National College of Ireland

Configuration Manual

MSc Research Project
Programme Name

Qasim Ali
Student ID: x23412241

School of Computing
National College of Ireland

Supervisor: Arundev Vamadevan

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Qasim Ali
Student ID:	X23412241
Programme:	MSc Artificial Intelligence
Year:	2024-25
Module:	MSc Research Project
Supervisor:	Arundev Vamadevan
Submission Due Date:	11/08/2025
Project Title:	Configuration Manual
Word Count:	3189
Page Count:	19

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Qasim Ali
Date:	11th August 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Qasim Ali

X23412241

1 Introduction

This configuration manual serves as a comprehensive guide for setting up and implementing my research project focused on automated diabetic retinopathy detection. The manual has been designed to be accessible to researchers and practitioners who wish to replicate this work or build upon the foundations established.

Diabetic retinopathy (DR) is a major cause of global health burden especially in regions with high diabetes prevalence. At the study site and in many other developing countries like Pakistan, a substantial gap exists between diabetic patients needing regular eye check-ups and ophthalmologists. This discrepancy often leads to diagnoses that are too late in the progress of the disease, when fewer and less effective treatment options remain.

This project is inspired by my perceived reality of preventable blindness in my own community. Traditional screening is a time-consuming process involving visual check up by professionals, hence leading to healthcare bottlenecks. A non-human-intelligence-based automated screening system could well offer the opportunity to democratise access to early detection services.

The software uses deep learning approaches to automatically analyse retinal fundus images for indication of the earliest signs of diabetic retinopathy applicable and defined under US regulations such as microaneurysms, haemorrhages, and exudates. This approach unites classical convolutional neural networks (Farhadi et al.,2019) with modern vision transformer architectures to deliver strong performance over a wide swath of image inputs.

The best part of this solution is its real-world use case Once set up, the system can work in resource poor environment with a basic computer system and internet connectivity. Retinal photography with standard fundus cameras will be done by a trained local health worker, the AI will provide instant evaluation of these images, highlighting cases at high risk for referral to Eye Care Specialist.

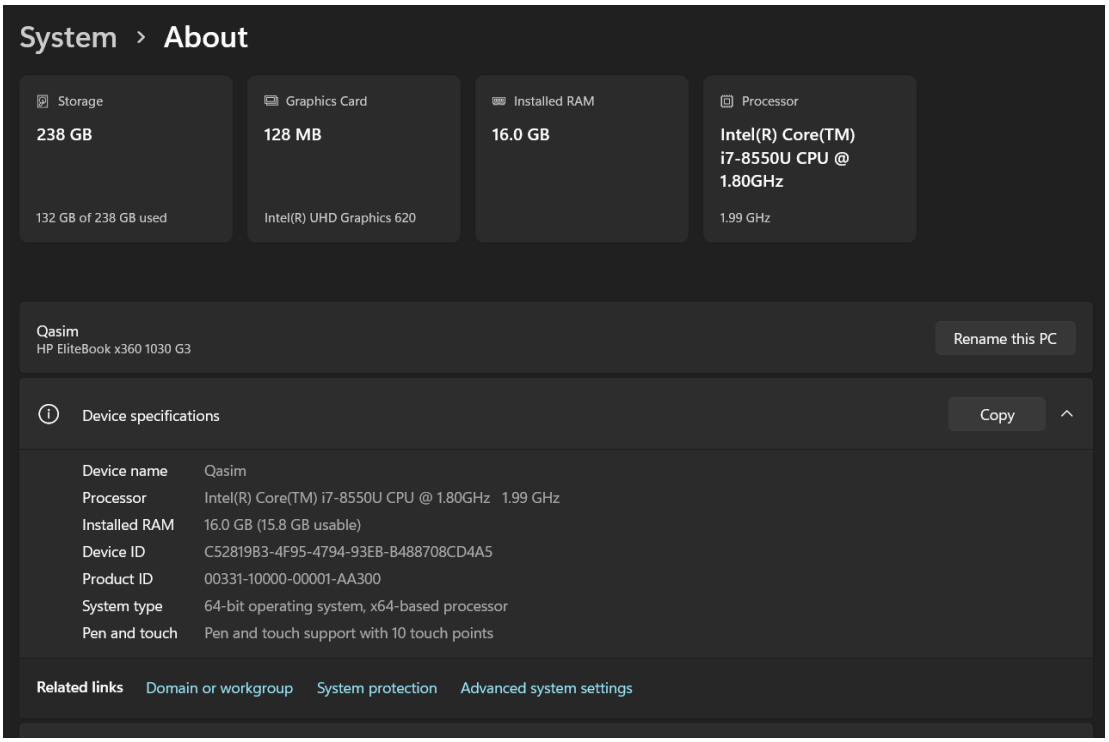
Required for this manual is minor confidence with the concepts of machine learning, but will give links to resources in case your technical knowledge boundaries are challenged. Every section goes step-by-step, so the whole process from setting up everything to deploy the model is covered.

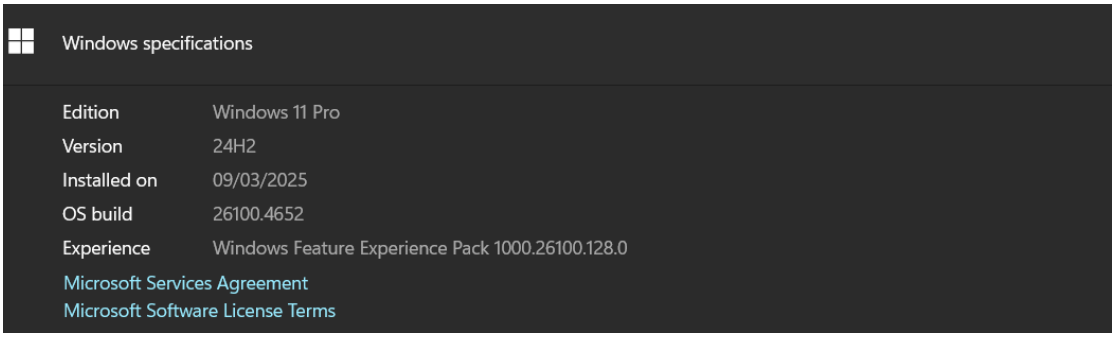
2 System requirements and specifications

2.1 Hardware Configuration

This project's development environment combines local computational resources with cloud-

based platforms to optimise availability and performance. The hybrid L2-ML framework also enables relatively small research groups that do not have substantial local resources to contribute meaningfully to state-of-the-art machine learning research.





2.2 Software Dependencies:

The software stack emphasises open-source solutions to maintain cost-effectiveness and reproducibility. All primary dependencies are freely available and well-supported by active communities.

Primary Development Environment:

- Google Colab: Browser-based Colab notebook environment
- Google Drive: Cloud storage for datasets and model persistence
- Python 3.8+: Core programming language with extensive machine learning libraries

Data Management Tools:

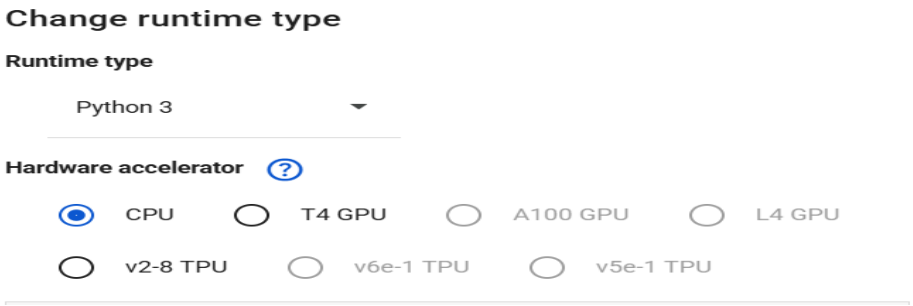
- Microsoft Excel: Initial data exploration and organisation
- Pandas: Structured data manipulation and analysis
- NumPy: Numerical computing foundation

Machine Learning Frameworks:

- PyTorch: Primary deep learning framework
- Transformers: State-of-the-art transformer model implementations
- Segmentation Models PyTorch: Pre-built segmentation architectures

The selection of these tools reflects a balance between functionality, community support, and learning curve considerations. Each component has been chosen for its reliability and extensive documentation.

Figure 2(A): Google Colab Interface



3 Dataset Preparation and Management

3.1 DIARETDB1 Dataset Overview

The foundation of this research rests upon the DIARETDB1 dataset, a carefully curated collection of retinal images specifically designed for diabetic retinopathy research. This dataset has gained widespread acceptance in the medical image analysis community due to its rigorous annotation standards and clinical relevance.

Dataset Characteristics:

- Total Images: 113 colour fundus photographs
- Image Resolution: Originally varying, standardised to 512×512 pixels
- Annotation Quality: Expert-validated ground truth masks
- Clinical Relevance: Representative of real-world screening scenarios

The images capture the posterior pole of the retina, including the optic disc, macula, and major vascular arcades. Each photograph undergoes professional medical annotation, with lesions precisely delineated by qualified ophthalmologists. This gold-standard labelling ensures that our models learn from clinically accurate examples.

Data Partitioning Strategy:

- Training Set: 70% (779 images) - Model parameter learning
- Validation Set: 15% (167 images) - Hyperparameter tuning and early stopping
- Test Set: 15% (167 images) - Final performance evaluation

This partitioning follows established machine learning best practices, ensuring sufficient data for model training whilst maintaining independent evaluation sets. The stratified sampling

approach maintains consistent representation of different lesion types across all subsets.

3.2 Data Acquisition and Preprocessing

The dataset is publicly available through Kaggle, supporting reproducible research and enabling comparison with published benchmarks. The acquisition process involves straightforward download and local organisation procedures.

Data Access:

URL: https://www.kaggle.com/datasets/standard-diabetic-retinopathy

License: Creative Commons (research use permitted)

Download Size: Approximately 2.3GB

Preprocessing Pipeline: The preprocessing stage standardises image characteristics and applies data augmentation techniques to improve model generalisation. This process addresses common challenges in medical imaging, including varying illumination conditions and image quality differences.

Standard preprocessing steps include:

1. Image Resizing: Uniform 512×512 pixel dimensions
2. Normalisation: Pixel value standardisation using ImageNet statistics
3. Quality Assessment: Automated filtering of poor-quality images
4. Augmentation: Geometric and photometric transformations during training

The preprocessing choices reflect clinical realities, where fundus images may vary in quality due to patient movement, operator skill, or equipment differences. By training on augmented data, our models become more robust to these real-world variations.

Figure 3(a): Representative Fundus Image

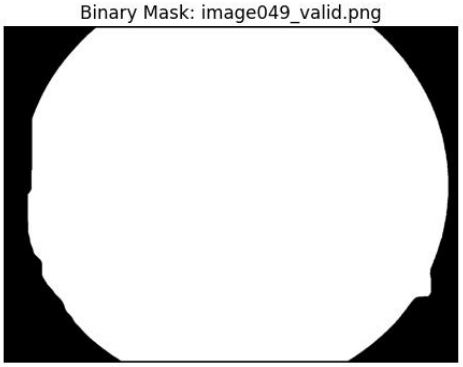
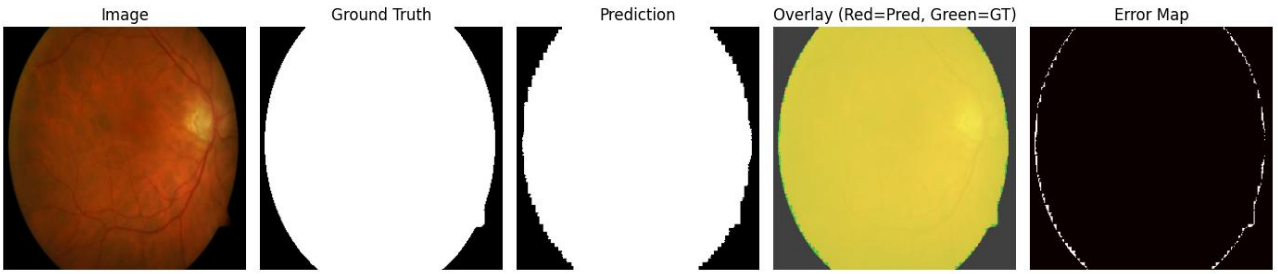


Figure 3(b) Corresponding Ground Truth Mask:



4. Tools and Libraries

Several tools and libraries were utilized to implement the project. This section lists the development tools/platforms and the major libraries along with their purposes:

Google Colab: Cloud-based Colab notebook environment used for running the experiments. Colab offers an easily configurable runtime with necessary libraries and allows mounting Google Drive for data access. It was used for writing and executing Python code for training the models and visualizing results.

Python 3.10 & Colab Notebooks: The programming language and interactive notebook interface in which all code was written. Python 3 (specifically the Colab default Python 3.10 environment) was used for its rich ecosystem of deep learning libraries. Colab notebooks allowed step-by-step execution and inline visualization of outputs (plots of training loss, sample segmentations, etc.).

Google Drive: Cloud storage used in conjunction with Colab. The DIARETDB1 dataset, pre-trained model weights, and outputs (like saved model checkpoints and result images) were stored on Google Drive. Colab's integration with Drive allowed seamless reading of input data and writing of results. For example, the dataset was placed in MyDrive/Colab Notebooks/diaretdb1/ directory, and Colab notebooks were also saved in Drive.

Libraries and Frameworks: The project relied on a range of Python libraries, primarily for deep learning and data processing:

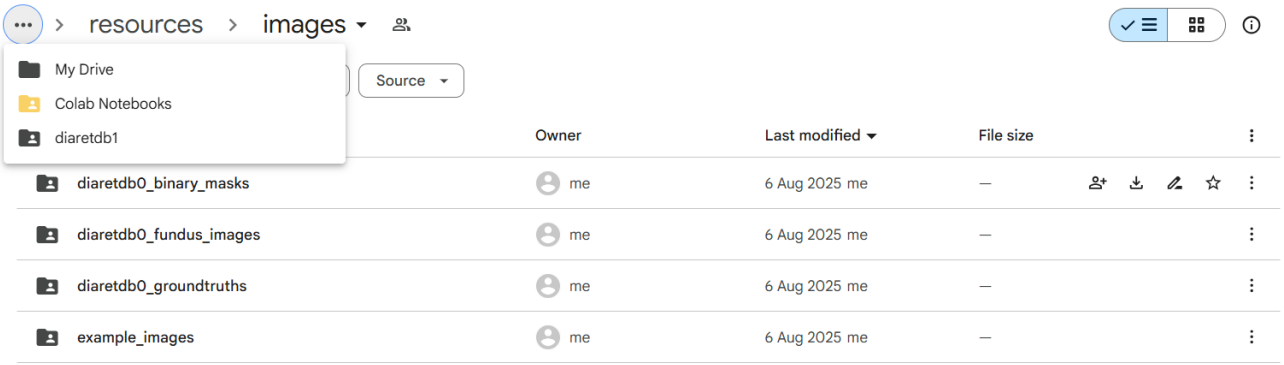
- PyTorch (torch)
- Torchvision
- Segmentation-Models-PyTorch (smp)
- Hugging Face Transformer
- Albumentations
- OpenCV (cv2) and PIL (Python Imaging Library
- NumPy
- Matplotlib
- scikit-learn: The ConfusionMatrixDisplay from scikit-learn was used to visualize the confusion matrix of predicted vs. true mask pixels.

All the above libraries are open-source and were installed either via Colab's pre-installed

packages or through pip in the Colab notebook. Notably, segmentation-models-pytorch, albumentations, and transformers were installed in Colab at runtime (using !pip install commands) since they might not be available by default. The Colab platform comes with many libraries pre-installed (including PyTorch and Matplotlib), but for completeness, instructions to install any missing packages are included in the notebooks. Ensuring all these libraries are available and at compatible versions is essential for replicating the environment.

5. Dataset Setup

The project utilizes the DIARETDB1 dataset, which is a publicly available collection of retinal fundus images for diabetic retinopathy research. In our context, we focus on lesion segmentation identifying areas of the retina with lesions (such as microaneurysms, hemorrhages, or exudates) that are early signs of diabetic retinopathy. The dataset setup involves organizing the images and masks, and applying preprocessing steps before feeding them into the models. Dataset Contents: DIARETDB1 consists of retinal photographs and their corresponding ground-truth lesion annotations. The images are high-resolution color fundus images (each roughly 1500×1152 pixels), and for each image there is a binary mask highlighting the lesion regions (the pixels belonging to any type of lesion are marked). Folder Structure: All dataset files were stored on Google Drive for access in Colab. The data is organized into two main folders – one for the input images and one for the ground truth masks. For instance, the structure in Google Drive might look like:



Each fundus image in the images/ directory has a corresponding binary mask file in the diaretdb0_groundtruths directory. The filenames are aligned such that sorting both directories alphabetically will match images to their masks (e.g., image_001.png aligns with image_001_mask.png).

```
Sample image files: ['image001.png', 'image002.png', 'image003.png']
Sample mask files: ['image001_valid.png', 'image002_valid.png', 'image003_valid.png']
```

In our code, we relied on sorting the file lists to pair images and masks. It's important to ensure the names match exactly (before the suffix) so that each retina image gets the correct mask during training. If the dataset uses a different naming convention, one should rename the mask files accordingly or adjust the code that matches them. Preprocessing Steps: Retinal images come in varying sizes and sometimes with different lighting conditions. We applied several preprocessing steps:

Resizing:

```
# Transformations
transform = T.Compose([
    T.Resize((512, 512)),
    T.ToTensor(),
])

# Dataset and loader
dataset = FundusSegmentationDataset(image_dir, mask_dir, transform=transform)
dataloader = DataLoader(dataset, batch_size=4, shuffle=True)

# Check sample
images, masks = next(iter(dataloader))
print("Image batch shape:", images.shape)
print("Mask batch shape:", masks.shape)
```

Normalization:

```
# --- Predict using SimpleSegNet ---
print("\nPredicting with SimpleSegNet...")
# Use the transform defined for SimpleSegNet
simple_segnet_transform = T.Compose([
    T.Resize((256, 256)),
    T.ToTensor(),
])

# --- Predict using UNet (assuming 'model' currently holds the trained UNet) ---
print("\nPredicting with UNet...")
# Use the transform defined for UNet (Albumentations)
unet_transform_predict = A.Compose([
    A.Resize(256, 256),
    A.Normalize(),
    ToTensorV2()
])


predicted_mask_unet = predict_single_image(model=model,
                                           image_path=test_image_path,
                                           transform=unet_transform_predict,
                                           device=device,
                                           model_type='UNet')
```

Data Augmentation:

```
class RetinaSegmentationDataset(Dataset):
    def __init__(self, image_dir, mask_dir, transform=None):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.transform = transform
        self.images = sorted([f for f in os.listdir(image_dir) if f.endswith('.png')])
        self.masks = sorted([f for f in os.listdir(mask_dir) if f.endswith('.png')])

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

    def __getitem__(self, idx):
        img_path = os.path.join(self.image_dir, self.images[idx])
        mask_path = os.path.join(self.mask_dir, self.masks[idx])

        image = cv2.imread(img_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
        mask = (mask > 0).astype('float32')

        if self.transform:
            augmented = self.transform(image=image, mask=mask)
            image = augmented['image']
            mask = augmented['mask'].unsqueeze(0)

        return image, mask
```

Dataset Split:

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

# Model, criterion, optimizer, scheduler
model = SimpleSegNet().to(device)
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.1)

# Split into training and validation sets
val_size = int(0.2 * len(dataset))
train_size = len(dataset) - val_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=4, shuffle=False)

# For plotting
train_losses = []
val_losses = []
```

After these steps, the data is ready to be loaded by the DataLoader. The custom dataset classes (implemented in code as **FundusSegmentationDataset**, **RetinaSegmentationDataset**, or **SegFormerDataset** for different contexts) handle loading an image-mask pair, applying the transformations, and returning tensor data. Memory considerations: A 512×512 image is not very large, so we could load multiple images per batch (batch size was typically 4) without running out of memory in Colab. If using a GPU, this image size is also comfortable for models like UNet and SegFormer B0. In summary, to use the DIARETDB1 dataset for this project, place the files in the described folder structure, apply resizing to 512×512, normalize the image pixel values, and split the data for training/validation. With this setup, we maintain consistency across all model experiments and ensure that differences in results come from the model architectures rather than data handling differences.

6. Model Configuration and Execution

Multiple deep learning models were configured and tested for the lesion segmentation task, ranging from a basic convolutional network to advanced transformer-based architectures. This section details the architecture and execution of each model, including hyperparameters and training procedures. All models output a segmentation mask given an input retinal image, and they were trained to minimize segmentation error (comparing predicted mask to ground truth). Common settings across models included a training duration of around 20 epochs, an 80/20 train-validation split, and similar batch sizes (mostly 4) to allow fair comparison. Below we describe each model in detail:

6.1 SimpleSegNet – Custom CNN from Scratch

Architecture: SimpleSegNet is a lightweight convolutional neural network designed from scratch for segmentation. It follows an encoder-decoder structure:

```python
class SimpleSegNet(nn.Module):
    def __init__(self):
        super(SimpleSegNet, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(32, 16, 2, stride=2), nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 2, stride=2), nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

This SimpleSegNet essentially is a tiny U-Net-like structure without skip connections – two levels down and two levels up. It's kept deliberately simple to act as a baseline. Configuration & Hyperparameters: The model was implemented in PyTorch as a class SimpleSegNet(nn.Module). Key configurations:

Loss Function & Optimizer:

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

# Model, criterion, optimizer, scheduler
model = SimpleSegNet().to(device)
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.1)

# Split into training and validation sets
val_size = int(0.2 * len(dataset))
train_size = len(dataset) - val_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=4, shuffle=False)

# For plotting
train_losses = []
val_losses = []
```

Learning Rate Scheduler

```
    # Training loop
    epochs = 20
    for epoch in range(epochs):
        start_time = time.time()
        model.train()
        train_loss = 0.0

        for imgs, masks in train_loader:
            imgs, masks = imgs.to(device), masks.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, masks)

            optimizer.zero_grad()
            loss.backward()
            utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()

            train_loss += loss.item()

        avg_train_loss = train_loss / len(train_loader)
        train_losses.append(avg_train_loss)
```

Batch Size: 4 images per batch were fed during training. This was found to be a reasonable batch size given Colab's memory constraints and the small model size.

Device: The code automatically uses GPU if available (device = 'cuda' if available else 'cpu'), but in our Colab runs it was typically the CPU. The model is small enough to train on CPU relatively quickly (each epoch taking only a few seconds).

Training Process:

```
# Dataset and loader
dataset = FundusSegmentationDataset(image_dir, mask_dir, transform=transform)
dataloader = DataLoader(dataset, batch_size=4, shuffle=True)
```

Checkpoint Saving:

```
    # Validation
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for imgs, masks in val_loader:
            imgs, masks = imgs.to(device), masks.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, masks)
            val_loss += loss.item()

    avg_val_loss = val_loss / len(val_loader)
    val_losses.append(avg_val_loss)

    scheduler.step()
    elapsed = time.time() - start_time

    print(f"Epoch [{epoch+1}/{epochs}] | Train Loss: {avg_train_loss:.4f} | Val Loss: {avg_val_loss:.4f} | Time: {elapsed:.2f}s")

    # Save checkpoint
    if (epoch + 1) % 10 == 0:
        torch.save(model.state_dict(), f"checkpoint_epoch_{epoch+1}.pt")
```

Execution Instructions: Running this model required first initializing the dataset and DataLoader, then instantiating SimpleSegNet and the optimizer/loss as described. All of these steps are encapsulated in the provided notebook. To reproduce:

Mount Drive and Dataset:

```
import os

# Root folder of dataset in Google Drive
dataset_root = '/content/drive/MyDrive/diaretdb1'

# Subfolders
image_dir = os.path.join(dataset_root, 'images')
mask_dir = os.path.join(dataset_root, 'groundtruth')
```

```
import os

# List folders inside "resources/images/"
image_dir = '/content/drive/MyDrive/Colab Notebooks/diaretdb1/resources/images/diaretdb0_fundus_images'
mask_dir = '/content/drive/MyDrive/Colab Notebooks/diaretdb1/resources/images/diaretdb0_binary_masks'
```

Initialize Dataset:

```
# Dataset and loader
dataset = FundusSegmentationDataset(image_dir, mask_dir, transform=transform)
dataloader = DataLoader(dataset, batch_size=4, shuffle=True)
```

Run Training Loop:

```
# Training loop
epochs = 20
for epoch in range(epochs):
    start_time = time.time()
    model.train()
    train_loss = 0.0

    for imgs, masks in train_loader:
        imgs, masks = imgs.to(device), masks.to(device)
        outputs = model(imgs)
        loss = criterion(outputs, masks)

        optimizer.zero_grad()
        loss.backward()
        utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        train_loss += loss.item()

    avg_train_loss = train_loss / len(train_loader)
    train_losses.append(avg_train_loss)

    # Validation
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for imgs, masks in val_loader:
            imgs, masks = imgs.to(device), masks.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, masks)
```

Loss Curve: After training, a plot of training vs validation loss is generated to verify convergence (generally, we expect the losses to decrease and stabilize, which they did).

Model Checkpoint:

```
    val_loss = 0.0
    with torch.no_grad():
        for imgs, masks in val_loader:
            imgs, masks = imgs.to(device), masks.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, masks)
            val_loss += loss.item()

    avg_val_loss = val_loss / len(val_loader)
    val_losses.append(avg_val_loss)

    scheduler.step()
    elapsed = time.time() - start_time

    print(f"Epoch [{epoch+1}/{epochs}] | Train Loss: {avg_train_loss:.4f} | Val Loss: {avg_val_loss:.4f} | Time: {elapsed:.2f}s")

    # Save checkpoint
    if (epoch + 1) % 10 == 0:
        torch.save(model.state_dict(), f"checkpoint_epoch_{epoch+1}.pt")

# Plotting training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss Curve')
plt.legend()
plt.grid(True)
plt.show()
```

6.2 Vision Transformer (ViT) from Scratch for Segmentation

Architecture: The Vision Transformer model in this project was implemented from scratch to perform segmentation. This model is inspired by the ViT architecture originally designed for image classification, but here we adapt it to output a segmentation map. The architecture can be summarized as:

Patch Embedding:

```
class PatchEmbedding(nn.Module):
    def __init__(self, img_size=256, patch_size=16, in_channels=3, emb_dim=256):
        super().__init__()
        self.n_patches = (img_size // patch_size) ** 2
        self.patch_embed = nn.Conv2d(in_channels, emb_dim, kernel_size=patch_size, stride=patch_size)
        self.pos_embed = nn.Parameter(torch.randn(1, self.n_patches, emb_dim))

    def forward(self, x):
        x = self.patch_embed(x)  # [B, emb_dim, H', W']
        x = x.flatten(2).transpose(1, 2)  # [B, N_patches, emb_dim]
        x = x + self.pos_embed
        return x
```

Transformer Encoder:

```
class TransformerBlock(nn.Module):
    def __init__(self, emb_dim=256, heads=4, ff_hidden=512, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(emb_dim)
        self.attn = nn.MultiheadAttention(embed_dim=emb_dim, num_heads=heads, dropout=dropout, batch_first=True)
        self.norm2 = nn.LayerNorm(emb_dim)
        self.ff = nn.Sequential(
            nn.Linear(emb_dim, ff_hidden),
            nn.ReLU(),
            nn.Linear(ff_hidden, emb_dim)
        )

    def forward(self, x):
        x2 = self.norm1(x)
        attn_out, _ = self.attn(x2, x2, x2)
        x = x + attn_out
        x2 = self.norm2(x)
        x = x + self.ff(x2)
        return x
```

Decoder for Segmentation:

```python
class ViT_Segmentation(nn.Module):
    def __init__(self, img_size=256, patch_size=16, in_channels=3, emb_dim=256, num_layers=4, num_classes=1):
        super().__init__()
        self.patch_embed = PatchEmbedding(img_size, patch_size, in_channels, emb_dim)
        self.transformer = nn.Sequential(
            *[TransformerBlock(emb_dim) for _ in range(num_layers)]
        )
        self.decoder = nn.Sequential(
        nn.ConvTranspose2d(emb_dim, 256, kernel_size=2, stride=2),  # 16 → 32
        nn.ReLU(),
        nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2),      # 32 → 64
        nn.ReLU(),
        nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2),       # 64 → 128
        nn.ReLU(),
        nn.ConvTranspose2d(64, num_classes, kernel_size=2, stride=2), # 128 → 256
        nn.Sigmoid()
    )

        self.img_size = img_size
        self.patch_size = patch_size
        self.emb_dim = emb_dim

    def forward(self, x):
        B = x.size(0)
        x = self.patch_embed(x)              # [B, N_patches, emb_dim]
        x = self.transformer(x)              # [B, N_patches, emb_dim]
        h = w = self.img_size // self.patch_size
        x = x.transpose(1, 2).reshape(B, self.emb_dim, h, w)  # [B, emb_dim, H', W']
        x = self.decoder(x)                  # [B, 1, H, W]
        return x
```

This scratch ViT model essentially treats segmentation as a patch classification problem and then reconstructs the image. It doesn't incorporate skip connections like UNet; it relies on the transformer's global context to propagate information. Configuration & Hyperparameters:

Input Resolution: We fed 256×256 images to this model (unlike 512×512 for others) to reduce computational load on the transformer part, given we were not using a high-memory GPU. This is an implementation detail: one could train it on 512×512, but that would result in 1024 patches which is more memory-intensive.

Loss Function & Optimizer:

```python
# Model, criterion, optimizer, scheduler
model = ViT_Segmentation().to(device)
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.1)
```

Training Procedure: We trained for 20 epochs. The procedure is analogous to SimpleSegNet iterate over batches, compute output and loss, backpropagate, update weights. We monitored training and validation loss.

```python
# Training loop
epochs = 20
for epoch in range(epochs):
    start_time = time.time()
    model.train()
    train_loss = 0.0

    for imgs, masks in train_loader:
        imgs, masks = imgs.to(device), masks.to(device)
        outputs = model(imgs)
        loss = criterion(outputs, masks)

        optimizer.zero_grad()
        loss.backward()
        utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        train_loss += loss.item()

    avg_train_loss = train_loss / len(train_loader)
    train_losses.append(avg_train_loss)

    # Validation
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for imgs, masks in val_loader:
            imgs, masks = imgs.to(device), masks.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, masks)
```

Overfitting Precaution:

```python
def __init__(self, emb_dim=256, heads=4, ff_hidden=512, dropout=0.1):
    super().__init__()
    self.norm1 = nn.LayerNorm(emb_dim)
    self.attn = nn.MultiheadAttention(embed_dim=emb_dim, num_heads=heads, dropout=dropout, batch_first=True)
    self.norm2 = nn.LayerNorm(emb_dim)
    self.ff = nn.Sequential(
        nn.Linear(emb_dim, ff_hidden),
        nn.ReLU(),
        nn.Linear(ff_hidden, emb_dim)
    )
```

Checkpointing:

```python
# Validation
model.eval()
val_loss = 0.0
with torch.no_grad():
    for imgs, masks in val_loader:
        imgs, masks = imgs.to(device), masks.to(device)
        outputs = model(imgs)
        loss = criterion(outputs, masks)
        val_loss += loss.item()

avg_val_loss = val_loss / len(val_loader)
val_losses.append(avg_val_loss)

scheduler.step()
elapsed = time.time() - start_time

print(f"Epoch [{epoch+1}/{epochs}] | Train Loss: {avg_train_loss:.4f} | Val Loss: {avg_val_loss:.4f} | Time: {elapsed:.2f}s")

# Save checkpoint
if (epoch + 1) % 10 == 0:
    torch.save(model.state_dict(), f"checkpoint_epoch_{epoch+1}.pt")
```

Evaluation: After training, use the model in evaluation mode to predict on validation images. As with other models, apply a threshold of 0.5 on the Sigmoid output to get binary masks for computing metrics.

```python
def evaluate_vit_scratch(model, dataloader, device):
    model.eval()
    all_preds = []
    all_targets = []

    with torch.no_grad():
        for images, masks in dataloader:
            images = images.to(device)
            masks = masks.to(device)

            outputs = model(images)
            preds = (outputs > 0.5).float()

            all_preds.append(preds.cpu().numpy().astype(np.uint8))
            all_targets.append(masks.cpu().numpy().astype(np.uint8))

    y_pred = np.concatenate(all_preds).reshape(-1)
    y_true = np.concatenate(all_targets).reshape(-1)

    # Compute metrics
    acc = accuracy_score(y_true, y_pred)
    prec = precision_score(y_true, y_pred)
    rec = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    iou = jaccard_score(y_true, y_pred)
```

The custom ViT model demonstrates how a transformer-based approach without convolutional inductive biases can segment images. It achieved good performance, though as we will see, it was outperformed by the more specialized SegFormer and UNet (likely due to those benefiting from either pretraining or architectural features like skip connections). The ViT approach is educational and shows the potential of transformer encoders for pixel-level prediction tasks.

6.3 SegFormer - Pretrained Transformer-based Segmentation Model

Architecture: SegFormer is a modern Transformer-based architecture specifically designed for efficient semantic segmentation (proposed by Xie et al. in 2021). Instead of implementing SegFormer from scratch, we leveraged a pretrained SegFormer model via the Hugging Face Transformers library. Specifically, we used SegFormer-B0, which is the smallest variant of SegFormer, pretrained on the ADE20K dataset (a general segmentation dataset). Key points about SegFormer:

Configuration: Using Hugging Face made the setup straightforward.

Dataset Integration: We wrote a custom SegFormerDataset that overrides how data is loaded:

```python
# Load HuggingFace SegFormer feature extractor
feature_extractor = SegformerFeatureExtractor(do_reduce_labels=False)

class SegFormerDataset(Dataset):
    def __init__(self, image_dir, mask_dir):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.image_names = sorted([f for f in os.listdir(image_dir) if f.endswith('.png')])
        self.mask_names = sorted([f for f in os.listdir(mask_dir) if f.endswith('.png')])

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

    def __getitem__(self, idx):
        img = Image.open(os.path.join(self.image_dir, self.image_names[idx])).convert('RGB')
        msk = Image.open(os.path.join(self.mask_dir, self.mask_names[idx])).convert('L')

        # Resize mask to 1 channel binary map
        msk = msk.resize((512, 512))
        msk = np.array(msk)
        msk = (msk > 0).astype(np.int64)  # Binary mask

        # Use feature extractor on image
        encoding = feature_extractor(images=img, return_tensors="pt")
        pixel_values = encoding['pixel_values'].squeeze()

        return pixel_values, torch.tensor(msk, dtype=torch.long)
```

Using the feature extractor ensures our images are processed exactly as the SegFormer model expects (normalized with the same mean/std and scaled properly). Hyperparameters:

Loss Function & Optimizer & Scheduler:

```python
# Optimizer, loss, scheduler
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)
loss_fn = torch.nn.CrossEntropyLoss()
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.1)
```

Batch Size: Due to higher memory usage of SegFormer, we initially tested with batch size 2 for debugging, but ultimately we used batch size 4 for training (the model is efficient and could handle 4 on Colab's RAM when using CPU; on GPU, 4 is also fine for B0).

Training Procedure:

```python
epochs = 20
for epoch in range(epochs):
    start_time = time.time()
    model.train()
    train_loss = 0.0

    for imgs, masks in train_loader:
        imgs, masks = imgs.to(device), masks.to(device)

        outputs = model(pixel_values=imgs)
        logits = outputs.logits  # [B, num_classes, H', W']

        # Resize logits to match masks
        upsampled_logits = F.interpolate(
            logits,
            size=masks.shape[-2:],
            mode='bilinear',
            align_corners=False
        )

        loss = loss_fn(upsampled_logits, masks)
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
```

We then computed the cross-entropy loss on those upsampled logits vs the ground truth mask.

Validation: Similar to others, we evaluated on a validation set and computed a validation loss

each epoch. SegFormer's performance was strong from early on due to pretrained weights, and we observed rapid convergence.

SegFormer ended up performing the best among our models due to its powerful pretrained features and efficient architecture. It was able to accurately capture even small lesions. The benefit of using a pretrained model is evident in both the performance and the reduced need for a large number of training epochs.

6.4 U-Net with Pretrained EfficientNet Encoder

Architecture: U-Net is a classic convolutional encoder-decoder network with skip connections, widely used for medical image segmentation. In this project, we used a U-Net implemented via the segmentation_models_pytorch (smp) library for convenience, specifically:

Encoder: EfficientNet-B0 (pretrained on ImageNet). EfficientNet-B0 is a lightweight convolutional network known for a good balance of accuracy and efficiency. Using it as the encoder means the first half of the U-Net is essentially the EfficientNet feature extractor, which transforms the image into a set of multi-scale feature maps. Pretrained weights (encoder_weights='imagenet') allow the model to start with learned low-level features like edge detectors, color blob detectors, etc., which should help given the limited dataset size.

```
# Choose UNet or UnetPlusPlus
model = smp.Unet(
    encoder_name="efficientnet-b0",        # Pretrained encoder
    encoder_weights="imagenet",            # Use ImageNet weights
    in_channels=3,
    classes=1                              # Binary segmentation
).to('cuda' if torch.cuda.is_available() else 'cpu')
```

Decoder:

```
# Get prediction
outputs = model(image)

# Process output based on model type
if model_type in ['SimpleSegNet', 'ViT from Scratch', 'UNet']:
    # For models with Sigmoid output (binary classification per pixel)
    preds = (torch.sigmoid(outputs) > 0.5).squeeze(0).cpu().numpy().astype(np.uint8)
elif model_type == 'SegFormer':
     # For SegFormer with CrossEntropy output (multiclass classification)
     # Resize logits to original image size (or a desired output size)
     upsampled_logits = F.interpolate(
        outputs.logits,
        size=image.shape[-2:], # Assuming desired output size matches input size after feature extractor
        mode='bilinear',
        align_corners=False
    )
     preds = torch.argmax(upsampled_logits, dim=1).squeeze(0).cpu().numpy().astype(np.uint8)
```

Loss Function & Optimizer:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
loss_fn = smp.losses.DiceLoss(mode='binary')
optimizer = optim.Adam(model.parameters(), lr=1e-4)
```

Training Loop: We trained for 20 epochs. In each epoch, we iterated through the training DataLoader:

```
epochs = 20
model.train()

for epoch in range(epochs):
    running_loss = 0.0
    for images, masks in dataloader:
        images, masks = images.to(device), masks.to(device)

        preds = model(images)
        loss = loss_fn(preds, masks)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss:.4f}")
```

The U-Net model with a pretrained encoder performed very well due to the strong features from EfficientNet and the robust architecture with skip connections (which help in capturing fine details of lesions). It strikes a good balance between the simplicity of CNNs and the power of transformers, and as expected, its results were among the top (closely rivaling SegFormer).

7. Clinical Integration and Future Directions

Successful implementation of an automated screening system will ultimately depend on how well it integrates into the existing healthcare workflows. Key considerations include:

UI Development:Develop user-friendly interfaces so health sector people can work on it without any rigorous technical training. That translates to automated image quality reviewer and simple risk stratification views.

Regulatory Compliance: Complying with regulations for Medical Device, GDPR etc. This would require significant documentation of how models are considered and validated.

Continual Learning: Deployment can support the deployment followed by learning frameworks for ongoing improvements as data from deployment sites become available.

8. Conclusion

This guide lays the foundation for a full-featured, automated diabetic retinopathy screening application with the latest in deep learning technology. The multimodal nature guarantees relevance to various clinical scenarios and the focus on access allows application in resource-limited settings.

The practical impact of this is more significant than a technical achievement. Lowering the barrier to specialist-level screening could help to prevent thousands of cases of avoidable blindness. Every early detection of success represents a family without the weight visual impairment imposes.

The methods described in this handbook are the state-of-the-art of medical image analysis, clearly combining well-established techniques with the latest transformer architectures. For the future, the next versions will include federated learning methods to perform privacy-preserving collaboratively studies with multiple medical centers.

This suggests that more advanced AI technologies are not exclusive to high-resource research centers. When done correctly, these tools can be targeted to high-need populations that have historically been excluded from specialty care.

References

Bala, R., Sharma, A. and Goel, N. (2024). Comparative Analysis of Diabetic Retinopathy Classification Approaches. *Archives of Computational Methods in Engineering