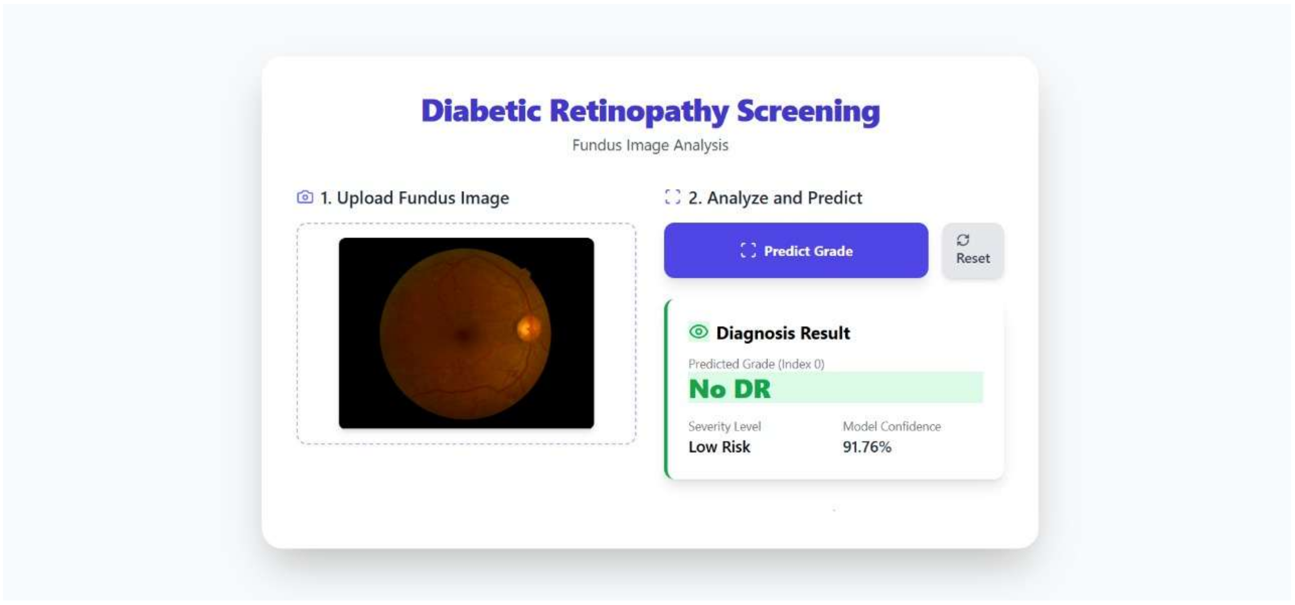
```
    checkpoint = {
        'epoch': epoch + 1,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'best_val_acc': best_acc
    }
    torch.save(checkpoint, f"checkpoint_epoch_{epoch+1}.pth")


    if acc > best_acc:
        best_acc = acc

        torch.save(model.state_dict(), "best_combined_model.pth")

        print(f"Checkpoint saved: Best model updated (Accuracy: {best_acc:.4f})")

print("\nTraining Complete. Best Validation Accuracy:", best_
```

5.2 Output:



```
Epoch 1/15
Training Epoch 1: 100%|████████████████████████████████████████| 184/184 [14:20<00:00,  4.68s/it, loss=2.86]
Validation Accuracy: 0.6412, F1 Score: 0.2811
Checkpoint saved: Best model updated (Accuracy: 0.6412)

Epoch 2/15
Training Epoch 2: 100%|████████████████████████████████████████| 184/184 [14:21<00:00,  4.68s/it, loss=1.17]
Validation Accuracy: 0.7381, F1 Score: 0.3447
Checkpoint saved: Best model updated (Accuracy: 0.7381)

Epoch 3/15
Training Epoch 3: 100%|████████████████████████████████████████| 184/184 [14:29<00:00,  4.72s/it, loss=0.511]
Validation Accuracy: 0.7394, F1 Score: 0.3874
Checkpoint saved: Best model updated (Accuracy: 0.7394)

Epoch 4/15
Training Epoch 4: 100%|████████████████████████████████████████| 184/184 [17:19<00:00,  5.65s/it, loss=1.1]
Validation Accuracy: 0.7422, F1 Score: 0.4050
Checkpoint saved: Best model updated (Accuracy: 0.7422)

Epoch 5/15
Training Epoch 5: 100%|████████████████████████████████████████| 184/184 [14:58<00:00,  4.88s/it, loss=0.877]
Validation Accuracy: 0.7640, F1 Score: 0.5327
Checkpoint saved: Best model updated (Accuracy: 0.7640)

Epoch 6/15
Training Epoch 6: 100%|████████████████████████████████████████| 184/184 [14:31<00:00,  4.74s/it, loss=1.31]
Validation Accuracy: 0.7476, F1 Score: 0.4700

Epoch 7/15
Training Epoch 7: 100%|████████████████████████████████████████| 184/184 [14:27<00:00,  4.72s/it, loss=0.512]
Validation Accuracy: 0.7626, F1 Score: 0.4929

Epoch 8/15
Training Epoch 8: 100%|████████████████████████████████████████| 184/184 [14:44<00:00,  4.81s/it, loss=0.634]
Validation Accuracy: 0.7558, F1 Score: 0.4976

Epoch 9/15
Training Epoch 9: 100%|████████████████████████████████████████| 184/184 [14:37<00:00,  4.77s/it, loss=0.842]
Validation Accuracy: 0.7585, F1 Score: 0.5234

Epoch 10/15
Training Epoch 10: 100%|███████████████████████████████████████| 184/184 [17:27<00:00,  5.69s/it, loss=2.28]
Validation Accuracy: 0.7640, F1 Score: 0.5195

Epoch 11/15
Training Epoch 11: 100%|███████████████████████████████████████| 184/184 [14:30<00:00,  4.73s/it, loss=0.861]
Validation Accuracy: 0.7626, F1 Score: 0.5267

Epoch 12/15
Training Epoch 12: 100%|███████████████████████████████████████| 184/184 [14:25<00:00,  4.70s/it, loss=0.651]
Validation Accuracy: 0.7626, F1 Score: 0.5091

Epoch 13/15
Training Epoch 13: 100%|███████████████████████████████████████| 184/184 [14:29<00:00,  4.73s/it, loss=2.37]
Validation Accuracy: 0.7708, F1 Score: 0.6095
Checkpoint saved: Best model updated (Accuracy: 0.7708)

Epoch 14/15
Training Epoch 14: 100%|███████████████████████████████████████| 184/184 [15:34<00:00,  5.08s/it, loss=2.18]
Validation Accuracy: 0.7694, F1 Score: 0.5554

Epoch 15/15
Training Epoch 15: 100%|███████████████████████████████████████| 184/184 [17:33<00:00,  5.73s/it, loss=1.45]
Validation Accuracy: 0.7681, F1 Score: 0.5753

Training Complete. Best Validation Accuracy: 0.7708049113233287
```

5.3 User Interface



Diabetic Retinopathy Screening

Fundus Image Analysis

1. Upload Fundus Image

2. Analyze and Predict

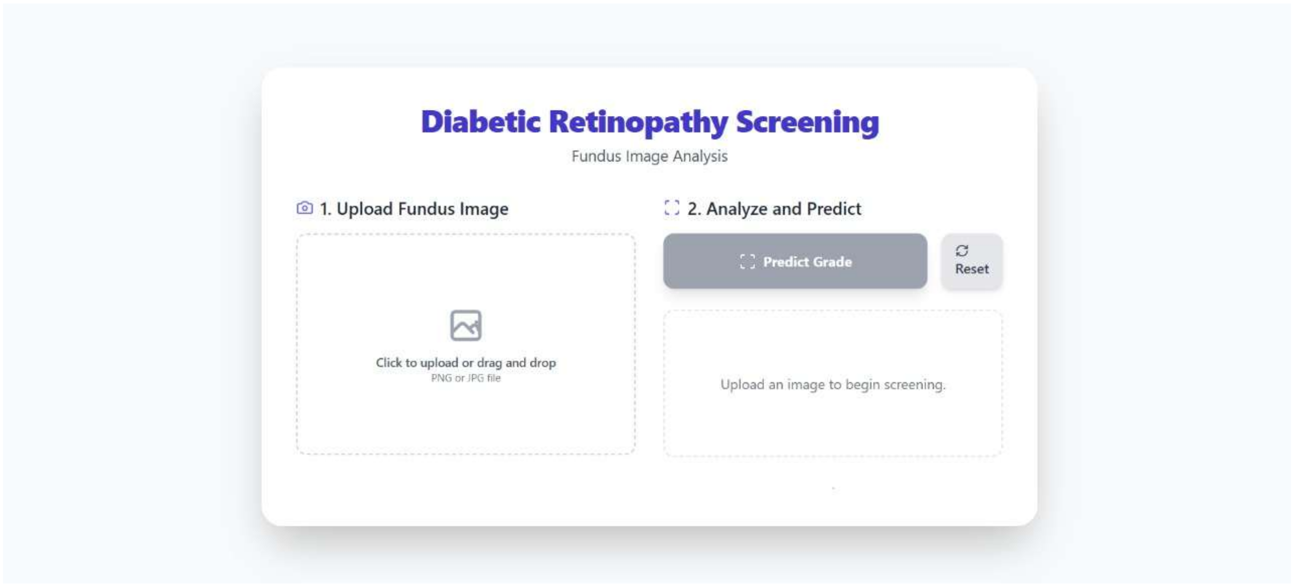
Diagnosis Result

Predicted Grade (Index 0)

No DR

Severity Level: Low Risk

Model Confidence: 91.76%



Diabetic Retinopathy Screening

Fundus Image Analysis

1. Upload Fundus Image

2. Analyze and Predict

Click to upload or drag and drop

PNG or JPG file

Upload an image to begin screening.

6. Results and Discussion

6.1. Model Training Performance

The custom CombinedNet model was trained for **15 epochs**. The training process was monitored using both training accuracy and validation accuracy, with the best-performing model being saved based on the highest validation accuracy achieved.

The training log (see figure below) reveals several key insights:

- **Training Accuracy:** The model achieved 100% training accuracy within the first epoch and maintained it throughout the training. This indicates that the model has a very high capacity and was able to perfectly "memorize" the training dataset.
- **Validation Accuracy:** The validation accuracy, which measures the model's ability to generalize to new data, provides a more realistic performance metric.
    - The model's validation accuracy started at **64.12%** in Epoch 1.
    - It progressively improved, reaching a **peak validation accuracy of 77.08%** in Epoch 13.
    - After this peak, the performance on the validation set began to plateau or slightly decrease, ending at 76.81% in Epoch 15.
- **Overfitting:** The significant gap between the 100% training accuracy and the ~77% validation accuracy is a clear sign of **overfitting**. While the model learned the training data perfectly, it did not generalize that knowledge as effectively to the unseen validation data. This is a common challenge in deep learning, and it suggests that the model's complexity might be too high for the amount of training data, or that more regularization is needed.
- **Checkpointing:** The training script successfully saved the model from **Epoch 13** as "best_combined_model.pth", as this epoch yielded the highest validation accuracy (0.7708). This is the model used for the final application.

2. Application & Deployment

The saved best_combined_model.pth was integrated into a web-based "Diabetic Retinopathy Screening" tool. The user interface (UI) demonstrates the practical application of the model.

- **Initial State:** The tool presents a clean interface where the user can upload or drag-and-drop a fundus image file.
- **Prediction State:** After an image is uploaded and the "Predict Grade" button is clicked, the model analyzes the image and displays the results.
    1. **Diagnosis:** The output clearly shows the predicted grade, such as **"No DR (Index 0)"**.

2. **Confidence:** It provides a **Model Confidence** score (e.g., 91.76%), giving the user an understanding of the model's certainty.
3. **Severity:** It translates the numeric grade into a human-readable **Severity Level**, such as **"Low Risk"**.

This UI successfully abstracts the complex CombinedNet model into an easy-to-use screening tool.

Conclusion

This project successfully developed and implemented a custom deep learning model, CombinedNet, for the classification of Diabetic Retinopathy. By using **parallel feature fusion** to combine the strengths of both ResNet-50 and DenseNet-121, the model achieved a peak **validation accuracy of 77.08%** on the APTOS 2019 dataset.

The model was then successfully deployed into a functional web application, proving its utility as a practical screening tool. The tool can load a fundus image, process it, and return a clear, understandable diagnosis, including severity level and model confidence.