**Image Classification with TensorFlow**

**Overview**

This script demonstrates the process of building and training an image classification model using TensorFlow and Keras. The script covers data loading, preprocessing, model building, training, and evaluation. The primary goal is to classify images into different categories based on a directory structure.

**Prerequisites**

* TensorFlow
* Keras
* NumPy
* Matplotlib
* scikit-learn

**1. Import Libraries**

import os

import numpy as np

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers, models

from keras import Sequential, layers

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.preprocessing.image import load\_img

from keras.layers import Dense, Conv2D, MaxPooling2D, Flatten, BatchNormalization, Dropout

from tensorflow.keras.regularizers import l1, l2

from tensorflow.keras.optimizers import Adam

import matplotlib.pyplot as plt

* **os**: For file and directory operations.
* **numpy**: For numerical operations.
* **tensorflow**: For deep learning.
* **keras**: High-level API for building and training models.
* **sklearn.model\_selection**: For splitting datasets.
* **matplotlib.pyplot**: For plotting.

**2. Load and Visualize Data**

**Set Data Directory**

data\_dir = r'C:\Users\Nour Hesham\Documents\Project\_NeuralNetwork\data'

Specify the directory containing the images organized in subdirectories per class.

**List and Count Classes**

classes = os.listdir(data\_dir)

print(classes)

from collections import Counter

class\_counts = Counter()

for cls in classes:

class\_dir = os.path.join(data\_dir, cls)

class\_counts[cls] = len(os.listdir(class\_dir))

* **os.listdir**: Lists directories (class names).
* **collections.Counter**: Counts number of images per class.
* **plt.barh**: Plots a horizontal bar chart of class distribution.

**Display Sample Images**

class\_dirs = os.listdir(data\_dir)

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

for i, cls in enumerate(class\_dirs, start=1):

img\_path = os.path.join(data\_dir, cls, os.listdir(os.path.join(data\_dir, cls))[0])

img = load\_img(img\_path, target\_size=(128, 128))

plt.subplot(3, 3, i)

plt.imshow(img)

plt.title(cls)

plt.axis('off')

plt.suptitle('Random sample')

plt.show()

Displays sample images from each class.

**3. Prepare the Dataset**

**Create TensorFlow Dataset**

dataset = tf.keras.preprocessing.image\_dataset\_from\_directory(

data\_dir,

shuffle=True,

image\_size=(256, 256),

batch\_size=32,

labels='inferred',

label\_mode='int'

)

* **image\_dataset\_from\_directory**: Loads images from a directory structure into a TensorFlow dataset.

**Split Dataset**

def split\_dataset(ds, train\_ratio=0.8, val\_ratio=0.1, test\_ratio=0.1, shuffle=True):

dataset\_size = len(ds)

train\_size = int(train\_ratio \* dataset\_size)

val\_size = int(val\_ratio \* dataset\_size)

test\_size = dataset\_size - train\_size - val\_size

if shuffle:

ds = ds.shuffle(dataset\_size)

train\_dataset = ds.take(train\_size)

val\_dataset = ds.skip(train\_size).take(val\_size)

test\_dataset = ds.skip(train\_size + val\_size).take(test\_size)

return train\_dataset, val\_dataset, test\_dataset

* **split\_dataset**: Splits the dataset into training, validation, and test sets.

**Cache and Prefetch**

train\_ds = train\_ds.cache().shuffle(1000).prefetch(buffer\_size=tf.data.experimental.AUTOTUNE)

val\_ds = val\_ds.cache().shuffle(1000).prefetch(buffer\_size=tf.data.experimental.AUTOTUNE)

test\_ds = test\_ds.cache().shuffle(1000).prefetch(buffer\_size=tf.data.experimental.AUTOTUNE)

* **cache**: Caches the dataset to speed up data loading.
* **shuffle**: Randomizes the order of the dataset.
* **prefetch**: Improves performance by prefetching batches.

**4. Build and Compile the Model**

resize\_and\_rescale = tf.keras.Sequential([

tf.keras.layers.Resizing(256, 256),

tf.keras.layers.Rescaling(1.0 / 255)

])

model = Sequential()

model.add(Conv2D(64, kernel\_size=(3, 3), padding='same', activation='relu', input\_shape=(256, 256, 3)))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(128, kernel\_size=(3, 3), padding='same', activation='relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(256, kernel\_size=(3, 3), padding='same', activation='relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256, activation='relu', kernel\_regularizer=l2(0.001)))

model.add(Dropout(0.2))

model.add(Dense(128, activation='relu', kernel\_regularizer=l2(0.001)))

model.add(Dropout(0.1))

model.add(Dense(3, activation='softmax'))

model.compile(optimizer=Adam(learning\_rate=0.0001),

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

* **Sequential**: Initializes a sequential model.
* **Conv2D**: Adds convolutional layers.
* **BatchNormalization**: Normalizes activations.
* **MaxPooling2D**: Applies max pooling.
* **Flatten**: Flattens the output for dense layers.
* **Dense**: Fully connected layers.
* **Dropout**: Regularizes the model by dropping out neurons during training.

**5. Train and Evaluate the Model**

**Training**

history = model.fit(train\_ds, epochs=20, batch\_size=32, verbose=1, validation\_data=val\_ds)

* **fit**: Trains the model.

**Evaluation**

model.evaluate(test\_ds)

* **evaluate**: Evaluates the model on the test set.

**Plot Training History**

plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.title('Training and Validation Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend()

plt.show()

plt.plot(history.history['accuracy'], label='Training Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.title('Training and Validation Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

* **plt.plot**: Plots training and validation loss/accuracy.

**6. Predict and Visualize Results**

y\_pred\_probs = model.predict(test\_ds)

print(y\_pred\_probs)

y\_pred = np.argmax(y\_pred\_probs, axis=1) # Predictions for the test set

print(y\_pred)

for images\_batch, labels\_batch in test\_ds.take(1):

plt.imshow((images\_batch[0].numpy().astype('uint8')))

plt.axis('off')

plt.show()

first\_image = images\_batch[0].numpy().astype('uint8')

first\_label = labels\_batch[0].numpy()

print('First image to predict')

plt.imshow(first\_image)

plt.title('First image actual label: ', class\_names[first\_label])

plt.axis('off')

plt.show()

batch\_prediction = model.predict(images\_batch)

print('Predicted label:', class\_names[np.argmax(batch\_prediction[0])])

* **predict**: Uses the model to predict classes for the test set.
* **imshow**: Displays images.
* **argmax**: Finds the index of the maximum value in predictions.

**7. Function for Prediction on New Images**

def predict(model, img):

img\_array = tf.keras.preprocessing.image.img\_to\_array(img)

img\_array = tf.expand\_dims(img\_array, 0)

predictions = model.predict(img\_array)

predicted\_class = class\_names[np.argmax(predictions[0])]

confidence = round(100 \* (np.max(predictions[0])), 2)

return predicted\_class, confidence

* **predict**: Function to predict the class and confidence of a new image.

**8. Display Predictions for a Batch of Images**

plt.figure(figsize=(15, 15))

for images, labels in test\_ds.take(2):

for i in range(9):

ax = plt.subplot(3, 3, i + 1)

plt.imshow(images[i].numpy().astype("uint8"))

predicted\_class, confidence = predict(model, images[i].numpy())

actual\_class = class\_names[labels[i]]

plt.title(f"Actual: {actual\_class},\n Predicted: {predicted\_class}.\n Confidence: {confidence}%")

plt.axis("off")

* **subplot**: Displays multiple images with predictions and confidence.