Building and Deploying an Image Classification Model for Flowers using TensorFlow

Introduction

This report details a tutorial on classifying images of flowers using a deep learning model built with TensorFlow's `tf.keras.Sequential`. The dataset is loaded using TensorFlow's `tf.keras.utils.image_dataset_from_directory`, which provides an efficient way to load and preprocess images directly from disk.

Objectives

The tutorial aims to demonstrate the following key machine learning concepts:

- **Efficient Data Loading**: Techniques for loading large datasets from disk in a way that is optimized for training neural networks.
- **Model Overfitting Mitigation**: Identifying signs of overfitting in models and applying strategies such as data augmentation and dropout to enhance model generalization.

Workflow Overview

The tutorial follows a structured machine learning workflow that includes:

1. **Examine and Understand Data**: Initial exploration and understanding of the dataset used for model training.
2. **Build an Input Pipeline**: Creating an efficient data pipeline to feed data into the model during training.
3. **Build the Model**: Constructing a convolutional neural network using the Sequential API in TensorFlow.
4. **Train the Model**: Training the model on the dataset, optimizing its parameters through backpropagation.
5. **Test the Model**: Evaluating the trained model's performance on a separate test dataset.
6. **Improve the Model**: Refining the model based on testing outcomes, and repeating the process to achieve better accuracy and generalization.

Setup

Before diving into the image classification task, we need to set up our environment by importing TensorFlow and other essential libraries. This will enable us to handle image data, build our deep learning model, and visualize the results.

Library Imports

The following libraries are imported to facilitate various stages of the workflow:

- **Matplotlib**: For creating visualizations of the data and model performance.
- **NumPy**: For efficient numerical computations and handling of image data arrays.
- **PIL (Python Imaging Library)**: To handle image data and perform any necessary transformations.
- **TensorFlow**: The main library used for building and training the neural network.

TensorFlow Setup

TensorFlow provides a comprehensive framework for developing machine learning models. We utilize its `keras` API to build and train a convolutional neural network. The specific imports include:

- **`tensorflow`**: The core library.
- **`keras`**: The high-level neural networks API, which is accessible through TensorFlow.
- **`layers`**: Provides a variety of layer classes that can be used to create neural network models.
- **`Sequential`**: A model type in Keras that allows us to build models layer-by-layer.

```
import matplotlib.pyplot as plt
import numpy as np
import PIL
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
```

By importing these libraries, we are equipped to process our dataset, construct the model architecture, and perform data visualization to gain insights into model performance.

Download and Explore the Dataset

This tutorial utilizes a dataset of approximately 3,700 photos of flowers. The dataset is structured into five sub-directories, each representing a different class of flowers:

```
flower_photo/
  daisy/
  dandelion/
  roses/
  sunflowers/
  tulips/
```

Dataset Acquisition

The dataset is hosted online and can be downloaded using TensorFlow's utility function. The following code snippet demonstrates how to download and extract the dataset:

```
import pathlib

dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
data_dir = tf.keras.utils.get_file('flower_photos.tar', origin=dataset_url, extract=True)
data_dir = pathlib.Path(data_dir).with_suffix('')
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz
228813984/228813984 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
```

+ Code + Markdown

Dataset Overview

After downloading, the dataset is structured into a directory with 3,670 total images. The dataset is well-organized, with each flower type in its respective sub-directory. Here's how you can count the total number of images:

```
image_count = len(list(data_dir.glob('*/*.jpg')))
print(image_count)
```

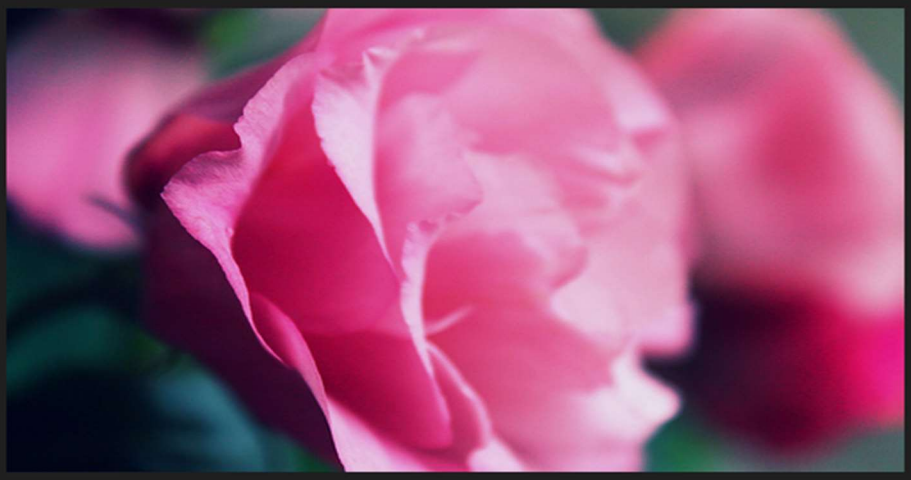
```
3670
```

Visualizing the Data

To gain an initial understanding of the dataset, we can visualize some sample images. Below are examples of roses and tulips from the dataset:

Roses:

```
roses = list(data_dir.glob('roses/*'))
PIL.Image.open(str(roses[0]))
```

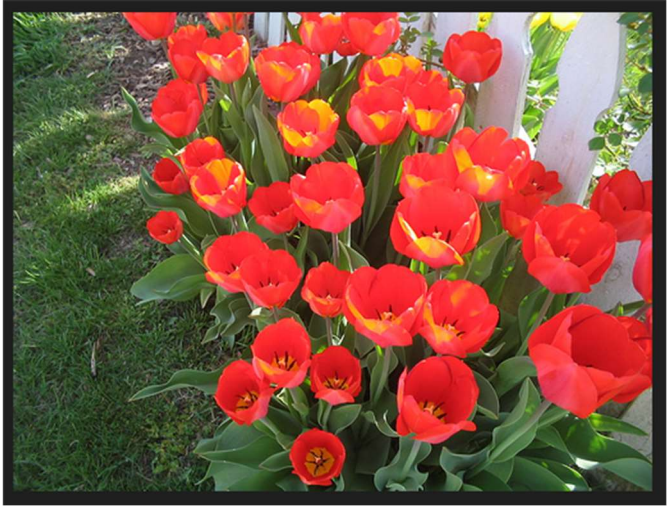


```
PIL.Image.open(str(roses[1]))
```

Tulips:

```
tulips = list(data_dir.glob('tulips/*'))
PIL.Image.open(str(tulips[0]))
```

```
PIL.Image.open(str(tulips[5]))
```



These images provide a glimpse into the variety and beauty of the flowers included in the dataset. Visual inspection helps in understanding the complexity and diversity of the data, which is crucial for the subsequent steps in model training and evaluation.

Load Data Using a Keras Utility

The next step involves loading the images from the disk into a format suitable for training a machine learning model. TensorFlow's `tf.keras.utils.image_dataset_from_directory` utility simplifies this process by converting a directory of images into a `tf.data.Dataset` object with just a few lines of code.

Create a Dataset

To load the dataset, we need to define some parameters:

- **`batch_size`**: The number of images to process in each batch during training. Here, it is set to 32.
- **`img_height`** and **`img_width`**: The dimensions to which all images will be resized. Both are set to 180 pixels.

It's a standard practice to split the dataset into training and validation subsets. In this case, we use 80% of the images for training and 20% for validation. This split helps in assessing the model's performance on unseen data and preventing overfitting.

Define some parameters for the loader:

```
batch_size = 32
img_height = 180
img_width = 180
```

It's good practice to use a validation split when developing your model. Use 80% of the images for training and 20% for validation.

```
train_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

```
Found 3670 files belonging to 5 classes.
Using 2936 files for training.
```

```
val_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

```
Found 3670 files belonging to 5 classes.
Using 734 files for validation.
```

Dataset Attributes

The created datasets (`train_ds` and `val_ds`) come with several useful attributes. One of the key attributes is `class_names`, which lists the class names in alphabetical order. These class names correspond to the sub-directory names in the dataset, allowing us to easily map between the dataset and the labels.

You can find the class names in the `class_names` attribute on these datasets. These correspond to the directory names in alphabetical order.

```
class_names = train_ds.class_names
print(class_names)
```

```
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

By loading the data in this manner, we ensure that it is efficiently batched, shuffled, and preprocessed for training the deep learning model. This setup is crucial for achieving optimal performance and accuracy in the model training phase.

Visualize the Data

Visualizing the data is a crucial step in understanding the dataset and verifying that the data loading process worked as expected. Here, we display the first nine images from the training dataset along with their corresponding labels.

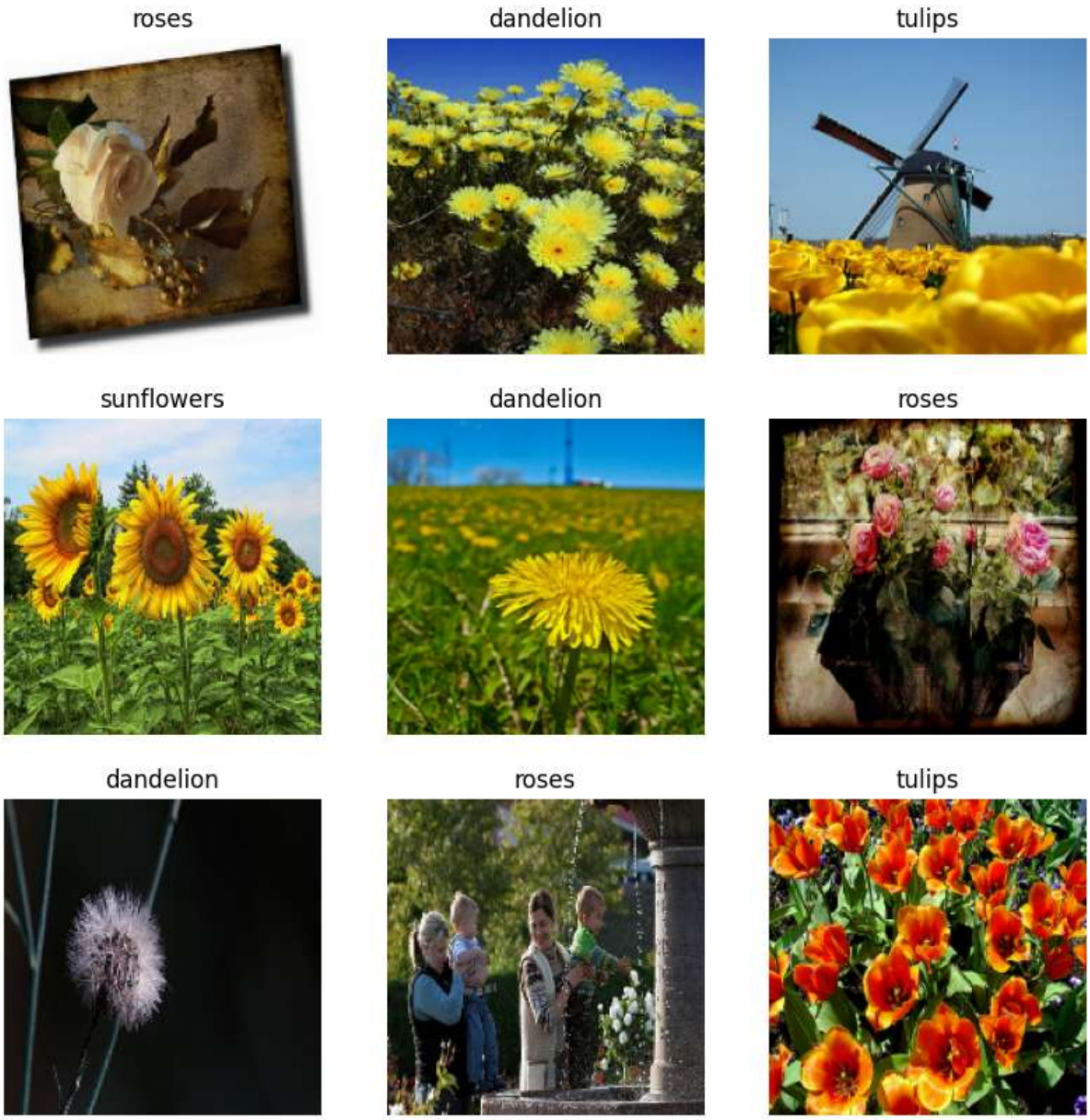
Displaying Sample Images

Using Matplotlib, we can create a grid of the first nine images in the training dataset. This visualization helps us ensure that the images are being correctly loaded and that the labels match the images.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```

This code will generate a 3x3 grid of images, displaying a small sample of each flower class from the dataset. Each image will be accompanied by its class label as the title.



Understanding the Dataset Structure

For further understanding, we can manually iterate over the dataset to explore the shape of the image batches and their corresponding labels.

You will pass these datasets to the Keras `Model.fit` method for training later in this tutorial. If you like, you can also manually iterate over the dataset and retrieve batches of images:

```
for image_batch, labels_batch in train_ds:
  print(image_batch.shape)
  print(labels_batch.shape)
  break
```

Python

```
(32, 180, 180, 3)
(32,)
```

+ Code + Markdown

The `image_batch` is a tensor with a shape of `(32, 180, 180, 3)`. This indicates a batch of 32 images, each of size `180x180` pixels with 3 color channels (RGB). The `labels_batch` is a tensor with a shape of `(32,)`, which represents the labels for the corresponding 32 images.

To convert these tensors to `numpy.ndarray` objects for further manipulation or analysis, you can use the `.numpy()` method.

Configuring the Dataset for Performance

Buffered Prefetching

Buffered prefetching is a technique used to enhance the efficiency of data loading during model training. It helps prevent I/O operations from becoming a bottleneck by overlapping data preprocessing with model execution. This ensures a continuous stream of data for training, reducing idle time and improving training performance.

Dataset Caching

Dataset.cache is a method used to store images in memory after they are loaded from disk during the first epoch. This method has the following benefits:

- **Reduces I/O Bottlenecks**: Once the dataset is cached, subsequent epochs can access the data more quickly as it does not need to be reloaded from disk.
- **On-Disk Caching**: If the dataset is too large to fit into memory, on-disk caching can still provide performance benefits by optimizing data access.

Prefetching Data

Dataset.prefetch allows for the overlapping of data preprocessing with model execution. This means while the model is training on one batch, the next batch can be prepared in advance. This approach minimizes wait times and improves overall training efficiency.

Implementation

The following code snippet demonstrates how to configure the training and validation datasets for optimal performance:

```
AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

- **`AUTOTUNE`**: TensorFlow's constant for automatically tuning the prefetch buffer size based on system resources and workload.
- **`cache()`**: Stores the dataset in memory after the first epoch to speed up future epochs.
- **`shuffle(1000)`**: Randomizes the order of the dataset with a buffer size of 1000 to improve model generalization.

- **`prefetch(buffer_size=AUTOTUNE)`**: Prepares batches in advance, reducing idle time during training.

Standardizing Image Data

Images used in neural networks typically have RGB channel values in the range `[0, 255]`. This range is not ideal for neural networks, which generally perform better with input values scaled to a smaller range. Standardizing the input values helps in faster convergence and improved model performance.

Standardization using `tf.keras.layers.Rescaling`

To standardize image pixel values to the `[0, 1]` range, you can use `tf.keras.layers.Rescaling`:

```
normalization_layer = layers.Rescaling(1./255)
```

There are two primary methods to apply this layer:

1. **Applying the Layer to the Dataset**

   You can standardize the dataset by mapping the normalization layer to the dataset:

```
normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
# Notice the pixel values are now in `[0,1]`.
print(np.min(first_image), np.max(first_image))
```

```
0.0 0.8430284
```

- **`map (lambda x, y: (normalization_layer(x), y))`**: Applies the normalization layer to the images in the dataset, keeping labels unchanged.
- **Pixel Value Range**: After normalization, pixel values should fall within the `[0, 1]` range.

- **Including the Layer Inside the Model**

For a more streamlined approach, especially during deployment, you can include the normalization layer directly in your model definition:

```
model = tf.keras.Sequential([
    normalization_layer,
    # Other layers...
])
```

Note: If you previously resized images using the `image_size` argument in `tf.keras.utils.image_dataset_from_directory`, you may also want to include resizing logic in your model using the `tf.keras.layers.Resizing` layer.

Building the Keras Model

Model Architecture

The model is built using the Keras `Sequential` API, consisting of three convolutional blocks, each followed by a max pooling layer, and a fully connected layer:

```
num_classes = len(class_names)

model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])
```

- **Convolutional Layers**: `Conv2D` layers with ReLU activation.
- **Max Pooling**: `MaxPooling2D` layers to reduce dimensionality.
- **Dense Layers**: Fully connected layers for classification.

Compiling the Model

The model is compiled with an optimizer, loss function, and metrics for evaluation:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

- **Optimizer**: `Adam` optimizer for adjusting weights.
- **Loss Function**: `SparseCategoricalCrossentropy` to handle multi-class classification.
- **Metrics**: `Accuracy` to track model performance.

Model Summary

The `Model.summary` method provides a detailed view of the model architecture, including each layer's configuration and parameters:

```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
rescaling_1 (Rescaling)	(None, 180, 180, 3)	0
conv2d (Conv2D)	(None, 180, 180, 16)	448
max_pooling2d (MaxPooling2D)	(None, 90, 90, 16)	0
conv2d_1 (Conv2D)	(None, 90, 90, 32)	4,640
max_pooling2d_1 (MaxPooling2D)	(None, 45, 45, 32)	0
conv2d_2 (Conv2D)	(None, 45, 45, 64)	18,496
max_pooling2d_2 (MaxPooling2D)	(None, 22, 22, 64)	0
flatten (Flatten)	(None, 30976)	0
dense (Dense)	(None, 128)	3,965,056
dense_1 (Dense)	(None, 5)	645

```
Total params: 3,989,285 (15.22 MB)

Trainable params: 3,989,285 (15.22 MB)

Non-trainable params: 0 (0.00 B)
```

Training the Model

The model is trained for a specified number of epochs using the `Model.fit` method:

```
epochs=10
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)
```

```
Epoch 1/10
92/92 ━━━━━━━━━━━━━━━━━━━━ 20s 154ms/step - accuracy: 0.3655 - loss: 1.4445 - val_accuracy: 0.5654 - val_loss: 1.0761
Epoch 2/10
92/92 ━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step - accuracy: 0.6055 - loss: 1.0077 - val_accuracy: 0.6376 - val_loss: 0.9530
Epoch 3/10
92/92 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.6931 - loss: 0.8065 - val_accuracy: 0.6485 - val_loss: 0.9309
Epoch 4/10
92/92 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - accuracy: 0.7567 - loss: 0.6440 - val_accuracy: 0.6608 - val_loss: 0.9058
Epoch 5/10
92/92 ━━━━━━━━━━━━━━━━━━━━ 3s 18ms/step - accuracy: 0.8662 - loss: 0.3902 - val_accuracy: 0.6499 - val_loss: 1.0644
Epoch 6/10
92/92 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - accuracy: 0.9320 - loss: 0.2373 - val_accuracy: 0.6431 - val_loss: 1.1101
Epoch 7/10
92/92 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - accuracy: 0.9602 - loss: 0.1287 - val_accuracy: 0.6553 - val_loss: 1.2790
Epoch 8/10
92/92 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - accuracy: 0.9750 - loss: 0.0969 - val_accuracy: 0.6226 - val_loss: 1.4799
Epoch 9/10
92/92 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - accuracy: 0.9800 - loss: 0.0821 - val_accuracy: 0.6676 - val_loss: 1.6600
Epoch 10/10
92/92 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - accuracy: 0.9956 - loss: 0.0252 - val_accuracy: 0.6444 - val_loss: 1.7559
```

- **Training Data**: `train_ds` for training.
- **Validation Data**: `val_ds` for evaluating model performance.

- **Epochs**: The model is trained for 10 epochs.

Visualizing Training Results

After training the model, it's crucial to evaluate its performance by visualizing loss and accuracy metrics. These plots help in understanding how well the model is performing and whether there are any issues such as overfitting.

Code for Visualization

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

The purpose of this code is to visualize how the model's performance changes over the training epochs by plotting accuracy and loss metrics for both training and validation datasets.

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)
```

Extracts accuracy and loss values for both training and validation datasets from the `history` object returned by the `model.fit()` method. Defines the range of epochs to be used for plotting.

```
plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
```

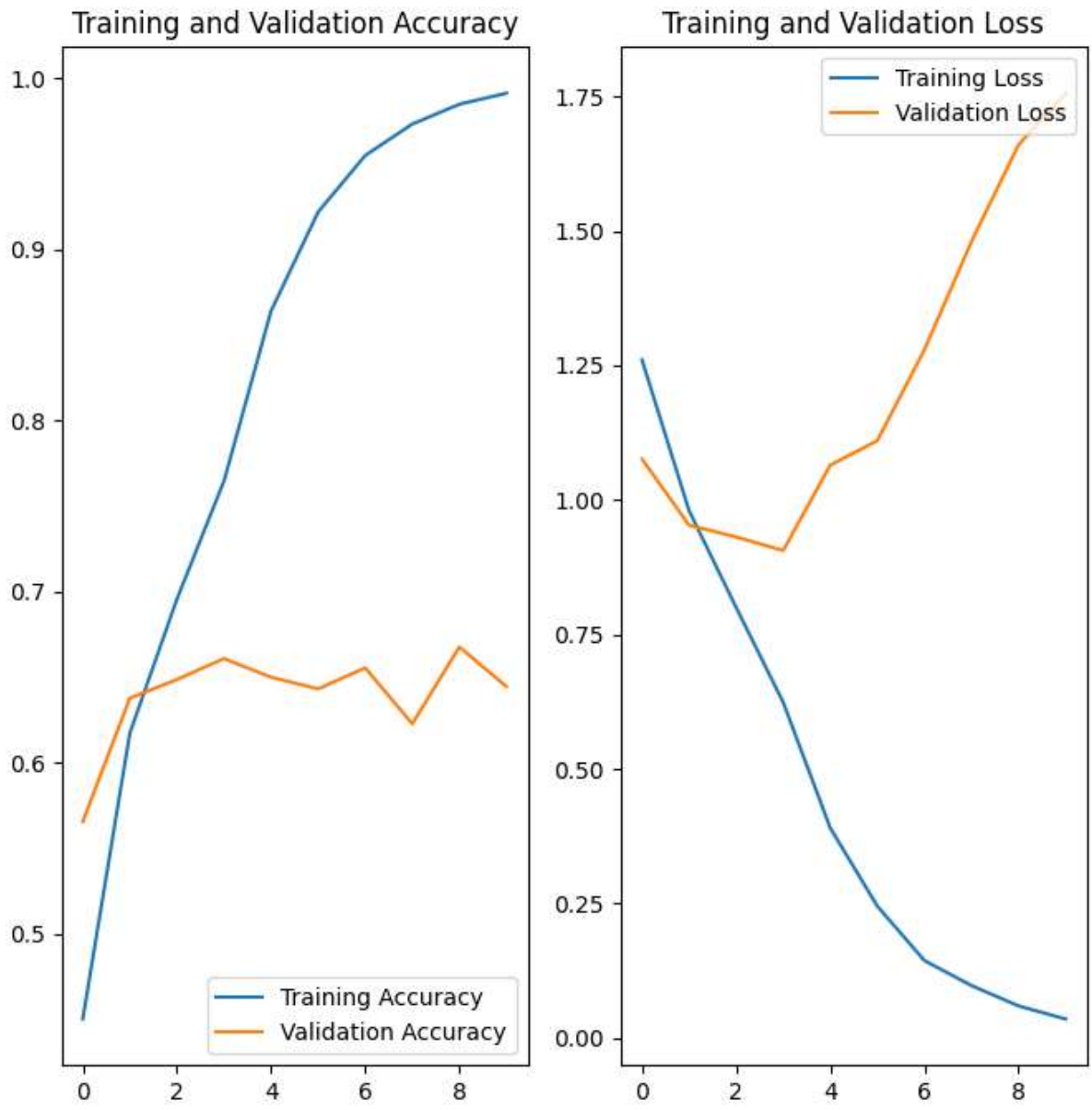
- `plt.subplot(1, 2, 1)`: Creates the first subplot for accuracy.
- `plt.plot(...)`: Plots the training and validation accuracy.
- `plt.legend(...)`: Adds a legend to distinguish between training and validation accuracy.

- `plt.title(...)`: Sets the title of the accuracy plot.

```
plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

- `plt.subplot(1, 2, 2)`: Creates the second subplot for loss.
- `plt.plot(...)`: Plots the training and validation loss.
- `plt.legend(...)`: Adds a legend for loss.
- `plt.title(...)`: Sets the title of the loss plot.

- `plt.show()`: Displays the plots.



Analysis

The plots reveal that while the training accuracy is steadily increasing, the validation accuracy remains stagnant around 60%. This discrepancy between training and validation accuracy suggests that the model is likely overfitting the training data. Overfitting occurs when a model performs well on training data but poorly on unseen data, indicating that it has learned to memorize the training examples rather than generalize from them.

Addressing Overfitting

Overfitting can be mitigated using several techniques, including data augmentation and dropout regularization.

Data augmentation helps to artificially increase the size and variability of the training dataset by applying random transformations.

Data Augmentation

Data augmentation involves generating additional training examples by applying random transformations to existing images. This approach exposes the model to a more diverse set of examples, helping it generalize better.

Implementation:

```
data_augmentation = keras.Sequential(
  [
    layers.RandomFlip("horizontal",
                      input_shape=(img_height,
                                   img_width,
                                   3)),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1),
  ]
)
```

Dropout

Dropout is a regularization technique where randomly selected neurons are ignored (dropped out) during training. This helps prevent the model from becoming overly reliant on specific neurons and improves its ability to generalize.

Implementation:

```
model = Sequential([
  data_augmentation,
  layers.Rescaling(1./255),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Dropout(0.2),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes, name="outputs")
])
```

Compile and Train the Enhanced Model

After adding data augmentation and dropout, recompile and train the model. The number of epochs is increased to 15 to allow the model to learn from the augmented data.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
model.summary()
```

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
sequential_1 (Sequential)	(None, 180, 180, 3)	0
rescaling_2 (Rescaling)	?	0 (unbuilt)
conv2d_3 (Conv2D)	?	0 (unbuilt)
max_pooling2d_3 (MaxPooling2D)	?	0 (unbuilt)
conv2d_4 (Conv2D)	?	0 (unbuilt)
max_pooling2d_4 (MaxPooling2D)	?	0 (unbuilt)
conv2d_5 (Conv2D)	?	0 (unbuilt)
max_pooling2d_5 (MaxPooling2D)	?	0 (unbuilt)
dropout (Dropout)	?	0 (unbuilt)
flatten_1 (Flatten)	?	0 (unbuilt)
dense_2 (Dense)	?	0 (unbuilt)
outputs (Dense)	?	0 (unbuilt)

```
Total params: 0 (0.00 B)


Trainable params: 0 (0.00 B)


Non-trainable params: 0 (0.00 B)
```

Train the Model:

```
epochs = 15
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs
)
```

Trains the model for 15 epochs, using `train_ds` for training and `val_ds` for validation.

These enhancements aim to improve the model's ability to generalize to new, unseen data and reduce overfitting. Further tuning and experimentation may be needed to optimize model performance based on specific requirements and dataset characteristics.

Visualize Training Results

After implementing data augmentation and dropout, it's important to visualize how these changes impacted the model's performance. This section plots the training and validation accuracy and loss, providing insight into how well the model has generalized.

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)
```

These lines retrieve the accuracy and loss values for both training and validation sets from the `history` object obtained from `model.fit()`.Creates a range object representing the number of epochs.

Create Accuracy Plot:

```
plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
```

- `plt.subplot(1, 2, 1)`: Sets up the first subplot for accuracy.
- `plt.plot(...)`: Plots training and validation accuracy.
- `plt.legend(...)`: Adds a legend for clarity.
- `plt.title(...)`: Sets the title for the accuracy plot.

Create Loss Plot:

```
plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

- `plt.subplot(1, 2, 2)`: Sets up the second subplot for loss.
- `plt.plot(...)`: Plots training and validation loss.
- `plt.legend(...)`: Adds a legend for the loss plot.
- `plt.title(...)`: Sets the title for the loss plot.
- `plt.show()`: Displays the plots.

The plots show how the model's training and validation accuracy and loss evolve over epochs. Ideally, the training and validation curves should converge, indicating good model generalization. Large discrepancies suggest potential overfitting or underfitting.

Predict on New Data

To evaluate the model's performance on new, unseen data, we make a prediction on an image that was not included in the training or validation sets.

Load Image:

```
sunflower_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/592px-Red_sunflower.jpg"
sunflower_path = tf.keras.utils.get_file('Red_sunflower', origin=sunflower_url)
```

Downloads the sunflower image from a URL and saves it locally.

Preprocess Image:

```
img = tf.keras.utils.load_img(
    sunflower_path, target_size=(img_height, img_width)
)
img_array = tf.keras.utils.img_to_array(img)
img_array = tf.expand_dims(img_array, 0) # Create a batch
```

- `tf.keras.utils.load_img(...)`: Loads the image and resizes it to the model's input size.
- `tf.keras.utils.img_to_array(...)`: Converts the image to a NumPy array.
- `tf.expand_dims(...)`: Adds a batch dimension to the image array.

Predict Image Class:

```
predictions = model.predict(img_array)
score = tf.nn.softmax(predictions[0])
```

- `model.predict(img_array)`: Generates predictions for the image.
- `tf.nn.softmax(predictions[0])`: Converts the model's output into probabilities.

Print Prediction:

```
print(
    "This image most likely belongs to {} with a {:.2f} percent confidence."
    .format(class_names[np.argmax(score)], 100 * np.max(score))
)
```

- `np.argmax(score)`: Gets the index of the highest probability.
- `class_names[np.argmax(score)]`: Retrieves the class name corresponding to the highest probability.
- `100 * np.max(score)`: Converts the probability to a percentage.

Note: Data augmentation and dropout are only applied during training. During inference, these layers are inactive, and the model uses the learned parameters directly.

Use TensorFlow Lite

TensorFlow Lite (TFLite) is a lightweight solution designed for running machine learning models on mobile, embedded, and edge devices. It converts trained models into a more compact format optimized for performance on these devices.

Convert the Keras Sequential Model to a TensorFlow Lite Model

To deploy the trained Keras Sequential model on mobile or edge devices, convert it to TensorFlow Lite format using the `TFLiteConverter`.

Code: Model Conversion

```
# Convert the model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model.
with open('model.tflite', 'wb') as f:
  f.write(tflite_model)
```

Convert the Keras model to a TensorFlow Lite model. This step optimizes the model for performance on mobile and edge devices. Save the TensorFlow Lite model to a file named `model.tflite`.

Run the TensorFlow Lite Model

Once the model is converted to TensorFlow Lite format, it can be loaded and used for inference on mobile or edge devices. Here's how to run the TensorFlow Lite model in Python:

- **Convert the Model:** Convert your trained Keras model to a TensorFlow Lite model format. This step reduces the model size and optimizes it for mobile and embedded devices.

- **Load the TensorFlow Lite Model:** Initialize a TensorFlow Lite interpreter with the converted model. Check the model's input and output details to understand how to interact with it.

- **Run Inference:** Use the interpreter to classify a new image by feeding it through the TensorFlow Lite model. Extract and process the model's output to get the classification results.

- **Compare Results:** Verify that the predictions from the TensorFlow Lite model are consistent with those from the original Keras model to ensure accuracy and reliability.

Verify Model Predictions:

To ensure that the TensorFlow Lite model produces similar predictions to the original model, compare the results:

Extract and Compare: Convert the output tensor to a NumPy array and compare it with the original model's predictions to ensure consistency.