Configuration Manual

MSc Research Project
Data Analytics

Kaushik Kalyanaraman
Student ID: 21122211

School of Computing
National College of Ireland

Supervisor: Aaloka Anant

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Kaushik Kalyanaraman
Student ID:	21122211
Programme:	Data Analytics
Year:	2022-2023
Module:	MSc Research Project
Supervisor:	Aaloka Anant
Submission Due Date:	15/12/2022
Project Title:	Optimized Deep Learning Model For Diabetic Retinopathy Screening
Word Count:	932
Page Count:	12

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:	
Date:	15th December 2022

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

Kaushik Kalyanaraman
21122211

1 Introduction

The research involves implementation of a custom optimized CNN model for diabetic retinopathy(DR) screening using retinal fundus images. In this configuration manual, the author has included all the processes that might be necessary for replication are listed. The overall flow of the project will be explained along with code snippets from the relevant phases for better understanding.

2 System Configuration

The system configuration and software for carrying out this research are as mentioned below.

- Windows 11 system with a 3.2 GHz quad-core Intel Core i5 processor, 16 GB of 3200 MHz DDR4 RAM, and a 2 GB Nvidia GeForce MX450 graphics card with 512 GB SSD storage. This research requires around 10 GB of free space for smooth running.
- The tools and software used in this research are Jupyter notebook using Anaconda for python code implementation, and Microsoft Excel for the labeled file of the dataset.

3 Data Collection

The dataset for this research has been taken from a Kaggle public repository[1]. This data can be used by the public for any purpose. It consists of 3662 colored retinal fundus images under 5 categories along with a file containing the image and its category details for mapping purposes.

4 Required Libraries

This research requires different libraries like pandas, numpy, seaborn, matplotlib, os, tensorflow, keras, shutil, pickle, sklearn, cv2, tqdm, glob, datetime, math, random, time and zipfile. The libraries of sklearn and tensorflow will be used to calculate the metrics, label encoding, application of pretrained models for which the relavant code snippets are mentioned in sections .

[1]Dataset: `https://rb.gy/cvauju`

5 Data Preprocessing & Transformation

The dataset has 5 different categories of retinal fundus images namely Mild, Moderate, Severe, Proliferate and No DR. In this research, since the aim is to screen for diabetic retinopathy, hence it will require only 2 broad categories of DR and No DR as illustrated in Figure 2.

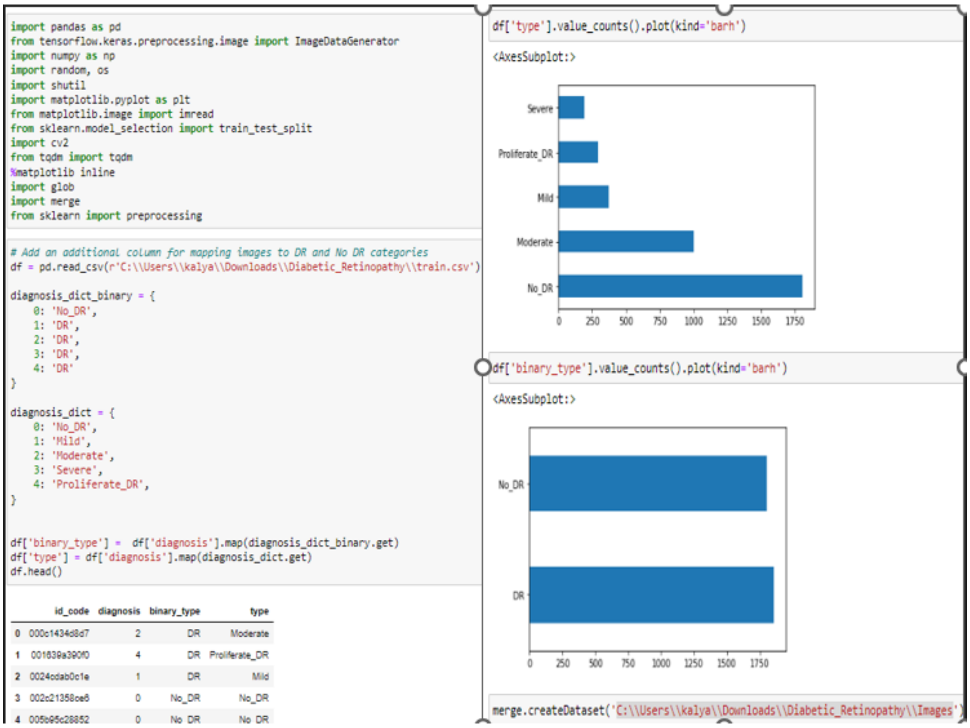

Figure 1: Data Understanding

Hence the pictures are merged according to their labels as shown in Figure 2. Once this is done, it is necessary to perform augmentation to increase the number of images and add more variety to the images so that model can be trained efficiently.Also, the images are then split into 80:20 train to test ratio so that it is prepared for modeling with the use of custom optimized CNN. For this, a new directory needs to be made in which train and test folders will be made through the code and those particular number of images will get transferred to the DR and No DR folders of train and test folders accordingly as shown in Figure 3.

At the end of this activity, there will be two separate folders. One will have two sub-folders of DR and No DR having 4800 images in each of them. The other folder will have test and train folders in which they will have their own DR and No DR folders containing 80% of images in train and 20% of the images in the test folders. Now, the images are ready in the required folders for the modeling process of each of the custom optimized CNN, ResNet 50 and Logistic Regression models.

```python
import shutil
import os


# Function to create new folder if not exists
def make_new_folder(folder_name, parent_folder):
    
    # Path
    path = os.path.join(parent_folder, folder_name)
    
    # Create the folder
    # 'new_folder' in
    # parent_folder
    try:
        # mode of the folder
        mode = 0o777

        # Create folder
        os.mkdir(path, mode)
    except OSError as error:
        print(error)

def createDataset(currPath):
    # current folder path
    current_folder = currPath

    print(current_folder)

    # list of folders to be merged
    list_dir = ['Mild',
    'Moderate',
    'Proliferate_DR',
    'Severe']

    # enumerate on list_dir to get the
    # content of all the folders ans store
    # it in a dictionary
    content_list = {}
    for index, val in enumerate(list_dir):
        path = os.path.join(current_folder, val)
        content_list[ list_dir[index] ] = os.listdir(path)

    # folder in which all the content will
    # be merged
    merge_folder = "DR"
```

```python
# folder in which all the content will
# be merged
merge_folder = "DR"

# merge_folder path - current_folder
# + merge_folder
merge_folder_path = os.path.join(current_folder, merge_folder)

# create merge_folder if not exists
make_new_folder(merge_folder, current_folder)

# loop through the list of folders
for sub_dir in content_list:

    # loop through the contents of the
    # list of folders
    for contents in content_list[sub_dir]:

        # make the path of the content to move
        path_to_content = sub_dir + "/" + contents

        # make the path with the current folder
        dir_to_move = os.path.join(current_folder, path_to_content )

        # move the file
        shutil.move(dir_to_move, merge_folder_path)

for entry in os.scandir(current_folder):
    if os.path.isdir(entry.path) and not os.listdir(entry.path) :
        os.rmdir(entry.path)
```

Figure 2: Data Merge

```python
target=4800#the target count for each class in df
gen=ImageDataGenerator(horizontal_flip=True,  rotation_range=20, width_shift_range=.2,
                        height_shift_range=.2, zoom_range=.2)
groups=df.groupby('labels') # group by class
for label in df['labels'].unique():  # for every class
    group=groups.get_group(label)  # a dataframe holding only rows with the specified label
    sample_count=len(group)   # determine how many samples there are in this class
    if sample_count< target: # if the class has less than target number of images
        aug_img_count=0
        delta=target-sample_count  # number of augmented images to create
        target_dir=os.path.join(r'C:\\Users\\kalya\\Downloads\\Diabetic_Retinopathy\\Images', label)  # define where to
        aug_gen=gen.flow_from_dataframe( group,  x_col='filepaths', y_col=None, target_size=(224,224), class_mode=None,
                                        batch_size=1, shuffle=False, save_to_dir=target_dir, save_prefix='aug-',
                                        save_format='png')
        while aug_img_count<delta:
            images=next(aug_gen)
            aug_img_count += len(images)
```

```
Found 1857 validated image filenames.
Found 1805 validated image filenames.
```

```python
# Create working directories for train/test
base_dir = 'C:\\Users\\kalya\\Downloads\\Diabetic_Retinopathy\\New_Folder'

train_dir = os.path.join(base_dir, 'train')
#val_dir = os.path.join(base_dir, 'val')
test_dir = os.path.join(base_dir, 'test')

if os.path.exists(base_dir):
    shutil.rmtree(base_dir)

if os.path.exists(train_dir):
    shutil.rmtree(train_dir)
os.makedirs(train_dir)

if os.path.exists(test_dir):
    shutil.rmtree(test_dir)
os.makedirs(test_dir)
```

```python
sdir=r'C:\\Users\\kalya\\Downloads\\Diabetic_Retinopathy\\Images'
classlist=os.listdir(sdir)
filepaths=[]
labels=[]
for klass in classlist:
    classpath=os.path.join(sdir,klass)
    if os.path.isdir(classpath):
        flist=os.listdir(classpath)
        for f in flist:
            fpath=os.path.join(classpath,f)
            filepaths.append(fpath)
            labels.append(klass)
Fseries=pd.Series(filepaths, name='filepaths')
Lseries=pd.Series(labels, name='labels')
df=pd.concat([Fseries, Lseries], axis=1)
print (df['labels'].value_counts())
```

```
DR       4800
No_DR    4800
Name: labels, dtype: int64
```

```python
#Split into train and test sets
train, test = train_test_split(df, test_size = 0.2, stratify = df['labels'])
```

```python
label_encoder = preprocessing.LabelEncoder()

df['labels'] = label_encoder.fit_transform(df['labels'])
```

```python
# Copy images to respective working directory
src_dir = r'C:\\Users\\kalya\\Downloads\\Diabetic_Retinopathy\\New_Folder'
for index, row in train.iterrows():
    diagnosis = row['labels']
    binary_diagnosis = row['labels']
    srcfile = row["filepaths"]
    dstfile = os.path.join(train_dir, binary_diagnosis)
    os.makedirs(dstfile, exist_ok = True)
    shutil.copy(srcfile, dstfile)

for index, row in test.iterrows():
    diagnosis = row['labels']
    binary_diagnosis = row['labels']
    srcfile = row["filepaths"]
    dstfile = os.path.join(test_dir, binary_diagnosis)
    os.makedirs(dstfile, exist_ok = True)
    shutil.copy(srcfile, dstfile)
```

Figure 3: Data Augmentation & Train Test Split

6 Custom Optimized CNN Model

Figure 4, Figure 5, Figure 6 and Figure 7 represent the code snippets for the custom optimized CNN. These include the import of the train and test split files, implementation code for CNN and the evaluation of the same.

```python
# Importing Necessary packages
%matplotlib inline
from sklearn.metrics import confusion_matrix
from datetime import timedelta
from os.path import isfile, isdir
from sklearn.metrics import confusion_matrix
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
import os
import cv2
import math
import pickle
import dataset
import random
import time
import zipfile
import seaborn as sns
```

```python
# Reading images from Training and Testing directories
images_folder_path = 'New_Folder'
train_path = images_folder_path +'/train'
test_path = images_folder_path +'/test'
checkpoint_dir = "models/"
```

```python
# batch size
batch_size = 32

# validation split, 20% of the data will be used for validation
validation_size = 0.2

# how long to wait after validation loss stops improving before terminating training
early_stopping = None  # use None if you don't want to implement early stopping

# image dimensions
img_size = 128

# Number of color channels for the images: 1 channel for grey-scale.
num_channels = 3

# Size of image when flattened to a single dimension
img_size_flat = img_size * img_size * num_channels

# Tuple with height and width of images used to reshape arrays.
img_shape = (img_size, img_size)

##Network graph params
#Conv Layer 1
filter_size_conv1 = 3
num_filters_conv1 = 32

#Conv Layer 2
filter_size_conv2 = 3
num_filters_conv2 = 32

#Conv Layer 3
filter_size_conv3 = 3
num_filters_conv3 = 64

# Fully-connected Layer
fc_layer_size = 128   # Number of neurons in fully-connected Layer.

#Number of different classes
classes = list(os.listdir(train_path))
num_classes = len(classes)
```

```python
data = dataset.read_train_sets(train_path, img_size, classes, validation_size=validation_size)
```

```
Loading training images
Reading files from folder DR (Index: 0)
Reading files from folder No_DR (Index: 1)
Finished reading train data
```

```python
#Load the test files
test_images, test_ids = dataset.read_test_set(test_path, img_size, classes)
print("Finished loading testing data")
```

```
Loading testing images
Reading files from folder DR (Index: 0)
Reading files from folder No_DR (Index: 1)
Finished loading test data
Finished loading testing data
```

```python
print("=================================================================")
print("Number of files in Training-set       ::\t{}".format(len(data.train.labels)))
print("Number of files in Validation-set     ::\t{}".format(len(data.valid.labels)))
print("Number of files in Test-set           ::\t{}".format(len(test_images)))
print("=================================================================")
```

```
=================================================================
Number of files in Training-set       ::        6144
Number of files in Validation-set     ::        1536
Number of files in Test-set           ::        1920
=================================================================
```

```python
def plot_images(images, cls_true, cls_pred=None):
    if len(images) == 0:
        print("no images to show")
        return
    else:
        random_indices = random.sample(range(len(images)), min(len(images), 9))

    images, cls_true  = zip(*[(images[i], cls_true[i]) for i in random_indices])
    # Create figure with 3x3 sub-plots.
    fig, axes = plt.subplots(3, 3)
    fig.subplots_adjust(hspace=0.8, wspace=1.8)

    for i, ax in enumerate(axes.flat):
        # Plot image.
        ax.imshow(images[i].reshape(img_size, img_size, num_channels))

        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, \nPred: {1}".format(cls_true[i], cls_pred[i])

        # Show the classes as the label on the x-axis.
        ax.set_xlabel(xlabel, horizontalalignment='center')

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

Figure 4: CNN Implementation - 1

Figure 4 describes some of the parameters that have been chosen for the custom optimized CNN model along with the loading process of the transformed data. Figure 5 explains the core part of the CNN code with all the details of the used layers in the optimized CNN model. It also contains the functions for the calculations for weights and biases for the model.

Figure 6 depicts the last part of the CNN function which includes the code for optimizer,

```python
from numpy.random import seed
seed(1)
tf.random.set_seed(2)
```

```python
# Method to create weights and biases
def create_weights(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.05))
def create_biases(size):
    return tf.Variable(tf.constant(0.05, shape=[size]))
```

```python
# Method to create feature detector
def create_convolutional_layer(input,              # The previous layer.
                               num_input_channels, # Num. channels in prev. layer.
                               conv_filter_size,   # Width and height of each filter.
                               num_filters,        # Number of filters.
                               use_pooling=True):  # Use 2x2 max-pooling.

    ## Define the weights that will be trained using create_weights function.
    weights = create_weights(shape=[conv_filter_size, conv_filter_size, num_input_channels, num_filters])

    ## Create biases using the create_biases function which are also trained.
    biases = create_biases(num_filters)

    ## Creating the convolutional layer
    layer = tf.nn.conv2d(input=input,
                         filter=weights,
                         strides=[1, 1, 1, 1],
                         padding='SAME')

    layer += biases

    ## We shall be using max-pooling.
    layer = tf.nn.max_pool(value=layer,
                           ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1],
                           padding='SAME')

    ## Output of pooling is fed to Relu which is the activation function for us.
    layer = tf.nn.relu(layer)

    # We return both the resulting layer and the filter-weights because we will plot the weights later.
    return layer, weights

# Method to convert n-dimensional array into flat array
def create_flatten_layer(layer):

    #Getting the shape of the layer from the previous layer.
    layer_shape = layer.get_shape()

    #Number of features will be img_height * img_width* num_channels.
    num_features = layer_shape[1:4].num_elements()

    #Flatten the layer so we shall have to reshape to num_features
    flattened_layer = tf.reshape(layer, [-1, num_features])

    # Return both the flattened layer and the number of features.
    return flattened_layer, num_features
```

```python
# Creating final neural netowk layers after feature detection
def create_fc_layer(input,
                    num_inputs,
                    num_outputs,
                    use_relu=True):

    #Let's define trainable weights and biases.
    weights = create_weights(shape=[num_inputs, num_outputs])
    biases = create_biases(num_outputs)

    #Calculate the layer as the matrix multiplication of the input and weights,
    #and add the bias-values, using matmul function
    layer = tf.matmul(input, weights) + biases

    #Using RELU
    if use_relu:
        layer = tf.nn.relu(layer)

    return layer
```

```python
#Placeholder variables for merging the type and shape of X and Y variables for model creation
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
x = tf.placeholder(tf.float32, shape=[None, img_size,img_size,num_channels], name='x')

#Labels
y_true = tf.placeholder(tf.float32, shape=[None, num_classes], name='y_true')
y_true_cls = tf.argmax(y_true, dimension=1)


#Conv Layer 1
layer_conv1, weights_conv1 = create_convolutional_layer(input=x,
                                                        num_input_channels=num_channels,
                                                        conv_filter_size=filter_size_conv1,
                                                        num_filters=num_filters_conv1,
                                                        use_pooling=True)


#Conv Layer 2
layer_conv2, weights_conv2 = create_convolutional_layer(input=layer_conv1,
                                                        num_input_channels=num_filters_conv1,
                                                        conv_filter_size=filter_size_conv2,
                                                        num_filters=num_filters_conv2,
                                                        use_pooling=True)

#Conv Layer 3
layer_conv3, weights_conv3 = create_convolutional_layer(input=layer_conv2,
                                                        num_input_channels=num_filters_conv2,
                                                        conv_filter_size=filter_size_conv3,
                                                        num_filters=num_filters_conv3,
                                                        use_pooling=True)

#Flatten Layer
layer_flat, num_features = create_flatten_layer(layer_conv3)

#FC Layer 1
layer_fc1 = create_fc_layer(input=layer_flat,
                            num_inputs=num_features, #layer_flat.get_shape()[1:4].num_elements(),
                            num_outputs=fc_layer_size,
                            use_relu=True)

#FC Layer 2
layer_fc2 = create_fc_layer(input=layer_fc1,
                            num_inputs=fc_layer_size,
                            num_outputs=num_classes,
                            use_relu=False)
```

Figure 5: CNN Implementation - 2

```python
#Predicted Class
y_pred = tf.nn.softmax(layer_fc2,name='y_pred')
y_pred_cls = tf.argmax(y_pred, dimension=1)


#Cost Function and optimizer
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=layer_fc2,
                                                        labels=y_true)

cost = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(cost)

#Performance measures
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

```python
# creating session variable to get and capture the runtime data accuracy
session = tf.Session()
session.run(tf.global_variables_initializer())

def show_progress(epoch, feed_dict_train, feed_dict_validate, val_loss):
    acc = session.run(accuracy, feed_dict=feed_dict_train)
    val_acc = session.run(accuracy, feed_dict=feed_dict_validate)
    msg = "Training Epoch {0} --> Training Accuracy: {1:>6.1%}, Validation Accuracy: {2:>6.1%},  Validation Loss: {3:.3f}"
    print(msg.format(epoch + 1, acc, val_acc, val_loss))
```

```python
saver = tf.train.Saver()

#Counter for total number of iterations performed so far.
total_iterations = 0

def train(num_iterations):
    #Ensure we are updating the global variable rather than a local copy.
    global total_iterations
    #Start-time used for printing time-usage below.
    start_time = time.time()

    best_val_loss = float("inf")
    patience = 0

    for i in range(total_iterations,
                   total_iterations + num_iterations):


        #Get a batch of training examples - x_batch now holds a batch of images and y_true_batch are the
        #true labels for those images.
        x_batch, y_true_batch, _, cls_batch = data.train.next_batch(batch_size)
        x_valid_batch, y_valid_batch, _, valid_cls_batch = data.valid.next_batch(batch_size)

        #Put the batch into a dict with the proper names for the placeholder variables in the TensorFlow graph.
        feed_dict_train =  {x:      x_batch,
                           y_true: y_true_batch}
        feed_dict_validatn = {x:      x_valid_batch,
                             y_true: y_valid_batch}

        #Run the optimizer using this training data batch
        session.run(optimizer, feed_dict=feed_dict_train)


        #Print status at end of each epoch (defined as full pass through training dataset).
        if i % int(data.train.num_examples/batch_size) == 0:
            val_loss = session.run(cost, feed_dict=feed_dict_validatn)
            epoch = int(i / int(data.train.num_examples/batch_size))

            show_progress(epoch, feed_dict_train, feed_dict_validatn, val_loss)
            saver.save(session, './DR Detection-model')
```

```python
            if early_stopping:
                if val_loss < best_val_loss:
                    best_val_loss = val_loss
                    patience = 0
                else:
                    patience += 1

                if patience == early_stopping:
                    break

    #Update the total number of iterations performed.
    total_iterations += num_iterations

    #Ending time.
    end_time = time.time()

    #Difference between start and end-times.
    time_dif = end_time - start_time

    #Print the time-taken.
    print("Time elapsed: " + str(timedelta(seconds=int(round(time_dif)))))

#train(num_iterations=3000)
```

```python
def plot_example_errors(cls_pred, correct):
    #cls_pred is an array of the predicted class-number for all images in the test-set.
    #correct is a boolean array whether the predicted class is equal to the true class for each image in the test-set.

    #Negate the boolean array.
    incorrect = (correct == False)

    #Get the images from the test-set that have been incorrectly classified.
    images = data.valid.images[incorrect]

    #Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    #Get the true classes for those images.
    cls_true = data.valid.cls[incorrect]

    #Plot the first 9 images.
    plot_images(images=images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9])

def plot_confusion_matrix(cls_pred):
    #Get the true classifications for the test-set.
    cls_true = data.valid.cls

    #Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_true,
                          y_pred=cls_pred)

    #Print the confusion matrix as text.
    print(cm)
    sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
    print('Sensitivity: ', sensitivity )
    specificity = cm[1,1]/(cm[1,0]+cm[1,1])
    print('Specificity: ', specificity)
    Precision = cm[0,0]/(cm[0,0]+cm[1,0])
    F1 = 2 * (Precision * sensitivity) / (Precision + sensitivity)
    print('F1 Score: ', F1)
    #Plot the confusion matrix as an image.
    plt.matshow(cm)
```

Figure 6: CNN Implementation - 3

cost function and cross entropy function. Apart from it, it shows the creation of a session for running the model with the function for confusion matrix and the evaluation metrics.

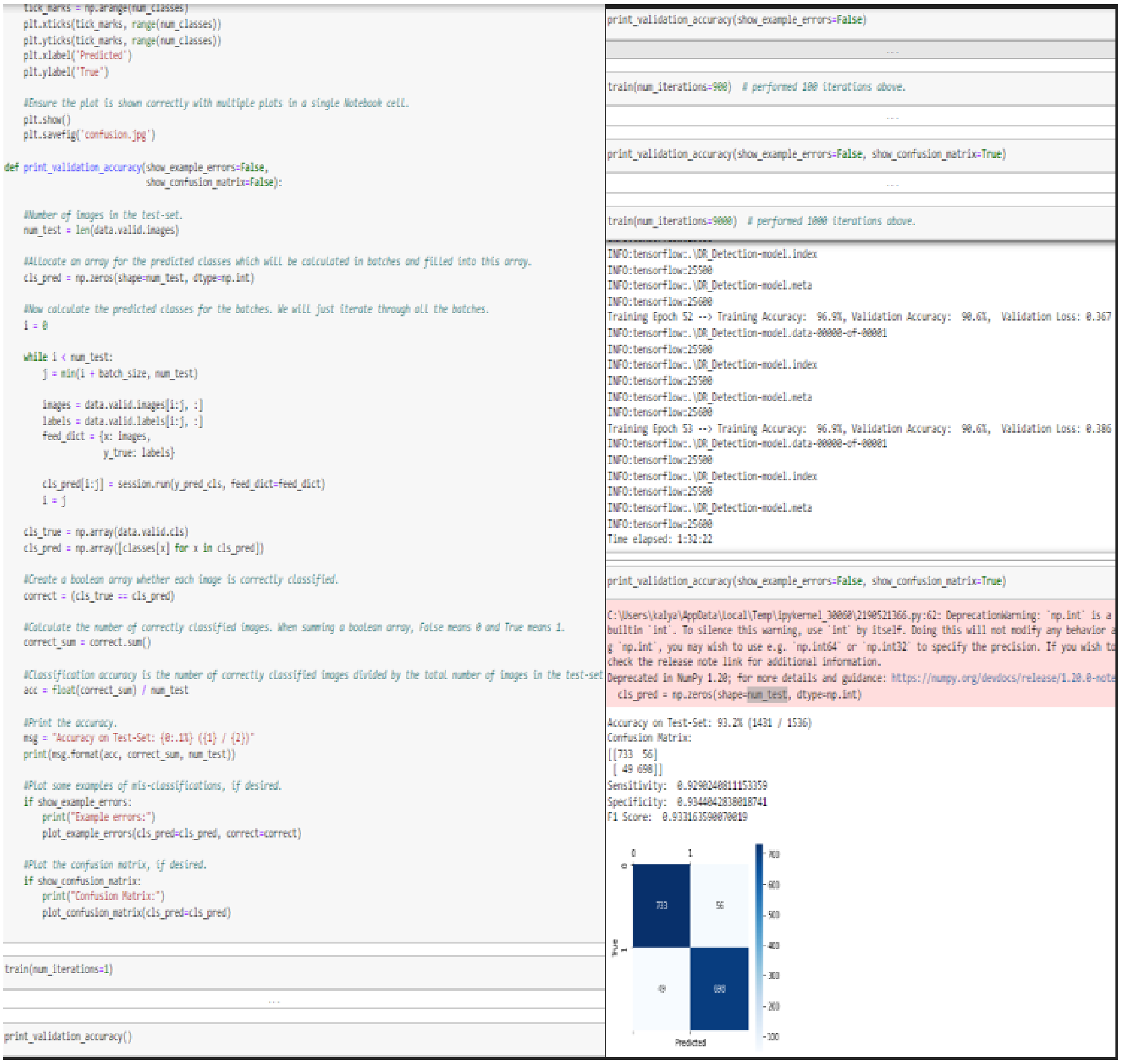

Figure 7: CNN Implementation - 4

7 ResNet 50 Model

In this model, the data is taken directly from the folders which have complete DR and No DR data. These are then loaded and label encoded so that they can be given as input to the model. It is then normalized and then split into train and test in ratio of 80:20. The train data is again divided into train and validation data in the same ratio. Now it is taken as an input into the model which is depicted by Figure 8.

```python
from tensorflow.keras.applications import ResNet50

res_model = ResNet50(input_shape=(64, 64,3), include_top=False, weights="imagenet")
```

```python
from tensorflow.keras.applications import ResNet50
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Flatten, GlobalAveragePooling2D
from tensorflow.keras.models import Sequential

for layer in res_model.layers:
    layer.trainable = False


res_model = Sequential()
res_model.add(ResNet50(include_top=False, weights='imagenet', pooling='max'))
res_model.add(Dense(1, activation='sigmoid'))
```

```python
res_model.compile(optimizer = tf.keras.optimizers.SGD(lr=0.0001), loss = 'binary_crossentropy', metrics = ['accuracy'])
```

...

```python
res_model.summary()
```

...

```python
resnet_history = res_model.fit(x_train, y_train, validation_data=([x_val], y_val),
                    epochs=10, batch_size=64, verbose=1, shuffle=True)
```

Figure 8: ResNet Implementation - 1

Figure 9 depicts the code snippet for the evaluation of the model along with the test result of the implementation.

```python
def plot_curves(resnet_history):

    loss = resnet_history.history["loss"]
    val_loss = resnet_history.history["val_loss"]

    accuracy = resnet_history.history["binary_accuracy"]
    val_accuracy = resnet_history.history["val_binary_accuracy"]

    epochs = range(len(resnet_history.history["loss"]))

    #plot loss
    plt.plot(epochs, loss, label = "training_loss")
    plt.plot(epochs, val_loss, label = "val_loss")
    plt.title("loss")
    plt.xlabel("epochs")
    plt.legend()

    #plot accuracy
    plt.figure()
    plt.plot(epochs, accuracy, label = "training_accuracy")
    plt.plot(epochs, val_accuracy, label = "val_accuracy")
    plt.title("accuracy")
    plt.xlabel("epochs")
    plt.legend()
```

```python
import matplotlib.pyplot as plt

# summarize history for accuracy
plt.plot(resnet_history.history['accuracy'])
plt.plot(resnet_history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(resnet_history.history['loss'])
plt.plot(resnet_history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```python
score = res_model.evaluate(x_test,y_test)
```

```
60/60 [==============================] - 62s 996ms/step - loss: 0.6441 - accuracy: 0.7719
```

```python
predict = res_model.predict(x_test)
```

```
60/60 [==============================] - 63s 992ms/step
```

```python
predict = predict.astype(int)
```

```python
#Get the confusion matrix using sklearn.
cm = confusion_matrix(y_true=y_test,y_pred=predict)

#Print the confusion matrix as text.
print(cm)
sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
print('Sensitivity: ', sensitivity )
specificity = cm[1,1]/(cm[1,0]+cm[1,1])
print('Specificity: ', specificity)
Precision = cm[0,0]/(cm[0,0]+cm[1,0])
F1 = 2 * (Precision * sensitivity) / (Precision + sensitivity)
print('F1 Score: ', F1)
#Plot the confusion matrix as an image.
plt.matshow(cm)

#Make various adjustments to the plot.
#plt.colorbar()
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
tick_marks = np.arange(2)
plt.xticks(tick_marks, range(2))
plt.yticks(tick_marks, range(2))
plt.xlabel('Predicted')
plt.ylabel('True')

#Ensure the plot is shown correctly with multiple plots in a single Notebook cell.
plt.show()
```

```
[[685 304]
 [134 797]]
Sensitivity:  0.6926188068756319
Specificity:  0.8560687432867884
F1 Score:  0.7577433628318585
```

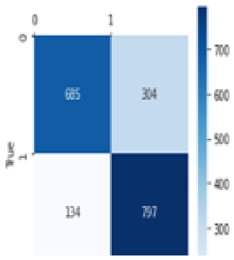

Figure 9: ResNet Implementation - 2

8 Logistic Regression Model

In this model, the complete set of 4800 images each in the DR and No DR folders are loaded and encoded post which they are converted into greyscale. Weight and bias are initialized to zero and sigmoid function is introduced. Figure 10 represents the forward and backward propagation code snippet.

```
# Forward and backward propagation
def propagate(X, Y, w, b):
    m = Y.shape[1]
    grads = {}
    A = sigmoid(w.T @ X + b) # forward propagation
    cost = -1/m * (np.sum (Y * np.log(A)) + np.sum((1 - Y) * np.log(1 - A))) # cost function

    # backward propagation
    grads['dw'] = 1/m * X @ (A - Y).T
    grads['db'] = 1/m * np.sum(A - Y, axis = 1, keepdims = True)

    return A, grads, cost

# Logistic regression function
def logistic_regression(X, Y, learning_rate = 0.0006, num_iter = 200, print_cost = True):
    w, b = initialize_parameters(X.shape[0]) # initailize the parameters
    costs = []

    # logistic regression
    for i in range(num_iter):
        A, grads, cost = propagate(X, Y, w, b)
        if print_cost and i % 20 == 0:
            print("Iteration #" + str(i) + "\tCost value = " + str(cost))
        costs.append(cost)
        w -= learning_rate * grads['dw']
        b -= learning_rate * grads['db']

    # compute the cost of the final parameter
    A, grads, cost = propagate(X, Y, w, b)
    print("Final cost value = " + str(cost))
    costs.append(cost)

    return w, b, costs
```

Figure 10: Logistic Regression Implementation - 1

Figure 11 depicts the model training phase of the Logisitc Regression model with the required parameters.

Figure 12 shows the evaluation part of the implemented Logistic Regression model.

```python
# Split the data to train/test set
X_train, X_test, Y_train, Y_test = train_test_split(X.T, Y.T, test_size = 0.25, random_state = 5)
X_train, X_test, Y_train, Y_test = X_train.T, X_test.T, Y_train.T, Y_test.T

# Print some detail
m = Y.shape[1]
m_train, m_test = Y_train.shape[1], Y_test.shape[1]
y1_train, y1_test = np.sum(Y_train), np.sum(Y_test)
print("number of train samples: " + str(m_train) + "(" + "{0:.2f}".format(m_train/m*100) + "%)")
print("number of DR cases in train samples: " + str(y1_train) + "(" + "{0:.2f}".format(y1_train/m_train*100) + "%)")
print("number of test samples: " + str(m_test) + "(" + "{0:.2f}".format(m_test/m*100) + "%)")
print("number of DR cases in test samples: " + str(y1_test) + "(" + "{0:.2f}".format(y1_test/m_test*100) + "%)")
```

...

```python
w, b = initialize_parameters(X.shape[0]) # initialize the parameters
w, b, costs = logistic_regression(X_train, Y_train, learning_rate = 0.0006, num_iter = 200)
_ = plt.plot(costs)
plt.xlabel("number of iterations")
plt.ylabel("Cost value")
plt.grid()
plt.show()
```

...

```python
# Predict function
def predict(X, Y, w, b):
    A, _, _ = propagate(X, Y, w, b)
    A[A >= 0.5] = 1
    A[A < 0.5] = 0
    diff = np.abs(A - Y)
    acc = 1 - np.sum(diff)/diff.shape[1]
    return A, diff, acc

yhat_train, _, acc_train = predict(X_train, Y_train, w, b) # Accuracy on train set
print("Accuracy on train set: " + "{0:.2f}".format(acc_train*100) + "%")

yhat_test, _, acc_test = predict(X_test, Y_test, w, b) # Accuracy on test set
print("Accuracy on test set: " + "{0:.2f}".format(acc_test*100) + "%")
```

Figure 11: Logistic Regression Implementation - 2

```
import seaborn as sns
#Get the confusion matrix using sklearn.
cm = confusion_matrix(y_true=Y_test[0],y_pred=yhat_test[0])
#Print the confusion matrix as text.
print(cm)
sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
print('Sensitivity: ', sensitivity )
specificity = cm[1,1]/(cm[1,0]+cm[1,1])
print('Specificity: ', specificity)
Precision = cm[0,0]/(cm[0,0]+cm[1,0])
F1 = 2 * (Precision * sensitivity) / (Precision + sensitivity)
print('F1 Score: ', F1)
#Plot the confusion matrix as an image.
plt.matshow(cm)

#Make various adjustments to the plot.
sns.heatmap(cm, annot=True,fmt="d", cmap='Blues')
tick_marks = np.arange(2)
plt.xticks(tick_marks, range(2))
plt.yticks(tick_marks, range(2))
plt.xlabel('Predicted')
plt.ylabel('True')

#Ensure the plot is shown correctly with multiple plots in a single Notebook cell.
plt.show()
```

```
[[ 799  393]
 [ 130 1078]]
Sensitivity:  0.6703020134228188
Specificity:  0.8923841059602649
F1 Score:  0.7534181989627534
```

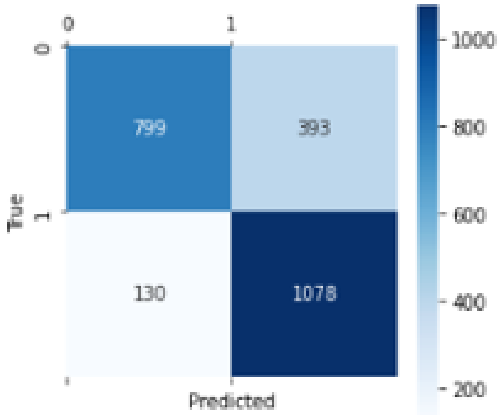

Figure 12: Logistic Regression Evaluation