```
# Code for Universal CNN Model. This code can train any kind of Image Dataset
# Author: Sujay Torvi , please contact me on sujay.torvi@gmail.com
# Copyright © 2024 MIT License

# Recommended System Requirements for Running the Below Code:
# CPU: Intel i5/Intel i7/Apple M1
# GPU: NVDIA GTX 1650 or better
# RAM: 8GB or Higher
# Storage: 256 GB or Higher
# Recommended to install CUDA library for Windows OS.


# The # code imports necessary libraries for building and training deep learning
models using TensorFlow and Keras.
# It includes TensorFlow for building and training neural networks, Keras for
building the model architecture, Matplotlib for visualization, Seaborn for creating
plots, and scikit-learn for computing classification metrics like classification
report and confusion matrix.
# Additionally, it imports VGG16, a pre-trained convolutional neural network model,
from Keras applications for transfer learning, and RMSprop optimizer for model
optimization.

import os
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
from tensorflow.keras.callbacks import LearningRateScheduler, ModelCheckpoint,
EarlyStopping
import seaborn as sns
from sklearn.metrics import classification_report, confusion_matrix
from keras.applications.vgg16 import VGG16, preprocess_input
from tensorflow.keras.optimizers import RMSprop

# The below code defines a function called setup_tpu.
# This function is designed to configure a Tensor Processing Unit (TPU) for use if
one is available.
# It first attempts to initialize the TPU cluster resolver and establish a
connection with the TPU system.
# If successful, it creates a distribution strategy specifically for TPUs.
# Otherwise, if it encounters an exception, it falls back to a default distribution
strategy for other hardware configurations.
# Finally, the function returns the selected strategy for use in distributing
computations across available devices.

def setup_tpu():
    try:
        tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
        print('Device:', tpu.master())
        tf.config.experimental_connect_to_cluster(tpu)
        tf.tpu.experimental.initialize_tpu_system(tpu)
        strategy = tf.distribute.experimental.TPUStrategy(tpu)
    except:
        strategy = tf.distribute.get_strategy()
    return strategy


# The function load_image_dataset is used to load image datasets from a specified
directory.
```

```
# It utilizes the image_dataset_from_directory function provided by TensorFlow's
Keras API.
# This function automatically creates a labeled dataset from image files in the
directory, splitting it into training and validation subsets, and applies
preprocessing such as resizing and batching.

def load_image_dataset(directory, subset, image_size, batch_size):
    return tf.keras.preprocessing.image_dataset_from_directory(
        directory,
        validation_split=0.2,
        subset=subset,
        seed=2024,
        image_size=image_size,
        batch_size=batch_size,
    )

# The function define_VGG_NET is used to define a VGG16 convolutional neural
network (CNN) architecture.
# It loads the pre-trained VGG16 model with ImageNet weights, excluding the fully
connected layers at the top.
# The function then sets the input shape according to the specified image_size and
returns the base VGG16 model.

def define_VGG_NET(image_size):
    # load base model
    vgg16_weight_path = 'imagenet'
    base_model = VGG16(
        weights=vgg16_weight_path,
        include_top=False,
        input_shape=image_size + (3,)
    )
    return base_model

# The function conv_block defines a block of convolutional layers with separable
convolutions followed by batch normalization and max-pooling.
# It takes the number of filters as input and returns a sequential model
representing the convolutional block.
# The function dense_block defines a block of dense (fully connected) layers
followed by batch normalization and dropout regularization.
# It takes the number of units and the dropout_rate as inputs and returns a
sequential model representing the dense block.

def conv_block(filters):
    block = tf.keras.Sequential([
        layers.SeparableConv2D(filters, 3, activation='relu', padding='same'),
        layers.SeparableConv2D(filters, 3, activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPool2D()
    ])
    return block

def dense_block(units, dropout_rate):
    block = tf.keras.Sequential([
        layers.Dense(units, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(dropout_rate)
    ])
    return block
```

```
# The function one_hot_label converts labels into one-hot encoded format, where
each label is represented as a binary vector with a single '1' indicating the
class.
# It takes an image, its corresponding label, and the total number of num_classes
as inputs, returning the one-hot encoded label.
# The function get_num_images calculates the number of images for each class in the
dataset directory.
# It takes class_names (list of class names) and dataset_dir (directory containing
class-wise image folders) as inputs, returning a list of the number of images for
each class.

# Function to one-hot encode labels
def one_hot_label(image, label, num_classes):
    label = tf.one_hot(label, num_classes)
    return image, label

# Function to get number of images for each class
def get_num_images(class_names, dataset_dir):
    NUM_IMAGES = []
    for label in class_names:
        dir_name = os.path.join(dataset_dir, label)
        NUM_IMAGES.append(len([name for name in os.listdir(dir_name)]))
    return NUM_IMAGES

# The function exponential_decay generates a function that computes the learning
rate for each epoch based on an exponential decay formula.
# It takes two arguments: lr0, the initial learning rate, and s, a parameter
controlling the decay rate.
# Inside the function, for each epoch, the learning rate is calculated using the
formula: lr0 * 0.1 ** (epoch / s), where epoch is the current epoch number. This
formula gradually reduces the learning rate over time.

def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1 ** (epoch / s)
    return exponential_decay_fn

# This code defines a function named plot_images_predictions.
# It takes a dataset of images, their corresponding labels, and a trained model as
input.
# For each batch of images in the dataset, it predicts the labels using the model
and plots the images along with their actual and predicted labels.

def plot_images_predictions(dataset, class_names, model):
    plt.figure(figsize=(10, 10))

    for images, labels in dataset.unbatch().batch(9).take(1):
        predictions = model.predict(images)
        predicted_class_indices = np.argmax(predictions, axis=1)
        confidences = np.max(predictions, axis=1)

        for i in range(9):
            plt.subplot(3, 3, i + 1)
            plt.imshow(images[i].numpy().astype("uint8"))
            plt.axis("off")

            actual_idx = np.argmax(labels[i].numpy()) if labels[i].numpy().size > 1
else labels[i].numpy()
            actual_label = class_names[actual_idx]
            predicted_label = class_names[predicted_class_indices[i]]
            confidence = confidences[i]

            plt.title(f"Actual: {actual_label}\nPredicted: {predicted_label}\
nConfidence: {confidence:.2f}")

    plt.tight_layout()
    plt.show()

#The code prompts the user to input the path for the dataset directory.
# It then sets up TensorFlow logging to suppress non-error messages, initializes
the TPU strategy, and defines a constant for data preprocessing.
# The parameters are then generalized based on the dataset directory, including
extracting class names, determining the number of classes, defining the image size,
batch size, and calculating the number of images for each class.

dataset_dir = input('Enter the path for the dataset:\n')
# Constants and setup
tf.get_logger().setLevel('ERROR')
strategy = setup_tpu()
AUTOTUNE = tf.data.experimental.AUTOTUNE

# Generalize parameters
#dataset_dir = "/Users/sujaymukundtorvi/Desktop/ALY 6980 Capstone/Eye_Disease"
class_names = [name for name in os.listdir(dataset_dir) if
os.path.isdir(os.path.join(dataset_dir, name))]
num_classes = len(class_names)
image_size = (150, 150)
batch_size = 16 * strategy.num_replicas_in_sync
NUM_IMAGES = get_num_images(class_names, dataset_dir)

# This code loads the training and validation datasets from the specified
directory, applying preprocessing and one-hot encoding to the labels.
# The `load_image_dataset` function is used to load the datasets with the specified
image size and batch size.
# After loading, the `map` function is applied to each dataset, transforming the
labels into one-hot encoded format and caching/prefetching the data for
optimization.

# Load datasets
train_ds = load_image_dataset(dataset_dir, "training", image_size, batch_size)
val_ds = load_image_dataset(dataset_dir, "validation", image_size, batch_size)

# Apply preprocessing and one-hot encoding
train_ds = train_ds.map(lambda x, y: one_hot_label(x, y, num_classes),
num_parallel_calls=AUTOTUNE).cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.map(lambda x, y: one_hot_label(x, y, num_classes),
num_parallel_calls=AUTOTUNE).cache().prefetch(buffer_size=AUTOTUNE)


### This code builds and compiles a neural network model using the TPU strategy.
### It first checks if there are more than 2 classes in the dataset.
### If there are more than 2 classes, it constructs a convolutional neural network
(CNN) model using specified layers.
### Otherwise, it uses a pre-trained VGG network as the base model and adds
additional layers for binary classification.
### The model is compiled with binary crossentropy loss, Adam optimizer with a
specified learning rate, and evaluation metrics.
### This ensures our code is as generalized as possible
```

```
# Build and compile the model using TPU strategy
with strategy.scope():
    if num_classes > 2:
        model = tf.keras.Sequential([
        layers.experimental.preprocessing.Resizing(*image_size),
        layers.Conv2D(16, 3, activation='relu', padding='same'),
        layers.Conv2D(16, 3, activation='relu', padding='same'),
        layers.MaxPool2D(),
        conv_block(32),
        conv_block(64),
        conv_block(128),
        layers.Dropout(0.2),
        conv_block(256),
        layers.Dropout(0.2),
        layers.Flatten(),
        dense_block(512, 0.7),
        dense_block(128, 0.5),
        dense_block(64, 0.3),
        layers.Dense(num_classes, activation='softmax')
    ])

    else:
        model = tf.keras.Sequential()
        model.add(define_VGG_NET(image_size))
        model.add(layers.Flatten())
        model.add(layers.Dropout(0.5))
        model.add(layers.Dense(num_classes, activation='sigmoid'))

        model.layers[0].trainable = False


    METRICS = [tf.keras.metrics.AUC(name='auc'),
tf.keras.metrics.CategoricalAccuracy(name='acc'), tf.metrics.F1Score()]
    model.compile(
    loss='binary_crossentropy',
    optimizer=tf.keras.optimizers.Adam(lr=0.0001),
    metrics=METRICS
    )

# This code initiates the training process of the neural network model.
# It specifies the number of epochs for training.
# Additionally, it sets up callbacks including learning rate scheduling, model
checkpointing to save the best model, and early stopping to prevent overfitting.
# The model is trained using the training dataset while validating it on the
validation dataset, and the training history is stored.

# Training and callbacks
EPOCHS = 100
lr_scheduler = LearningRateScheduler(exponential_decay(0.01, 20))
checkpoint_cb = ModelCheckpoint("universal_cnn.h5", save_best_only=True)
early_stopping_cb = EarlyStopping(patience=15, restore_best_weights=True)
history = model.fit(train_ds, validation_data=val_ds, callbacks=[checkpoint_cb,
early_stopping_cb, lr_scheduler], epochs=EPOCHS)

test_dir = input('Enter the path for the test set')


# This code evaluates the trained model on the test dataset to assess its
```

```
performance.
# It loads the test dataset from the specified directory with the given image size
and batch size.
# The test dataset is then preprocessed and one-hot encoded, and the model's
performance metrics are computed and printed.

# Evaluate the model on the test dataset
test_ds = tf.keras.preprocessing.image_dataset_from_directory(
    test_dir,
    image_size=image_size,
    batch_size=batch_size,
)
test_ds = test_ds.map(lambda x, y: one_hot_label(x, y, num_classes),
num_parallel_calls=AUTOTUNE).cache().prefetch(buffer_size=AUTOTUNE)
_ = model.evaluate(test_ds)

# This code loads a trained model from a specified file path.
# It then plots images with their corresponding predictions using the loaded model
and the validation dataset.
# Additionally, it generates and displays a confusion matrix based on the model's
predictions and true labels from the validation dataset.

# Load the trained model
model_path = 'universal_cnn.h5'
model = tf.keras.models.load_model(model_path)

# Plot images with predictions
plot_images_predictions(val_ds, class_names, model)

# Generate and display confusion matrix
y_true = []
y_pred = []

for img_batch, label_batch in val_ds:
    preds = model.predict(img_batch, verbose=False)
    pred_labels = np.argmax(preds, axis=1)
    y_pred.extend(pred_labels)

    true_labels = np.argmax(label_batch.numpy(), axis=1)
    y_true.extend(true_labels)


cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='g', cmap='Blues', xticklabels=class_names,
yticklabels=class_names)
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
```