A Project Report
On
“Implementation of Vision Transformer on Diabetic Retinopathy Dataset”

Under the Supervision of Dr. Anshul Sharma

Department of Computer Science & Engineering

NATIONAL INSTITUTE OF TECHNOLOGY PATNA

University Campus, Bihta - 801103

Submitted by:

Group No.-21

Name	Roll No.
Harsh Anand	2206038
Srijan Sarvshresth	2106010
Gobinda Prasad Bag	2206005

Course Code: CS64123

Table of Contents

Problem Statement:

The aim of this project is to automate the grading of diabetic retinopathy using fundus (retinal) images. Traditional methods often struggle to detect fine-grained lesions, which are small but important indicators of the disease. To overcome this limitation, we propose using a Vision Transformer (ViT), a deep learning model that uses attention mechanisms to focus on the most relevant parts of the image. This approach not only improves the accuracy of detection but also enhances interpretability, allowing medical professionals to understand which areas influenced the model's decisions. ViT offers a powerful and efficient solution for better retinal disease diagnosis.

Motivation:

Diabetic Retinopathy (DR) is a leading cause of blindness worldwide, and early detection is crucial to prevent vision loss. However, in many rural and underserved areas, there is a shortage of eye care specialists, making timely diagnosis difficult. While Convolutional Neural Networks (CNNs) have been widely used for medical image analysis, they often struggle to detect long-range dependencies between lesions, which are essential for accurate DR grading. Vision Transformers (ViTs) offer a promising solution by capturing global context, scaling effectively with large datasets, and using attention mechanisms to focus on key regions, thereby improving both accuracy and interpretability.

Dataset Description:

The dataset used is sourced from the **Kaggle Diabetic Retinopathy Detection** competition, as referenced in the research paper titled *"Lesion-Aware Transformers for Diabetic Retinopathy Grading"* (CVPR 2021). It consists of **35,126 training images** and **53,576 testing images**

of retinal fundus photographs. The training labels are provided in a file named `trainingLabels`, which includes two columns: `image` (image filename) and `level` (severity of diabetic retinopathy). There are **5 classification levels** (from **0 to 4**), indicating increasing severity of the disease.

The Proposed Methodology:

Model Architectures used :

- Vision Transformer(ViT)
- CNN

Vision Transformer:

A **Vision Transformer (ViT)** is a type of deep learning model used to analyze images. It is based on the **Transformer architecture**, which was originally designed for natural language processing (NLP) tasks, like translating languages or understanding text. ViT applies this same concept to images.

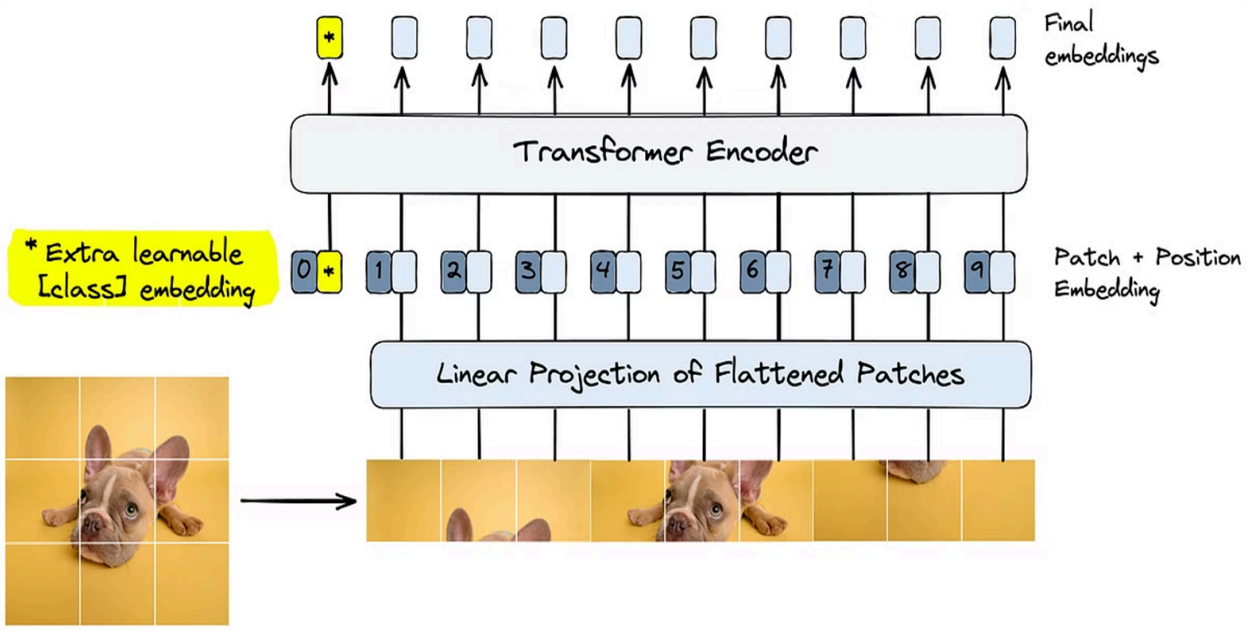
Applications of Vision Transformers:

Image classification

Object detection

Image segmentation

Medical image analysis



Principles of ViT:

Key Principles:Breaks down images into patches, similar to words in a sentence.

Breakthrough Design:Applies transformer layers directly to these patches.

Global Context: Enables the model to capture global relationships within the image.

Working of ViT:

Splitting the Image into Patches- Instead of processing the whole image at once (like CNNs do), the Vision Transformer splits the image into smaller patches. Each of these patches is treated like a separate unit, similar to how words are treated in a Transformer for language tasks.

Linear Projection of Flattened Patches- Each patch (small image) is flattened (converted into a long 1D list of numbers instead of a 2D square).Then, a linear projection (a simple mathematical transformation) is applied to convert the patch into a vector of

numbers. This step turns image patches into meaningful numerical representations that the Transformer can understand.

Adding Positional Encoding- Transformers don't inherently understand spatial relationships (where each patch comes from in the image).To help with this, Positional Encoding is added to each patch's numerical representation. This ensures that the Transformer knows where each patch was originally located in the image.

Adding the Special [CLS] Token- A special token called the [CLS] (Class Token) is added to the sequence. This token acts as a summary of the entire image and will later be used to make the final classification. This is similar to how Transformers process text, where a special token is used to gather all the necessary information.

Transformer Encoder- The Transformer Encoder processes all the patch embeddings together using Multi-Head Self-Attention (MHSA) and Feedforward Neural Networks. This allows the model to learn relationships between different parts of the image (e.g., how the dog's nose relates to its eyes).This is the core of the Vision Transformer, as it enables the model to understand the image globally.

Output Final Embeddings- After passing through the Transformer Encoder, each patch now has a refined numerical representation. The [CLS] token stores the most important information, summarizing the whole image.

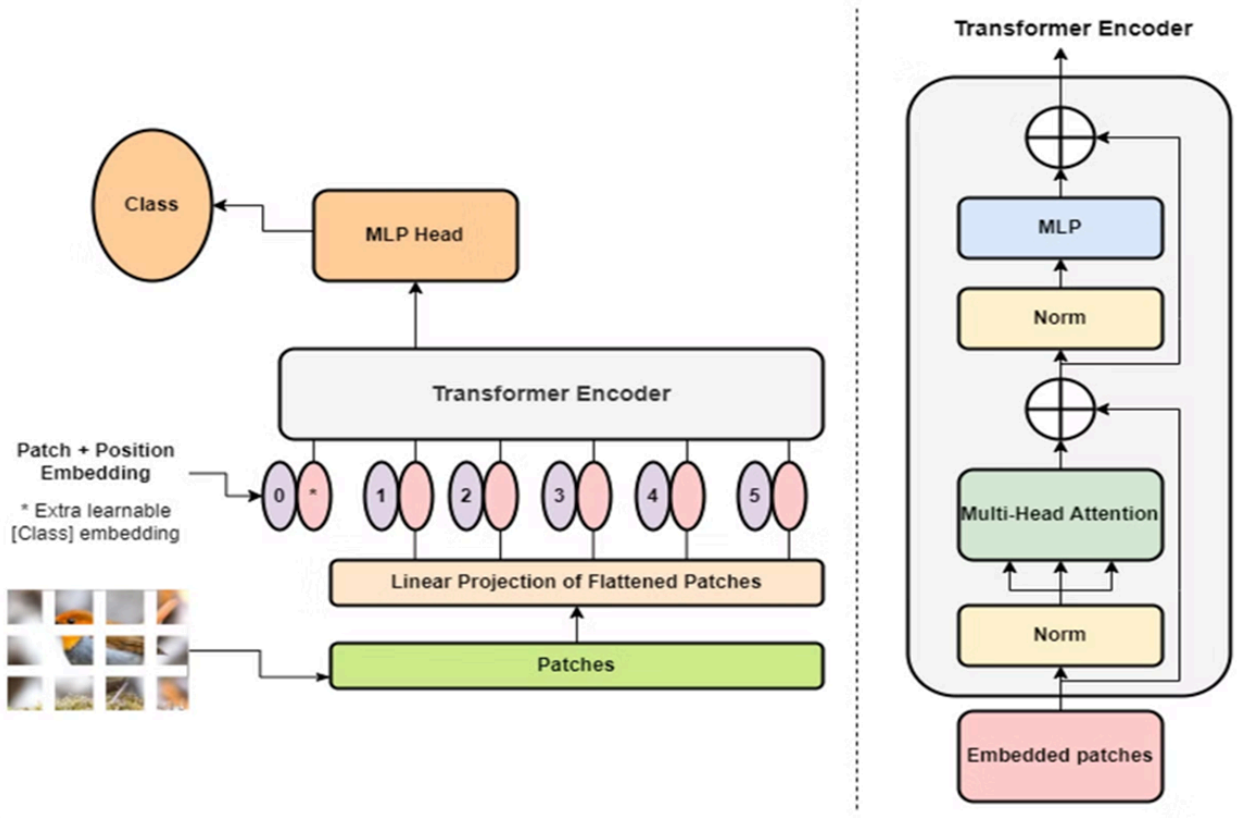
Final Classification- The final representation of the [CLS] token is sent to a simple Feedforward Neural Network (MLP head). This network makes the final prediction about what the image contains (e.g., dog, cat, airplane, etc.). The output is passed through a Softmax function to determine the final class probabilities.

Summary of the Entire Process 1. Divide Image into Patches → Cut the image into small square sections.2. Convert Patches into Vectors → Flatten each patch and apply a transformation.3. Add Positional Encoding → Help the Transformer understand spatial information.4. Add a Special [CLS] Token → A token that will store the final image representation.5. Process Through Transformer Encoder → Learn

relationships between patches using self-attention. 6.Extract the Final Embedding → The refined representation of the image.7. Pass Through Classifier → Predict the final label (e.g., dog, cat).

Why is Vision Transformer Important? Captures long-range dependencies → Understands relationships between distant parts of the image. More flexible than CNNs → Doesn't rely on fixed filters like convolutional networks. Scales well with data → Performs well when trained on large datasets.

How Image is Processed:



CNN:

Introduction:

Convolutional Neural Networks (CNNs) are a type of deep learning model that are very effective for analyzing images. In this research, CNNs are used to automatically detect **diabetic retinopathy (DR)**—a disease that damages the retina due to diabetes.

The main idea behind CNNs is to allow computers to "see" and learn patterns from images in a way that mimics how the human brain works. For example, CNNs can identify key features in eye images such as **blood vessel damage, hemorrhages**, or **exudates**, which are important for diagnosing DR.

CNNs reduce the need for manual feature extraction. Instead of telling the computer what to look for, CNNs learn what features are important directly from the images.

CNN Architecture:

A CNN is made up of several key layers, each performing a specific task:

Input Layer:

Takes in the original image (e.g., a fundus photo of the retina).

Convolutional Layers:

These layers apply filters (small matrices) to the image to extract features such as edges, textures, or shapes.

Early layers detect simple features, while deeper layers detect more complex patterns like lesions or blood vessel abnormalities.

Activation Functions (e.g., ReLU):

These introduce non-linearity, helping the network learn complex patterns.

Pooling Layers (e.g., Max Pooling):

Reduce the size of the feature maps and make the model faster and more robust by keeping only the most important information.

Fully Connected Layers:

These layers make final predictions by combining all the extracted features and mapping them to output classes (e.g., different stages of DR: no DR, mild, moderate, etc.).

Output Layer (Softmax):

Produces probabilities for each DR class, and the class with the highest probability is the final prediction.

An example of CNN Architecture for image Classification:

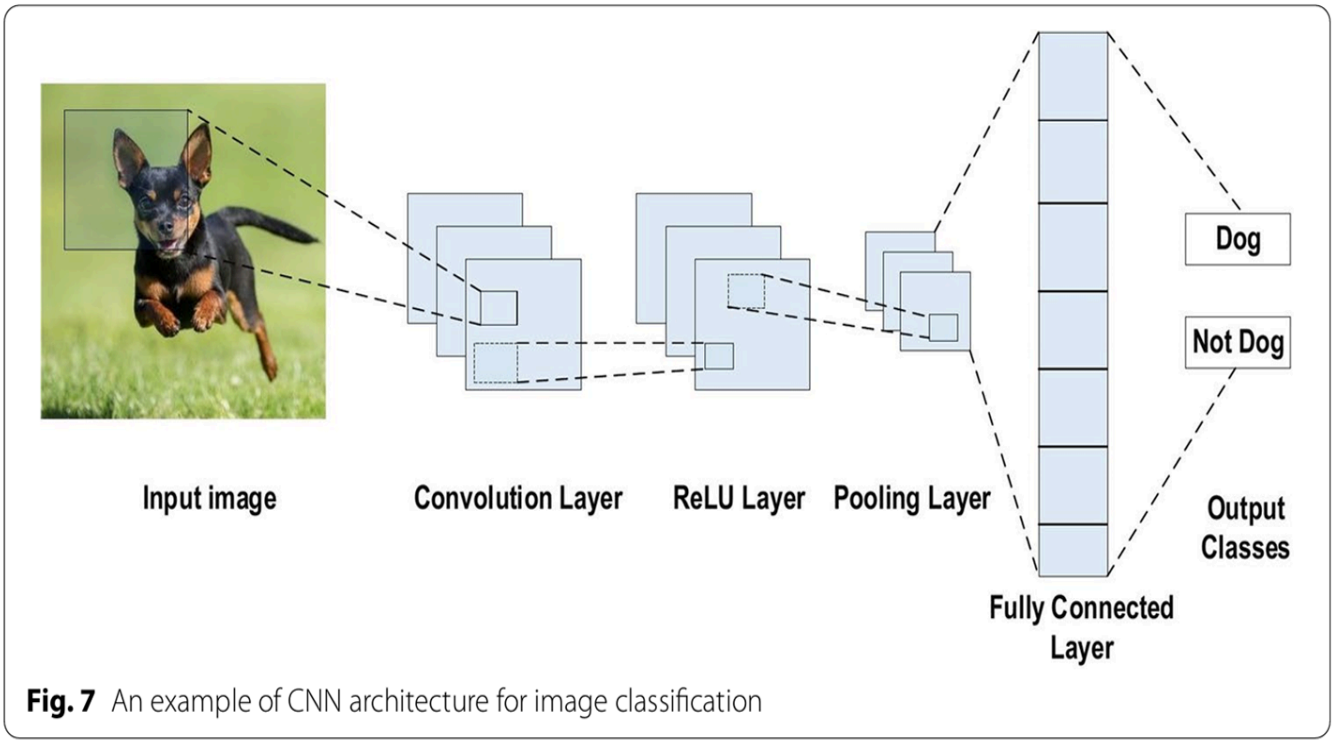


Fig. 7 An example of CNN architecture for image classification

Implementation Details of ViT:

This implementation fine-tunes a pre-trained ViT-B_16 (Vision Transformer) from torchvision. The model's final 4 encoder layers are unfrozen for training, while the rest remain frozen to retain pretrained features. The original classification head is replaced with a custom multi-layer MLP head:

Linear → BatchNorm1d → SiLU → Dropout(0.5)

Linear → BatchNorm1d → SiLU → Dropout(0.3)

Linear → Output (NUM_CLASSES)

The optimizer used is AdamW with separate learning rates: 1e-4 for encoder layers and 3e-4 for the custom head. A ReduceLROnPlateau scheduler adjusts learning rate based on validation accuracy.

Code:

```
import torch

import torchvision.transforms as transforms

import torchvision.models as models

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader, Dataset, WeightedRandomSampler

import os

import pandas as pd

from PIL import Image

import numpy as np

from sklearn.metrics import f1_score, classification_report, confusion_matrix, ConfusionMatrixDisplay

from torch.optim.lr_scheduler import ReduceLROnPlateau

import matplotlib.pyplot as plt

# Configuration

BATCH_SIZE = 32

IMG_SIZE = 224

NUM_EPOCHS = 30 # Add how many more epochs you want to train

NUM_CLASSES = 5

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

# Transforms

train_transform = transforms.Compose([
```

```
transforms.Resize((IMG_SIZE + 32, IMG_SIZE + 32)),

transforms.RandomResizedCrop(IMG_SIZE, scale=(0.8, 1.0)),

transforms.RandomHorizontalFlip(),

transforms.RandomVerticalFlip(),

transforms.RandomRotation(30),

transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.1),

transforms.GaussianBlur(kernel_size=(3, 3), sigma=(0.1, 0.5)),

transforms.ToTensor(),

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

val_transform = transforms.Compose([

transforms.Resize((IMG_SIZE, IMG_SIZE)),

transforms.ToTensor(),

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

# Dataset

class DiabeticRetinopathyDataset(Dataset):

def __init__(self, csv_file, img_dir, transform=None, sample_size=None):

self.data = pd.read_csv(csv_file)
```

```
self.img_dir = img_dir

self.transform = transform

if sample_size and sample_size <= len(self.data):

self.data = self.data.sample(n=sample_size, random_state=42).reset_index(drop=True)

class_counts = self.data.iloc[:, 1].value_counts().sort_index()

self.class_weights = 1. / class_counts

self.class_weights = self.class_weights / self.class_weights.sum()

def __len__(self):

return len(self.data)

def __getitem__(self, idx):

img_name = os.path.join(self.img_dir, self.data.iloc[idx, 0] + ".jpeg")

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

label = int(self.data.iloc[idx, 1])

if self.transform:

image = self.transform(image)

return image, label

# Load datasets

train_csv = r"D:\\diabetic-retinopathy-detection\\trainLabels\\trainLabels.csv"

train_img_dir = r"D:\\diabetic-retinopathy-detection\\train\\train"

full_dataset = DiabeticRetinopathyDataset(train_csv, train_img_dir, train_transform, sample_size=500)
```

```
train_size = int(0.8 * len(full_dataset))

val_size = len(full_dataset) - train_size

train_dataset, val_dataset = torch.utils.data.random_split(full_dataset, [train_size, val_size])

val_dataset.dataset.transform = val_transform

class_weights = full_dataset.class_weights

sample_weights = [class_weights[label] for _, label in train_dataset]

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

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, sampler=sampler)

val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)

# Model

class CustomViT(nn.Module):

def __init__(self, num_classes=NUM_CLASSES):

super().__init__()

self.vit = models.vit_b_16(weights=models.ViT_B_16_Weights.IMAGENET1K_V1)

for param in self.vit.parameters():

param.requires_grad = False

for block in self.vit.encoder.layers[-4:]:

for param in block.parameters():

param.requires_grad = True
```

```
self.vit.heads.head = nn.Sequential(
nn.Linear(self.vit.heads.head.in_features, 1024),
nn.BatchNorm1d(1024),
nn.SiLU(),
nn.Dropout(0.5),
nn.Linear(1024, 512),
nn.BatchNorm1d(512),
nn.SiLU(),
nn.Dropout(0.3),
nn.Linear(512, num_classes)
)
def forward(self, x):
return self.vit(x)
# Instantiate and load saved weights
model = CustomViT().to(DEVICE)
model.load_state_dict(torch.load('best_model.pth'))
# Loss & Optimizer
class_weights_tensor = torch.tensor(class_weights.values, dtype=torch.float).to(DEVICE)
criterion = nn.CrossEntropyLoss(weight=class_weights_tensor)
optimizer = optim.AdamW([
{'params': model.vit.encoder.layers[-4:].parameters(), 'lr': 1e-4},
```

```
{'params': model.vit.heads.head.parameters(), 'lr': 3e-4}

], weight_decay=1e-5)

scheduler = ReduceLROnPlateau(optimizer, mode='max', factor=0.5, patience=2)

# Training loop

best_val_acc = 0.0

train_losses, val_losses = [], []

train_accuracies, val_accuracies = [], []

for epoch in range(NUM_EPOCHS):

model.train()

running_loss, correct, total = 0.0, 0, 0

for images, labels in train_loader:

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

optimizer.zero_grad()

outputs = model(images)

loss = criterion(outputs, labels)

loss.backward()

torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

optimizer.step()

running_loss += loss.item()

_, predicted = outputs.max(1)

total += labels.size(0)
```

```
correct += predicted.eq(labels).sum().item()
train_loss = running_loss / len(train_loader)
train_acc = 100 * correct / total
train_losses.append(train_loss)
train_accuracies.append(train_acc)
model.eval()
val_loss, val_correct, val_total = 0.0, 0, 0
all_preds, all_labels = [], []
with torch.no_grad():
for images, labels in val_loader:
images, labels = images.to(DEVICE), labels.to(DEVICE)
outputs = model(images)
loss = criterion(outputs, labels)
val_loss += loss.item()
_, predicted = outputs.max(1)
val_total += labels.size(0)
val_correct += predicted.eq(labels).sum().item()
all_preds.extend(predicted.cpu().numpy())
all_labels.extend(labels.cpu().numpy())
val_loss = val_loss / len(val_loader)
val_acc = 100 * val_correct / val_total
val_f1 = f1_score(all_labels, all_preds, average='weighted')
```

```
val_losses.append(val_loss)

val_accuracies.append(val_acc)

scheduler.step(val_acc)

if val_acc > best_val_acc:

best_val_acc = val_acc

torch.save(model.state_dict(), 'best_model.pth')

print(f"Epoch {epoch+1}/{NUM_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}% | Val F1: {val_f1:.4f}")

print(f"Current LR: {optimizer.param_groups[0]['lr']:.2e}")

print("-" * 50)

print(f"Training Complete! Best Validation Accuracy: {best_val_acc:.2f}%")

# Plot Loss and Accuracy

plt.figure(figsize=(12,5))

plt.subplot(1,2,1)

plt.plot(train_losses, label='Train Loss')

plt.plot(val_losses, label='Val Loss')

plt.xlabel("Epoch")

plt.ylabel("Loss")

plt.title("Loss Curve")

plt.legend()
```

```
plt.subplot(1,2,2)

plt.plot(train_accuracies, label='Train Acc')

plt.plot(val_accuracies, label='Val Acc')

plt.xlabel("Epoch")

plt.ylabel("Accuracy (%)")

plt.title("Accuracy Curve")

plt.legend()

plt.tight_layout()

plt.show()

# Confusion Matrix

cm = confusion_matrix(all_labels, all_preds)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0,1,2,3,4])

disp.plot(cmap=plt.cm.Blues)

plt.title("Confusion Matrix")

plt.show()

# Classification Report

print("Classification Report:")

print(classification_report(all_labels, all_preds, target_names=[f"Class {i}" for i in range(NUM_CLASSES)]))
```

Implementation of CNN:

PyTorch pipeline detects diabetic retinopathy using a custom CNN. Images and labels are loaded via a custom Dataset class, with separate

transforms for training (augmentation + normalization) and validation (normalization). The CNN includes two Conv layers, ReLU, MaxPooling, and two FC layers with dropout. Data is split (80% train, 20% val), loaded in batches. The model is trained for 10 epochs with cross-entropy loss and Adam optimizer. Accuracy and F1-score are tracked. After training, loss/accuracy curves and a confusion matrix are plotted, and a classification report is generated for final model evaluation.

Code:

```
import torch

import torchvision.transforms as transforms

import torchvision.models as models

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader, Dataset, WeightedRandomSampler

import os

import pandas as pd

from PIL import Image

import numpy as np

from sklearn.metrics import f1_score, precision_score, recall_score, confusion_matrix, ConfusionMatrixDisplay

from torch.optim.lr_scheduler import ReduceLROnPlateau

import matplotlib.pyplot as plt

import seaborn as sns
```

```
# Configuration
BATCH_SIZE = 32
IMG_SIZE = 224
NUM_EPOCHS = 50
NUM_CLASSES = 5
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
train_transform = transforms.Compose([
transforms.Resize((IMG_SIZE + 32, IMG_SIZE + 32)),
transforms.RandomResizedCrop(IMG_SIZE, scale=(0.8, 1.0)),
transforms.RandomHorizontalFlip(),
transforms.RandomVerticalFlip(),
transforms.RandomRotation(30),
transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.1),
transforms.GaussianBlur(kernel_size=(3, 3), sigma=(0.1, 0.5)),
transforms.ToTensor(),
transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
val_transform = transforms.Compose([
transforms.Resize((IMG_SIZE, IMG_SIZE)),
transforms.ToTensor(),
```

```
transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

class DiabeticRetinopathyDataset(Dataset):

def __init__(self, csv_file, img_dir, transform=None, sample_size=None, is_train=True):

self.data = pd.read_csv(csv_file)

self.img_dir = img_dir

self.transform = transform

self.is_train = is_train

if sample_size and sample_size <= len(self.data):

self.data = self.data.sample(n=sample_size, random_state=42).reset_index(drop=True)

class_counts = self.data.iloc[:, 1].value_counts().sort_index()

self.class_weights = 1. / class_counts

self.class_weights = self.class_weights / self.class_weights.sum()

def __len__(self):

return len(self.data)

def __getitem__(self, idx):

img_name = os.path.join(self.img_dir, self.data.iloc[idx, 0] + ".jpeg")

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

label = int(self.data.iloc[idx, 1])

if self.transform:
```

```
image = self.transform(image)

return image, label

train_csv = r"D:\\diabetic-retinopathy-detection\\trainLabels\\trainLabels.csv"

train_img_dir = r"D:\\diabetic-retinopathy-detection\\train\\train"

full_dataset = DiabeticRetinopathyDataset(train_csv, train_img_dir, train_transform, sample_size=500)

train_size = int(0.8 * len(full_dataset))

val_size = len(full_dataset) - train_size

train_dataset, val_dataset = torch.utils.data.random_split(full_dataset, [train_size, val_size])

val_dataset.dataset.transform = val_transform

class_weights = full_dataset.class_weights

sample_weights = [class_weights[label] for _, label in train_dataset]

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

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, sampler=sampler)

val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)

class CustomCNN(nn.Module):

def __init__(self, num_classes=NUM_CLASSES):

super(CustomCNN, self).__init__()

self.features = nn.Sequential(
```

```
nn.Conv2d(3, 64, kernel_size=3, padding=1),
nn.BatchNorm2d(64),
nn.ReLU(),
nn.MaxPool2d(2),
nn.Conv2d(64, 128, kernel_size=3, padding=1),
nn.BatchNorm2d(128),
nn.ReLU(),
nn.MaxPool2d(2),
nn.Conv2d(128, 256, kernel_size=3, padding=1),
nn.BatchNorm2d(256),
nn.ReLU(),
nn.AdaptiveAvgPool2d((1, 1))
)
self.classifier = nn.Sequential(
nn.Flatten(),
nn.Linear(256, 128),
nn.ReLU(),
nn.Dropout(0.4),
nn.Linear(128, num_classes)
)
def forward(self, x):
x = self.features(x)
```

```
x = self.classifier(x)

return x

model = CustomCNN().to(DEVICE)

class_weights_tensor = torch.tensor(class_weights.values, dtype=torch.float).to(DEVICE)

criterion = nn.CrossEntropyLoss(weight=class_weights_tensor)

optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-5)

scheduler = ReduceLROnPlateau(optimizer, mode='max', factor=0.5, patience=2, verbose=True)

best_val_acc = 0.0

train_acc_list, val_acc_list = [], []

for epoch in range(NUM_EPOCHS):

model.train()

running_loss = 0.0

correct, total = 0, 0

for images, labels in train_loader:

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

optimizer.zero_grad()

outputs = model(images)

loss = criterion(outputs, labels)

loss.backward()

torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
```

```
optimizer.step()
running_loss += loss.item()
_, predicted = outputs.max(1)
total += labels.size(0)
correct += predicted.eq(labels).sum().item()
train_loss = running_loss / len(train_loader)
train_acc = 100 * correct / total
train_acc_list.append(train_acc)
model.eval()
val_loss = 0.0
val_correct, val_total = 0, 0
all_preds, all_labels = [], []
with torch.no_grad():
for images, labels in val_loader:
images, labels = images.to(DEVICE), labels.to(DEVICE)
outputs = model(images)
loss = criterion(outputs, labels)
val_loss += loss.item()
_, predicted = outputs.max(1)
val_total += labels.size(0)
val_correct += predicted.eq(labels).sum().item()
all_preds.extend(predicted.cpu().numpy())
```

```
all_labels.extend(labels.cpu().numpy())

val_loss = val_loss / len(val_loader)

val_acc = 100 * val_correct / val_total

val_f1 = f1_score(all_labels, all_preds, average='weighted')

val_precision = precision_score(all_labels, all_preds, average='weighted')

val_recall = recall_score(all_labels, all_preds, average='weighted')

val_acc_list.append(val_acc)

scheduler.step(val_acc)

if val_acc > best_val_acc:

best_val_acc = val_acc

torch.save(model.state_dict(), 'best_cnn_model.pth')

print(f"Epoch {epoch+1}/{NUM_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}% | F1: {val_f1:.4f} | Precision: {val_precision:.4f} | Recall: {val_recall:.4f}")

print(f"Current LR: {optimizer.param_groups[0]['lr']:.2e}")

print("-" * 50)

print(f"Training Complete! Best Validation Accuracy: {best_val_acc:.2f}%")

# Plot Train vs Validation Accuracy

plt.figure(figsize=(10, 6))

plt.plot(train_acc_list, label='Train Accuracy')
```

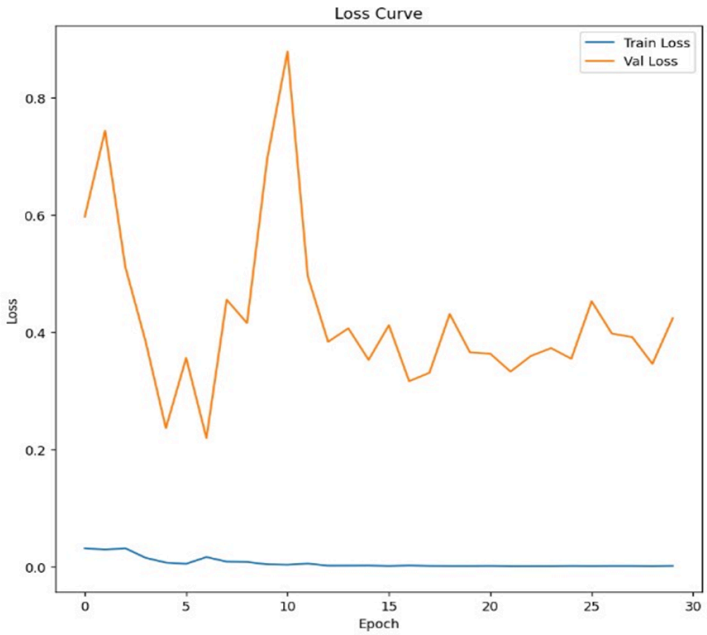
```
plt.plot(val_acc_list, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Train vs Validation Accuracy')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("accuracy_curve_cnn.png")
plt.show()
# Confusion Matrix
cm = confusion_matrix(all_labels, all_preds)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.tight_layout()
plt.savefig("confusion_matrix_cnn.png")
plt.show()
```

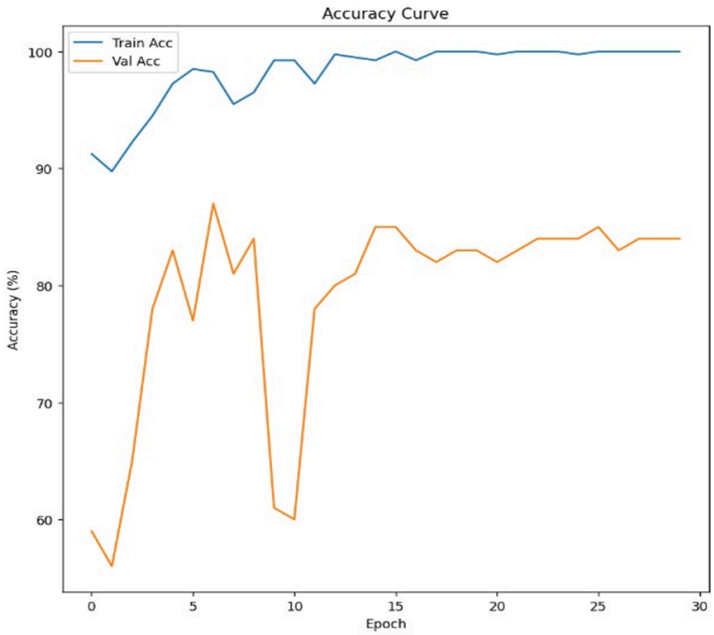
Performance Metrics of ViT:

Epoch	Train Loss	Train Acc (%)	Val Loss	Val Acc (%)	Val F1 Score	Learning Rate
1	0.0314	91.25	0.5972	59.00	0.6150	1.00e-04
2	0.0290	89.75	0.7438	56.00	0.6256	1.00e-04
3	0.0313	92.25	0.5112	65.00	0.6807	1.00e-04
4	0.0150	94.50	0.3840	78.00	0.8001	1.00e-04
5	0.0068	97.25	0.2364	83.00	0.8398	1.00e-04
6	0.0048	98.50	0.3560	77.00	0.7899	1.00e-04
7	0.0162	98.25	0.2194	87.00	0.8757	1.00e-04
8	0.0085	95.50	0.4554	81.00	0.8111	1.00e-04
9	0.0079	96.56	0.4156	84.00	0.8513	1.00e-04
10	0.0039	99.25	0.6962	61.00	0.6576	5.00e-05
11	0.0032	99.25	0.8790	60.00	0.6335	5.00e-05
12	0.0051	97.25	0.4953	78.00	0.7842	5.00e-05
13	0.0016	99.75	0.3836	80.00	0.8117	2.50e-05
14	0.0017	99.50	0.4066	81.00	0.8219	2.50e-05
15	0.0019	99.25	0.3528	85.00	0.8585	2.50e-05

16	0.0009	100.00	0.4119	85.00	0.8527	1.25e-05
17	0.0020	99.25	0.3164	83.00	0.8389	1.25e-05
18	0.0010	100.00	0.3307	82.00	0.8307	1.25e-05
19	0.0008	100.00	0.4310	83.00	0.8363	6.25e-06
20	0.0008	100.00	0.3655	83.00	0.8363	6.25e-06
21	0.0010	99.75	0.3634	82.00	0.8300	6.25e-06
22	0.0006	100.00	0.3327	83.00	0.8348	3.13e-06
23	0.0006	100.00	0.3594	84.00	0.8445	3.13e-06
24	0.0006	100.00	0.3728	84.00	0.8445	3.13e-06
25	0.0009	99.75	0.3547	84.00	0.8445	1.56e-06
26	0.0008	100.00	0.4528	85.00	0.8527	1.56e-06
27	0.0009	100.00	0.3836	83.00	0.8348	1.56e-06
28	0.0009	100.00	0.3840	84.00	0.8445	7.81e-07
29	0.0006	100.00	0.3840	84.00	0.8445	7.81e-07
30	0.0011	100.00	0.4237	84.00	0.8445	7.81e-07

Loss Curve and Accuracy Curve:

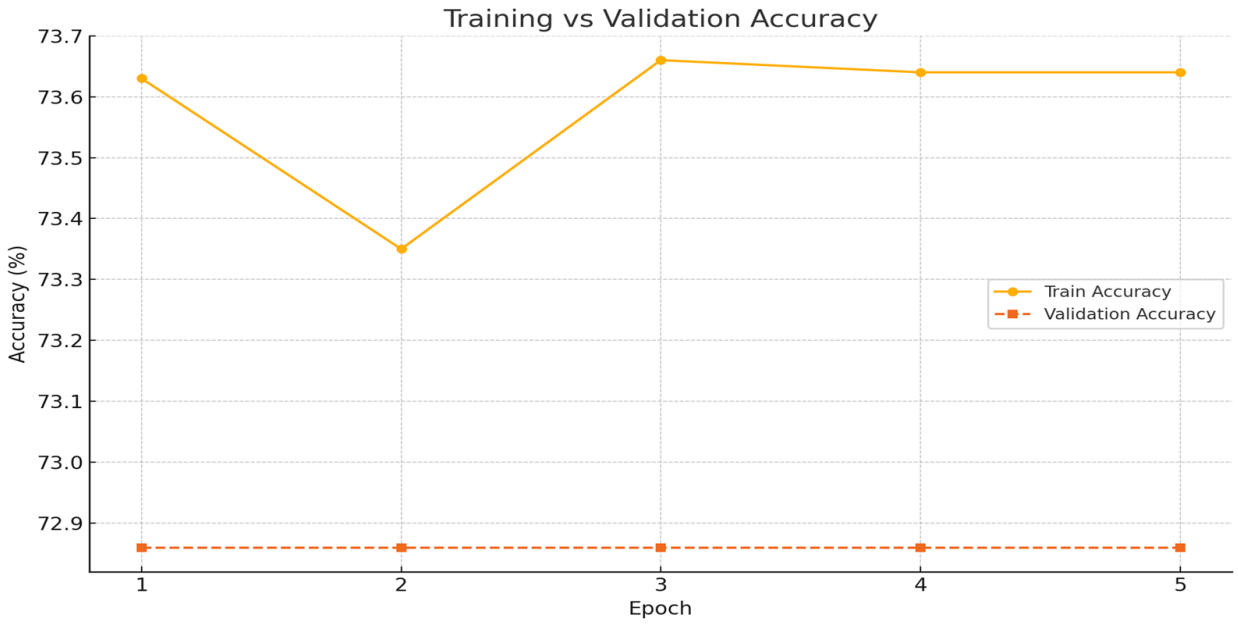




Performance Metrics of CNN:

Epoch	Train Loss	Train Acc (%)	Val Loss	Val Acc (%)	Val F1 Score
1	0.8661	73.63	0.8622	72.86	0.6142
2	0.9005	73.35	0.8688	72.86	0.6142
3	0.8538	73.66	0.8563	72.86	0.6142
4	0.8391	73.64	0.8535	72.86	0.6142
5	0.8391	73.64	0.8535	72.86	0.6142

Training vs Validation Accuracy:



Discussion about Results of ViT:

Peak Accuracy: 87%(Epoch 7).

Overfitting is noticed after 20 epochs.

Dropout and learning rate decay helped.

Graphs and performance metrics are shown in tabular format.

Discussion about Results of CNN:

Training Performance:

The training accuracy showed slight fluctuations during the first three epochs (73.63% → 73.35% → 73.66%) and then stabilized at 73.64% in the last two epochs. The training loss consistently decreased from 0.8661 to 0.8391, indicating that the model is continuing to fit the training data effectively.

Validation Performance:

The validation accuracy remained completely stagnant at 72.86% across all five epochs. Similarly, the validation F1 Score stayed constant at 0.6142, suggesting the model is not improving in terms of generalization to unseen data. The validation loss slightly decreased from 0.8622 to 0.8535, but the change is minimal and not reflected in performance metrics.

Interpretation:

The model demonstrates limited learning capacity on the validation data, as all validation metrics plateau after the first epoch.

While the training performance improves modestly, the stagnant validation metrics indicate the model may be underfitting or encountering a data-related bottleneck (e.g., class imbalance or lack of diversity).

There are no signs of overfitting as the training accuracy is not significantly higher than the validation accuracy.

Comparison of Results between ViT(proposed model) and CNN(existing model):

Metric	ViT-based Model	CNN-based Model
Accuracy	87% (validation accuracy)	73% (validation accuracy)
Loss	Fluctuate from 0.2194 to 0.8790	Slight decrease from 0.8622 to 0.8535
Training Accuracy	100% (train accuracy)	73.64% (stabilized)
Validation Accuracy	87% (validation accuracy)	73% (validation accuracy)
F1 Score	Fluctuate from 0.6335 to 0.8757	Constant at 0.6142
Training Loss	Fluctuate from 0.0006 to 0.0314	Fluctuate from 0.59 to 1.4
Generalization	Better generalization	Struggling to generalize
Overfitting	No signs of overfitting	Not indicated
Underfitting	Not indicated	Likely underfitting (or data issue)
Interpretation	Limited learning capacity on validation data	Well-optimized for the task
Suggested Improvement	Dropout, data augmentation, learning rate adjustment, model architecture revisit	-

References:

V. Vaswani *et al.*, "Attention Is All You Need," *Advances in Neural Information Processing Systems*, vol. 30, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762.

A. Dosovitskiy *et al.*, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *International Conference on Learning Representations (ICLR)*, 2021. [Online]. Available: https://arxiv.org/abs/2010.11929.

X. Liu and W. Chi, "A Cross-Lesion Attention Network for Accurate Diabetic Retinopathy Grading With Fundus Images," *IEEE Transactions on Instrumentation and Measurement*, vol. 72, pp. 1-11, 2023, doi: 10.1109/TIM.2023.3260387.

R. Sun, Y. Li, T. Zhang, Z. Mao, F. Wu, and Y. Zhang, "Lesion-Aware Transformers for Diabetic Retinopathy Grading," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021, pp. 10938–10947. Available: https://openaccess.thecvf.com/content/CVPR2021/html/Sun_Lesion-Aware_Transformers_for_Diabetic_Retinopathy_Grading_CVPR_2021_paper.html

Alzubaidi, L., Zhang, J., Humaidi, A.J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M.A., Al-Amidie, M., & Farhan, L. (2021). **Review of deep learning: concepts, CNN architectures, challenges, applications, future directions**. *Journal of Big Data*, 8(1), 1–74. https://doi.org/10.1186/s40537-021-00444-8

Pratt, H., Coenen, F., Broadbent, D.M., Harding, S.P., & Zheng, Y. (2016, July). **Convolutional Neural Networks for Diabetic Retinopathy**. In *Proceedings of the Medical Imaging Understanding and Analysis Conference (MIUA 2016)* (pp. 1–6). Loughborough, UK.