MODEL IMPLEMENTATION

Olivier Kanamugire

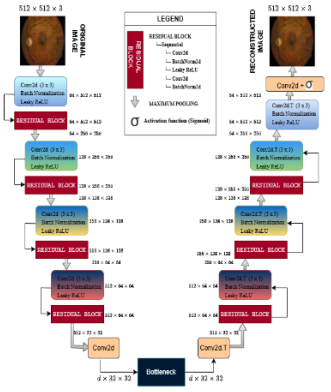
EXPERIMENT 1: LATENT REPRESENTATION BY AUTOENCODERS

This part presents how the autoencoders were implemented to find the best latent representation of color fundus photographs.

1.1 Data preprocessing: dataset.py

In Code 1 directory, the preprocessing is performed by Retinaldataset class. The image is loaded and converted to RGB, resized to $512 \times 512 \times 3$, apply Contrastive Limited Adaptive Histogram Equalization and transform to image tensor.

1.2 Model design: network.py



This is where the actual model is formulated. Two classes are implemented: ResidualBlock and Autoencoder. ResidualBlock is an identity map that uses two conv. layers and batch normalization and LeakyRELU activation function. Autoencoders on the other hand consist of contraction and expansion paths. The residual connection does not cross from the encoder to the decoder part but instead within each part. Encoder part has four conv. layers with two times number of channels consecutively. Each layer consists of conv. layer followed by batch normalization, LeakyRELU function, residual block and maximum pooling. From encoder to bottleneck maps to $32 \times 32 \times d$ where d represents the number of latent channels. From bottleneck to decoder takes the same map to previous space and then use similar upward paths by using conv.Transpose layers back to original input image.

1.3 Configuration and Utilities: config.py and utils.py

Config.py file contains all hyperparameters and their values. Utils file consists of losses that were utilized. There is a perceptual loss class that implements perceptual loss which is based on the vgg-16 pretrained model.

1.4 Training, testing and evaluation: train_test_loop.py and cross_validation.py

These two files present the training and testing loops but one uses normal training while cross_validation.py file incorporates cross validation strategy.

Parameters/hyperparameters	Description/value
Loss	Perceptual + MSE
Optimizer	Adam
Early stopping	10 epochs
Batch size	4
Scheduler	Patience = 3

In cross_validation.py, 5-fold cross-validation was utilized. It splits the entire dataset into 5 folds, trains the model k times (once per fold), each time using a different fold as the validation set and the remaining for training. Reconstructions, latent space and loss plots are stored for further analysis.

1.5 Classification model for computing Reconstruction Accuracy Ratio

This model utilized transfer learning where ResNet-50 is utilized as a pretrained model to perform classification on both original and corresponding reconstructions. For fine-tuning purposes, the final

fully connected layer was replaced with a custom classification head with a linear layer, ReLU activation function, and dropout. The final linear layer outputted five logits corresponding to the five classes.

EXPERIMENT 2: NEURAL ORDINARY DIFFERENTIAL EQUATIONS

2.1 ConvLatentODEFunc:

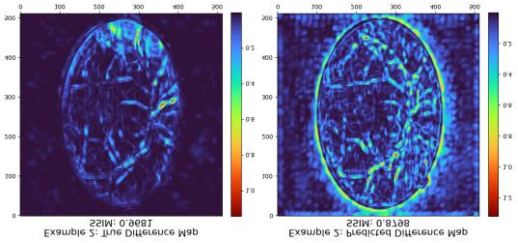
This module defines the temporal dynamics of latent image features and is implemented using convolutional layers. The architecture differs depending on the ODE solver:

- NODE_V1: For this version, two architectures were implemented similarly but with different depths according to a solver. For Euler and Fourth-Order Runge-Kutta (RK4), the model uses four convolutional layers. The first three layers transform the input to 128 channels, followed by Group Normalization (with d/2 groups), Sigmoid Linear Unit (SiLU) activation, and 10% Dropout. The final convolution maps feature back to the original number of channels. A residual connection is included to stabilize training and allow the model to learn relative changes from the initial state. Dormand Prince5 (Dopri5) and Adaptive Euler-Heun solvers are more computationally demanding. To accommodate resource limitations, version of the model with only three convolutional layers was used.

- NODE_V2: This is a smaller version with two convolutional layers, group normalization, and residual connection for stability and convergence. With NODE_V1, Dopri5 converges while RK4 and Euler fail to converge.

2.2 LatentODEModel:

This module integrates the learned latent dynamics using the odeint_adjoint function from the torchdiffeq library. It solves the differential equation given an initial latent state z0 over a specified time interval. The architecture is compatible with all used solvers (Euler, RK4, Dopri5, and Adaptive Euler-Heun). Training Process: The training loop is very similar to what was defined before in autoencoders; however, the perceptual loss was excluded because on this stage, high level features were not needed rather; element-wise values.



FULL MODEL PIPELINE