prog2_cnn_flowers

September 26, 2024

1 CPSC320: Program 2 - Flower Classification Using CNN and Transfer Learning

In this programming assignment, you will build a CNN Model using Keras to classify images of flowers, visualize intermediate representations and use transfer learning to improve prediction accuracy.

Important: The notebook you will submit must be the one you have RUN all the cells (DO NOT CLEAR OUTPUTS OF ALL CELLS)

[3]:
```python
# Import TensorFlow and Keras
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

[4]:
```python
####################################################################################
# This is the Path to the training and validation dataset
# you may need to change the path if your dataset is stored in a different␣
↪directory
####################################################################################
train_dir = '../flowers_train_validation/train'
validation_dir = '../flowers_train_validation/validation'
TARGET_SIZE=(150,150)
```

1.1 1. CNN Model on Flower Prediction

1.1.1 1.1: Data Preparation and Augmentation

We'll use the ImageDataGenerator class to augment our training data and rescale the images. Data augmentation helps in increasing the diversity of the training data, which helps in reducing overfitting.

[7]:
```python
# Create ImageDataGenerators for training and validation data
# The train_datagen is used for augmenting the training data with various␣
↪transformations.
# The validation_datagen is only rescaled, as no augmentation is needed.
# Step 1: Create ImageDataGenerators for training and validation
train_datagen = ImageDataGenerator(rescale=1./255,
                                   rotation_range=20,
                                     width_shift_range=0.2,
                                     height_shift_range=0.2,
                                     shear_range=0.2,
                                     zoom_range=0.2,
                                     horizontal_flip=True,
                                     fill_mode='nearest')

validation_datagen = ImageDataGenerator(rescale=1./255)
```

[8]:
```
# Step 2: Create generators
train_generator = train_datagen.flow_from_directory(
    train_dir,  # This is the target directory
    target_size=TARGET_SIZE,  # All images will be resized to 150x150
    batch_size=32,
    class_mode='categorical'  # Since we use binary classification
)

validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=TARGET_SIZE,
    batch_size=32,
    class_mode='categorical'
)
```

```
Found 3456 images belonging to 5 classes.
Found 865 images belonging to 5 classes.
```

[9]:
```
# Get the original image shape
original_image_shape = train_generator.image_shape
print("Original image shape:", original_image_shape)
```

```
Original image shape: (150, 150, 3)
```

[ ]:
```

```

1.1.2 1.2: Building the CNN Model

Now we will build the CNN model using Keras' Sequential API. We'll start by adding convolutional layers followed by max-pooling layers.

1.1.3 Task 1: Building the CNN Model

You will build the CNN model using Keras' Functional API. You may choose any combination of convolutional layers followed by max-pooling layers, and finally the classifier layeres. You should pay attention to the followings: - Create an input layer that accepts the shape that is as same as the shape from the datagenerator (150,150,3) - You final output layer should match the number of classes (5 in our dataset) - Display model summary

Note: It is stronly suggested you build the model using functional API instead of Sequential model.

[12]:
```
import tensorflow as tf
from tensorflow.keras import layers, models

######################################################################
# Build your model (your source code below)
######################################################################
```

```
Model: "functional_1"


 Layer (type)                     Output Shape                Param #

 input_layer_1 (InputLayer)       (None, 150, 150, 3)               0

 conv2d_2 (Conv2D)                (None, 148, 148, 32)            896

 max_pooling2d_2 (MaxPooling2D)   (None, 74, 74, 32)                0

 conv2d_3 (Conv2D)                (None, 72, 72, 64)           18,496

 max_pooling2d_3 (MaxPooling2D)   (None, 36, 36, 64)                0

 flatten_1 (Flatten)              (None, 82944)                     0

 dense_2 (Dense)                  (None, 128)              10,616,960

 dense_3 (Dense)                  (None, 5)                       645


 Total params: 10,636,997 (40.58 MB)

 Trainable params: 10,636,997 (40.58 MB)

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

[ ]:
```

```

1.1.4 1.3: Compiling the Model

The model is compiled using the Adam optimizer and categorical crossentropy loss, which is suitable for multi-class classification problems.

[15]:
```
# Step 4: Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

1.1.5 1.4: Training the Model

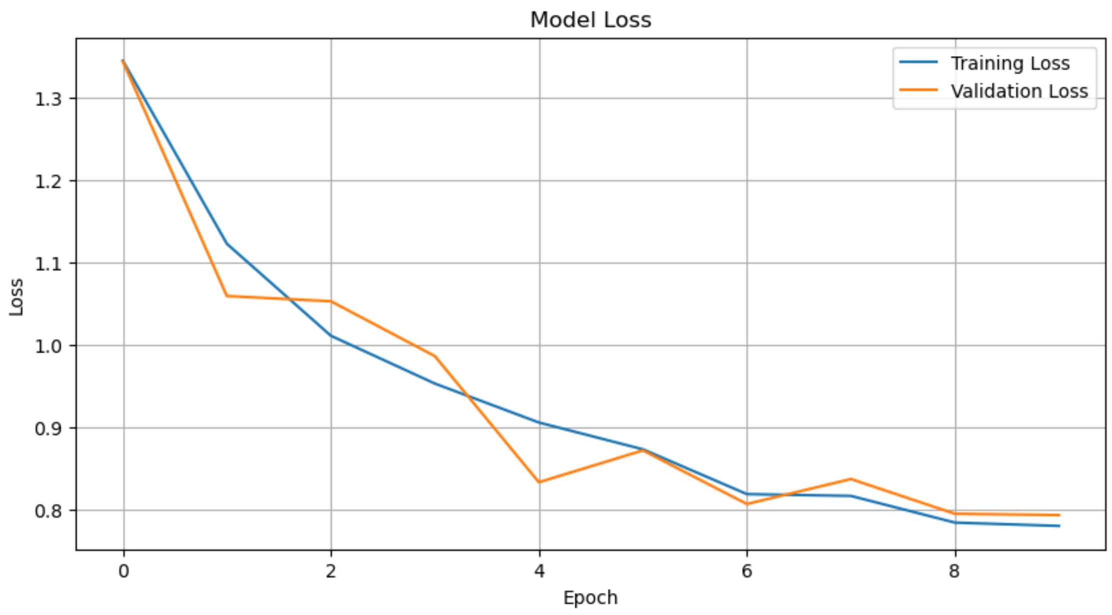
We train the model using the training data, and validate it using the validation data to monitor its performance and avoid overfitting.

[ ]:
```
# Train the model
history = model.fit(train_generator, validation_data=validation_generator,␣
 ↪epochs=10)
```

[ ]:
```
```

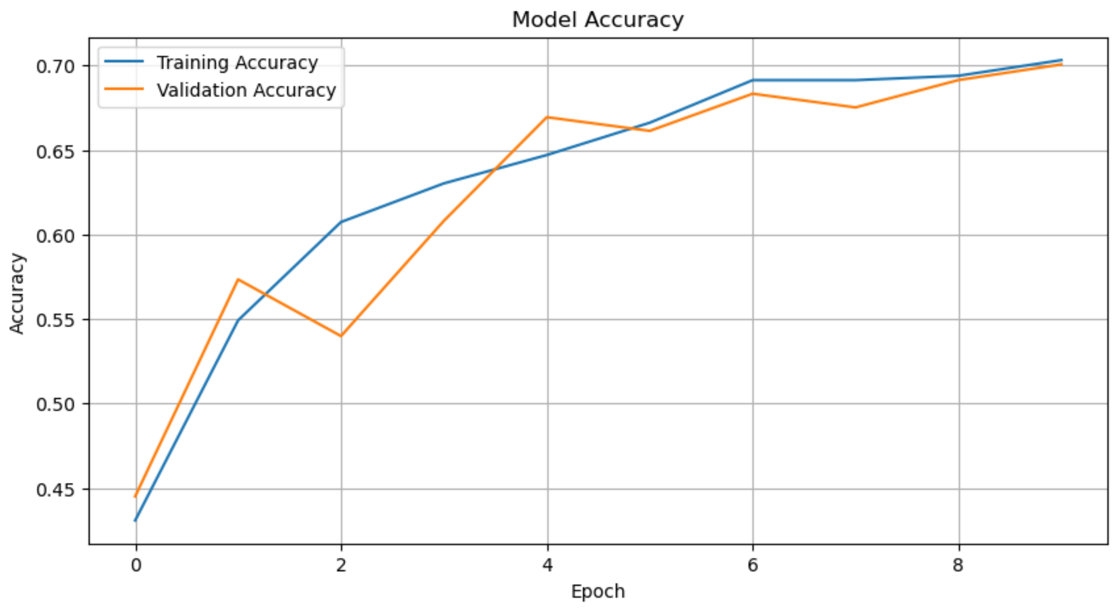
1.1.6 1.5: Plot train/valiation loss and accuracy

[20]:
```
import matplotlib.pyplot as plt

# Plot the training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(history['loss'], label='Training Loss')
plt.plot(history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.grid()
plt.show()
```



```
[ ]:
```

```
[21]: # Plot the training and validation accuracy
plt.figure(figsize=(10, 5))
plt.plot(history['accuracy'], label='Training Accuracy')
plt.plot(history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.grid()
plt.show()
```



```
[ ]:
```

1.1.7 1.6: Comparing predicted classes and actual classes

1.6.1 Extract class names

```
[24]: import os
import numpy as np

'''
Given an data_path that is the folder that contains the folder names, and the␣
 ↪folder name are the class names
return the list of class names, stored in np.array (not a list)
'''
def get_classes(data_path):
    CLASSES = [name for name in os.listdir(data_path) if os.path.isdir(os.path.
 ↪join(data_path, name))]
    CLASSES = np.array(CLASSES)
    return (CLASSES)

CLASSES = get_classes(train_dir)
print(CLASSES)
```

```
['daisy' 'dandelion' 'rose' 'sunflower' 'tulip']
```

1.1.8 1.6.2 Predict the first batch of validation

[26]:
```
# Access the first batch of images (x_test) and labels (y_test)
x_test, y_test = validation_generator[0]  # Access the first batch
print("The first batch shapes: ", x_test.shape, y_test.shape)
```

```
The first batch shapes:  (32, 150, 150, 3) (32, 5)
```

[27]:
```
preds = model.predict(x_test)
preds= CLASSES[np.argmax(preds, axis=-1)]
actual = CLASSES[np.argmax(y_test, axis=-1)]

print("The first 10 predicted classes: ", preds[:10])
print("The first 10 actual classes: ", actual[:10])
```

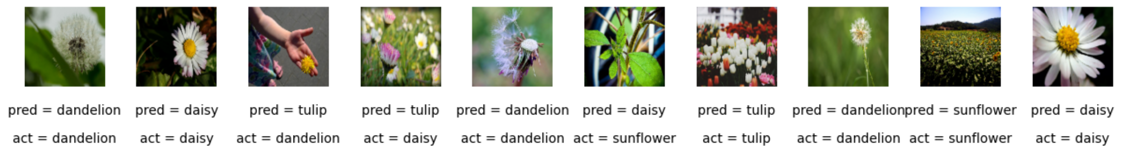
```
1/1              0s 141ms/step
The first 10 predicted classes:  ['dandelion' 'daisy' 'tulip' 'tulip'
'dandelion' 'daisy' 'tulip'
 'dandelion' 'sunflower' 'daisy']
The first 10 actual classes:  ['dandelion' 'daisy' 'dandelion' 'daisy'
'dandelion' 'sunflower' 'tulip'
 'dandelion' 'sunflower' 'daisy']
```

1.1.9 Task 2: Compare predicted and actual class

- Understand the following function
- Call the function to visualize the first ten flower images (stored in x_test), and display the predicted labels (stored in preds) and the actual labels (stored in actual) (See the above cell)

[28]:
```
def pred_act_visualization(num, x_test, preds, actuals):
    """
    Visualize the predictions and actual labels of a set of images.

    Parameters:
    - num: Number of images to display.
    - x_test: Array of test images.
    - preds: Array of predicted labels.
    - actuals: Array of actual labels.
    """

    fig = plt.figure(figsize=(15, 3))
    fig.subplots_adjust(hspace=0.4, wspace=0.4)

    for i in range(num):
        img = x_test[i]
        ax = fig.add_subplot(1, num, i + 1)
        ax.axis("off")
        ax.text(
            0.5,
            -0.35,
            "pred = " + str(preds[i]),
            fontsize=10,
            ha="center",
            transform=ax.transAxes,
        )
        ax.text(
            0.5,
            -0.7,
            "act = " + str(actuals[i]),
            fontsize=10,
            ha="center",
            transform=ax.transAxes,
        )
        ax.imshow(img)
```

[29]:
```
##########################################################################
# CaLL the the above defined function to visualize the first ten flowers,
# display predicted classes and actual classes (your source code below)
# import necessary libaries if needed
##########################################################################
```



1.2 2. Visualizing Intermediate Representations

[31]:
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

[ ]:
```
```

1.2.1 Task 3A: Visualizing Intermediate Representations

- Depending on the model (either Functional API or Sequential) you build, you may **copy the implemenation of the following functions** from the given example notebooks.
- Be sure to understand each of the copied functions

[ ]:
```
```

[32]:
```
# Define visualization model
def define_visualization_model(model):
    pass
```

[33]:
```
# Preprocess an input image
def preprocess_image(x):
    pass
```

[34]:
```
# Generate feature maps
def generate_feature_maps(visualization_model, x):
    pass
```

[35]:
```
# Normalize and scale feature map slice
def normalize_feature_map(feature_map_slice):
    pass
```

[36]:
```
# Visualize feature maps
def visualize_feature_maps(model, successive_feature_maps):
    pass
```

[38]:
```
# Main process
def main_visualization_process(model, x):
    pass
```

[39]:
```
from tensorflow.keras.preprocessing.image import load_img, img_to_array
import os

def prepare_image(data_dir, image_file, target_size=TARGET_SIZE):
    """
    Prepares an image from a given directory by loading it, converting it to an␣
 ↪array,
    normalizing it, and reshaping it to match the expected input format of a␣
 ↪model.

    Parameters:
    - data_dir: The directory where the image is stored.
    - image_file: The name of the image file (e.g., 'image_file.png').
    - target_size: The target size for the image (default is (150, 150)).

    Returns:
    - x: Numpy array with shape (1, target_size[0], target_size[1], 3).
    """
    # Construct the full image path
    img_path = os.path.join(data_dir, image_file)
    print(f"Image path: {img_path}")

    # Load the image with the specified target size
    img = load_img(img_path, target_size=target_size)

    # Convert the image to a numpy array
    x = img_to_array(img)

    # Reshape the array to add an extra dimension (batch size of 1)
    x = x.reshape((1,) + x.shape)  # Shape becomes (1, 150, 150, 3)

    return x
```

[ ]:
```

```

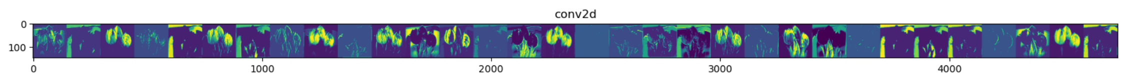
1.2.2 Task 3B: Visualizing Intermediate Representations

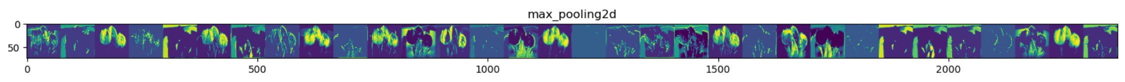
- Display intermediate representations for the following three images:
  - validation/tulip/tulip_000524.png
  - validation/rose/rose_000231.png
  - validation/daisy/daisy_000182.png
- For each image, you should call the two functions:
  - prepare_image
  - main_visualization_process

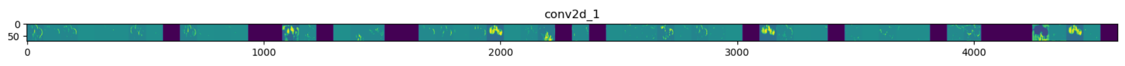
[40]:
```
######################################################################
# test case 1: tulip image (your source code below)
######################################################################
```

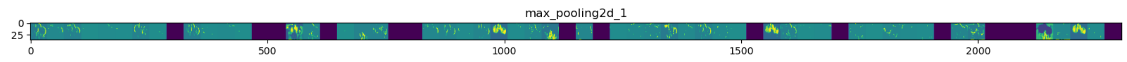
```
Image path: ../flowers_train_validation/validation\tulip\tulip_000524.png
1/1              0s 132ms/step
```

```
C:\Users\dche\AppData\Local\Temp\ipykernel_11288\3769701417.py:16:
RuntimeWarning: invalid value encountered in divide
  x /= x.std()   # Divide by the standard deviation
C:\Users\dche\AppData\Local\Temp\ipykernel_11288\3769701417.py:19:
RuntimeWarning: invalid value encountered in cast
  x = np.clip(x, 0, 255).astype('uint8')
```



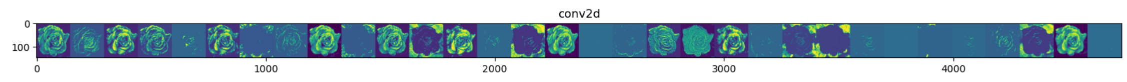


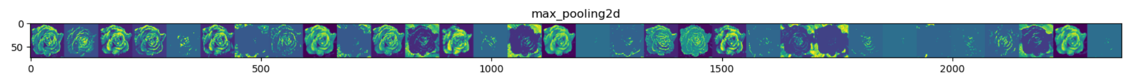


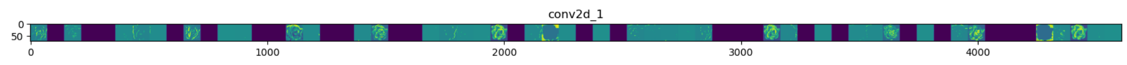


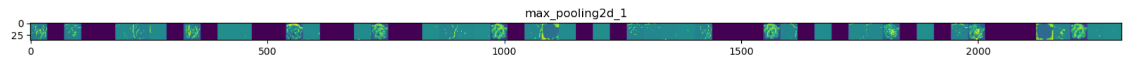
[41]:
```
######################################################################
# test case 2: rose image (your source code below)
######################################################################
```

```
Image path: ../flowers_train_validation/validation\rose\rose_000231.png
1/1              0s 130ms/step

C:\Users\dche\AppData\Local\Temp\ipykernel_11288\3769701417.py:16:
RuntimeWarning: invalid value encountered in divide
  x /= x.std()   # Divide by the standard deviation
C:\Users\dche\AppData\Local\Temp\ipykernel_11288\3769701417.py:19:
RuntimeWarning: invalid value encountered in cast
  x = np.clip(x, 0, 255).astype('uint8')
```





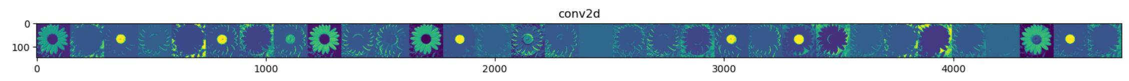


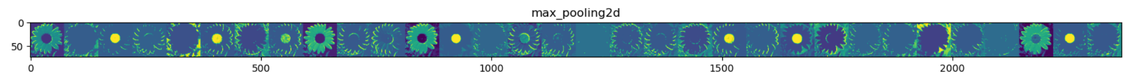


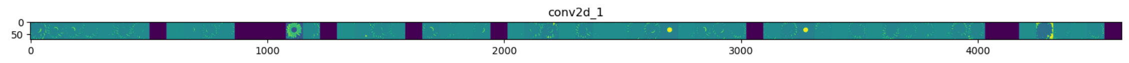
[42]:
```
######################################################################
# test case 3: daisy image (your source code below)
######################################################################
```

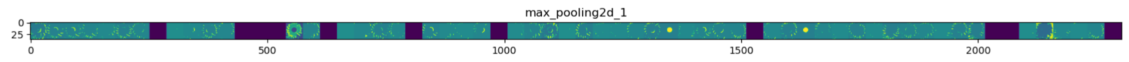
```
Image path: ../flowers_train_validation/validation\daisy\daisy_000182.png
1/1              0s 125ms/step
```

```
C:\Users\dche\AppData\Local\Temp\ipykernel_11288\3769701417.py:16:
RuntimeWarning: invalid value encountered in divide
  x /= x.std()   # Divide by the standard deviation
C:\Users\dche\AppData\Local\Temp\ipykernel_11288\3769701417.py:19:
RuntimeWarning: invalid value encountered in cast
  x = np.clip(x, 0, 255).astype('uint8')
```









```
[ ]:
```

1.3 3. Transfer Learning

```
[ ]: from tensorflow.keras.applications import VGG16
     from tensorflow.keras.layers import Flatten, Dense, Dropout,␣
      ↪GlobalAveragePooling2D
     from tensorflow.keras.models import Model
     from tensorflow.keras.optimizers import Adam
```

```
[ ]:
```

1.3.1 Task 4: Building the Transfer Learning based CNN Model

You will use the functional API to build transfer learning based CNN model as follows:
- Load the VGG16 model, exclude the top (fully connected) layers - Freeze the convolutional layers of the VGG16 model - Add new top layers for flower classification - GlobalAveragePooling2D() layer - Dense layer with 'relu' activation - Drop layer - Softmax layer - Display model summary

```
[ ]: ######################################################################
     # Build your model (your source code below)
     ######################################################################
```

```
[45]: transfer_model.summary()
```

```
Model: "functional_12"


 Layer (type)                      Output Shape                 Param #

 input_layer_2 (InputLayer)        (None, 150, 150, 3)                0

 block1_conv1 (Conv2D)             (None, 150, 150, 64)           1,792

 block1_conv2 (Conv2D)             (None, 150, 150, 64)          36,928

 block1_pool (MaxPooling2D)        (None, 75, 75, 64)                 0

 block2_conv1 (Conv2D)             (None, 75, 75, 128)           73,856

 block2_conv2 (Conv2D)             (None, 75, 75, 128)          147,584

 block2_pool (MaxPooling2D)        (None, 37, 37, 128)                0

 block3_conv1 (Conv2D)             (None, 37, 37, 256)          295,168

 block3_conv2 (Conv2D)             (None, 37, 37, 256)          590,080

 block3_conv3 (Conv2D)             (None, 37, 37, 256)          590,080

 block3_pool (MaxPooling2D)        (None, 18, 18, 256)                0

 block4_conv1 (Conv2D)             (None, 18, 18, 512)        1,180,160

 block4_conv2 (Conv2D)             (None, 18, 18, 512)        2,359,808

 block4_conv3 (Conv2D)             (None, 18, 18, 512)        2,359,808

 block4_pool (MaxPooling2D)        (None, 9, 9, 512)                  0

 block5_conv1 (Conv2D)             (None, 9, 9, 512)          2,359,808

 block5_conv2 (Conv2D)             (None, 9, 9, 512)          2,359,808

 block5_conv3 (Conv2D)             (None, 9, 9, 512)          2,359,808
```

```
block5_pool (MaxPooling2D)          (None, 4, 4, 512)                  0

global_average_pooling2d            (None, 512)                        0
(GlobalAveragePooling2D)

dense_4 (Dense)                     (None, 1024)                 525,312

dropout (Dropout)                   (None, 1024)                       0

dense_5 (Dense)                     (None, 5)                      5,125
```

```
Total params: 15,245,125 (58.16 MB)
```

```
Trainable params: 530,437 (2.02 MB)
```

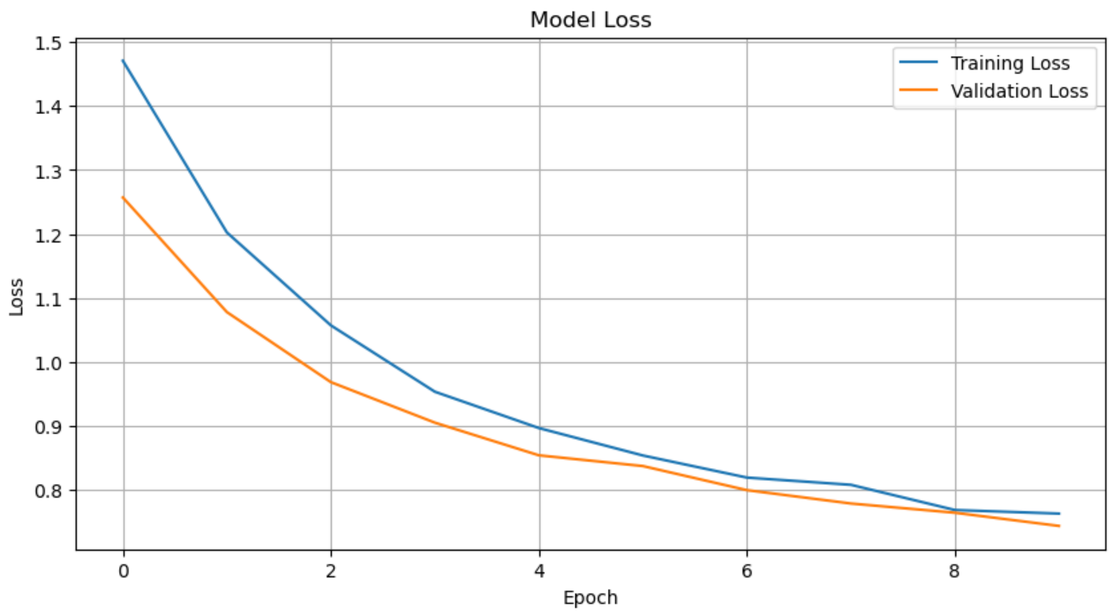
```
Non-trainable params: 14,714,688 (56.13 MB)
```

[46]:
```
# Compile the model
transfer_model.compile(optimizer=Adam(learning_rate=0.0001),␣
 ↪loss='categorical_crossentropy', metrics=['accuracy'])
```

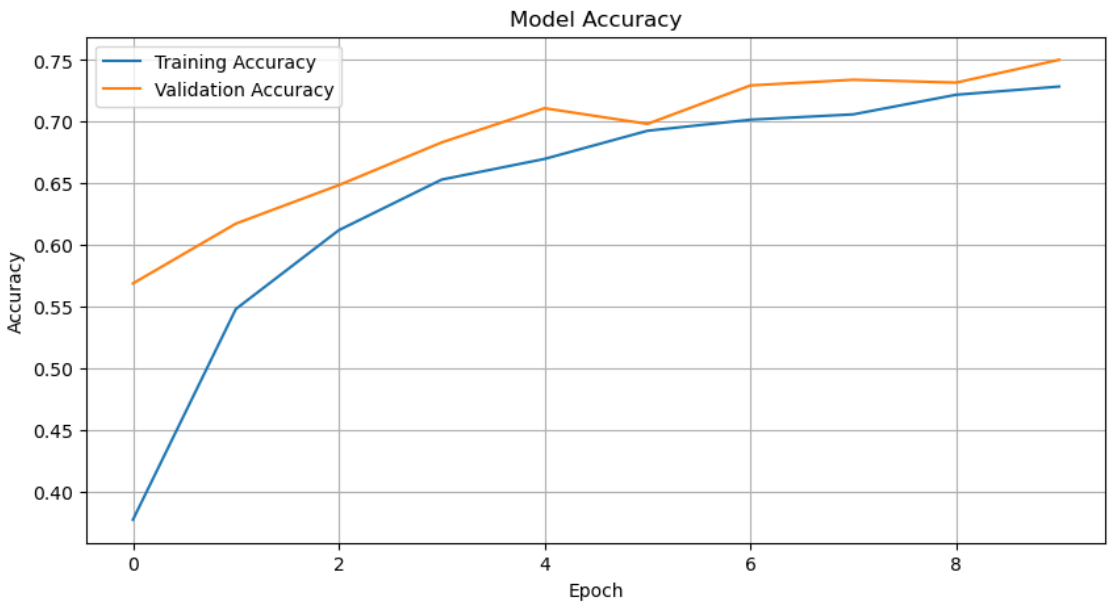
[ ]:
```

```

[ ]:
```
# Train the model using your dataset
# Warning: It might take a couple of hours to run your own CPU

transfer_history = transfer_model.fit(train_generator,␣
 ↪validation_data=validation_generator, epochs=10)
```

[49]:
```
import matplotlib.pyplot as plt

# Plot the training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(transfer_history['loss'], label='Training Loss')
plt.plot(transfer_history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.grid()
plt.show()
```



[50]:
```
# Plot the training and validation accuracy
plt.figure(figsize=(10, 5))
plt.plot(transfer_history['accuracy'], label='Training Accuracy')
plt.plot(transfer_history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.grid()
plt.show()
```


Model Accuracy
Training Accuracy
Validation Accuracy
Accuracy
Epoch
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0
2
4
6
8