Image Classification

Kelly Trinh
Dr. Karen Mazidi
CS 4375.004

In this assignment, I performed image classification on a rice dataset. In the dataset, there are images of 5 types of rice, and the goal was to classify each image of rice into the correct type. I used a Kaggle Notebook to do this assignment as importing the images was much easier. An important thng to note about this assignment was that for my models, I only used 5 epochs. Kaggle Notebook was quite slow in compiling the models, so I settled with using a lesser amount of epochs. Also, it's important to note that I didn't include any analysis on the evaluations of my models until the end of the assignment, where I combined all of my analyses.

Description of Data

This data set contains images of 5 different types of rice. The rice types are separated into five different folders, and models that I create in this assignment should be able to predict which type of rice is portrayed in the image (out of the five different rice types).

Below are imports that I used for this assignment

```
# Basic Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
sns.set_style('darkgrid')
# import cv2
import itertools

# Imports for importing the data
import os
import pathlib

# Imports for TF and keras and some plotting
import tensorflow as tf
import keras
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Activation,Dropout
from keras.models import Model, Sequential
# from keras.optimizers import Adam
from keras.metrics import categorical_crossentropy
from sklearn.metrics import confusion_matrix,classification_report
from sklearn.utils import shuffle
import imageio
import matplotlib.image as img
from tensorflow.keras.applications import imagenet_utils
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
```

Importing the Dataset

To import the dataset, I looked at how other Kaggle Notebooks were importing the dataset. This was the method I found the most efficient. I used the Keras ImageDataGenerator object to rescale and split the data into an 80/20 Train-Validation split.

Below, I instanstiated the path of the dataset, and I created a list for each type of rice in the dataset.

```
path = pathlib.Path('/kaggle/input/rice-image-dataset/Rice_Image_Dataset/')

arborio = list(path.glob('Arborio/*'))[:1000]
basmati = list(path.glob('Basmati/*'))[:1000]
ipsala = list(path.glob('Ipsala/*'))[:1000]
jasmine = list(path.glob('Jasmine/*'))[:1000]
karacadag = list(path.glob('Karacadag/*'))[:1000]
```

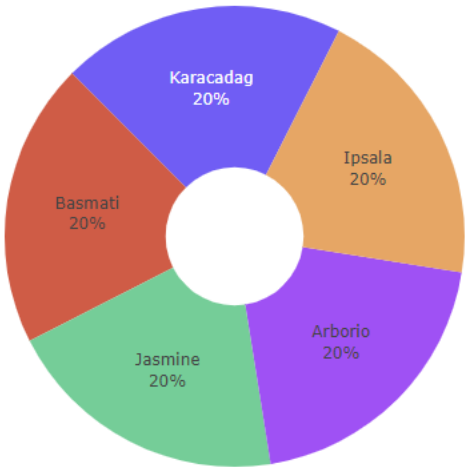
Then, I created a data dictionary for each of the lists of rice types, as well as the labels for each type of rice. I also initialized some constants that I would be using throughout the notebook.

```
data = { 'arborio' : arborio, 'basmati' : basmati, 'ipsala' : ipsala, 'jasmine' : jasmine, 'karacadag': karacadag }

labels = { 0: "Arborio", 1: "Basmati", 2: "Ipsala", 3:"Jasmine", 4:"Karacadag" }


batch_size = 128
num_classes = len(data)
epochs = 5
img_size = (224, 224)
```

Below, I am exploring the dataset by looking at the distribution of the target classes. I did this by creating two arrays, one of which holds the paths of the images, and the other which holds the labels of the images. I added the images and the labels to these arrays, and I was able to see how many of each were in the array by outputting the plot. Due to the way I imported the data, I was inspired to view the data distribution in this way by looking at how other Kaggle datasets viewed the distribution.
As we can see, there is an even distribution of all the rice types in the data set, with each of the images being 20% of the set. Since the set has 75,000 images, this means there are 15,000 images for each type of rice.

```
image_count = len(list(path.glob('*/*.jpg')))
print(f'Total images: {image_count}')
print(f'Total number of classes: {len(labels)}')

all_paths = []
all_labels = []
name_labels = ['Arborio', 'Basmati', 'Ipsala', 'Jasmine', 'Karacadag']
data_path = '/kaggle/input/rice-image-dataset/Rice_Image_Dataset/'

for label in name_labels:
    for image_path in os.listdir(data_path+label):
        all_paths.append(data_path+label+'/'+image_path)
        all_labels.append(label)


all_paths, all_labels = shuffle(all_paths, all_labels)

     Total images: 75000
     Total number of classes: 5


values = [len([x for x in all_labels if x==label]) for label in name_labels]
graph = go.Figure(data=[go.Pie(labels=name_labels,
                values=values,
                rotation=-45,
                hole=.3,
                textinfo='label+percent')])
graph.update_layout(showlegend=False)
graph.show()
```

I inserted an image so that you could view the graph. I had problems with displaying the graph by itself through the code.



Preparing the Dataset

Below is where I prepared the dataset by using the ImageDataGeneartor object that I mentioned above. I split the data into 80/20 Test-Validation sets using the ImageDataGenrator object, and I rescaled the images by 255.0. The class mode is categorical, as I am performing classification on the images.

```
generate_data = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255.0, validation_split=0.2)
train_x = generate_data.flow_from_directory(path,
                                            subset='training',
                                            shuffle=True,
                                            class_mode='categorical',
                                            batch_size = batch_size,
                                            target_size = img_size)

test_x = generate_data.flow_from_directory(path,
                                           subset='validation',
                                           shuffle=False,
                                           class_mode='categorical',
                                           batch_size = batch_size,
                                           target_size = img_size)
```

```
    Found 60000 images belonging to 5 classes.
    Found 15000 images belonging to 5 classes.
```

Create the First Model

Below, I created a sequential model of the data. There are two dense layers in the model, and the activation function I used was 'softmax' because there are multiple categories of rice that it could be. I ouputted the summary of the model, and then I compiled it. I only used 5 epochs for the model. I tried attempting the model with 20 epochs, but it was taking a very long time because Kaggle is not very efficient in terms of analyzing images, so I settled with 5.

```
model1 = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(224, 224, 3)),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(num_classes, activation='softmax'),
])
```

```
model1.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 150528)            0

 dense (Dense)               (None, 512)               77070848

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 512)               262656

 dropout_1 (Dropout)         (None, 512)               0

 dense_2 (Dense)             (None, 5)                 2565

=================================================================
Total params: 77,336,069
Trainable params: 77,336,069
Non-trainable params: 0
_________________________________________________________________
```

```
model1.compile(loss='categorical_crossentropy',
             optimizer='rmsprop',
             metrics=['accuracy'])

history1 = model1.fit(train_x,
                      epochs=epochs,
                      batch_size=batch_size,
                      verbose=1,
                      validation_data=test_x)
```

```
Epoch 1/5
469/469 [==============================] - 668s 1s/step - loss: 1.5586 - accuracy: 0.8549 - val_loss: 0.1949 - val_accuracy: 0.9297
Epoch 2/5
469/469 [==============================] - 288s 615ms/step - loss: 0.1846 - accuracy: 0.9406 - val_loss: 0.0852 - val_accuracy: 0.9693
Epoch 3/5
469/469 [==============================] - 273s 582ms/step - loss: 0.1344 - accuracy: 0.9561 - val_loss: 0.1435 - val_accuracy: 0.9588
Epoch 4/5
469/469 [==============================] - 273s 581ms/step - loss: 0.1191 - accuracy: 0.9625 - val_loss: 0.0965 - val_accuracy: 0.9707
Epoch 5/5
469/469 [==============================] - 274s 584ms/step - loss: 0.1068 - accuracy: 0.9664 - val_loss: 0.0893 - val_accuracy: 0.9764
```

Evaluate the Model

I evaluated the sequential model below. The loss and the accuracy of the model predicting on the test set are outputted. I also outputted the graph of the model's accuracy on the train set and the test set.

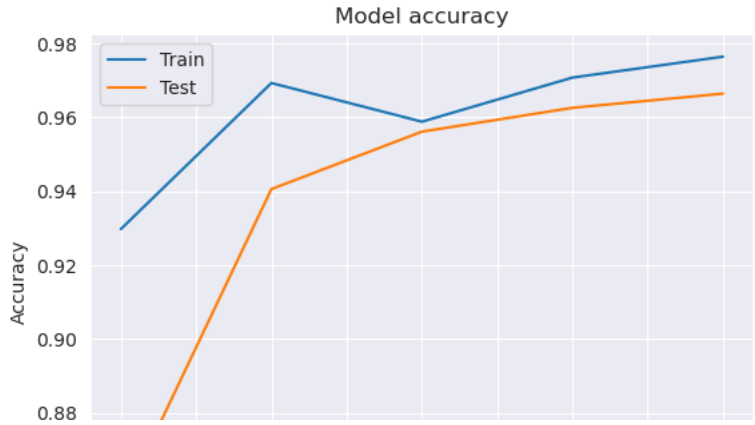
```
score1 = model1.evaluate(test_x, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.08927176147699356
Test accuracy: 0.9764000177383423
```

```
history1.history.keys()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
# Plot training & validation accuracy values
plt.plot(history1.history['val_accuracy'])
plt.plot(history1.history['accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



CNN Model

To extend on the previous model, I created a CNN model on the dataset. The difference between this model and the previous sequential model are the layers within it. In this model, there are more layers, such as:

- Conv2D - which allows a filter to go slide over the data and perform a convolution function
- MaxPooling - which reduces the dimensionality of the images

Although the sequential model performed well, I wanted to see if a CNN model would produce better accuracies. Similar to the sequential model, below, I created the CNN model and outputted the summary. Then, I compiled the model. It's important to note that I used a different optimizer for this model in comparison to the sequential model, so that may have altered the results.

```
model2 = tf.keras.models.Sequential(
    [
        tf.keras.Input(shape=(224, 224, 3)),
        tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(num_classes, activation="softmax"),
    ]
)
```

```
model2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 222, 222, 32)      896

 max_pooling2d (MaxPooling2D  (None, 111, 111, 32)     0
 )

 conv2d_1 (Conv2D)           (None, 109, 109, 64)      18496

 max_pooling2d_1 (MaxPooling  (None, 54, 54, 64)       0
 2D)

 flatten_1 (Flatten)         (None, 186624)            0

 dropout_2 (Dropout)         (None, 186624)            0

 dense_3 (Dense)             (None, 5)                 933125

=================================================================
Total params: 952,517
Trainable params: 952,517
Non-trainable params: 0
_________________________________________________________________
```

```
model2.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

```
history2 = model2.fit(train_x,
                      batch_size=batch_size,
                      epochs=epochs,
                      verbose=1,
                      validation_data=test_x)
```

```
Epoch 1/5
469/469 [==============================] - 300s 618ms/step - loss: 0.1329 - accuracy: 0.9567 - val_loss: 0.0435 - val_accuracy: 0.9863
Epoch 2/5
469/469 [==============================] - 292s 622ms/step - loss: 0.0466 - accuracy: 0.9848 - val_loss: 0.0221 - val_accuracy: 0.9935
Epoch 3/5
469/469 [==============================] - 291s 620ms/step - loss: 0.0319 - accuracy: 0.9898 - val_loss: 0.0193 - val_accuracy: 0.9941
Epoch 4/5
469/469 [==============================] - 292s 623ms/step - loss: 0.0430 - accuracy: 0.9860 - val_loss: 0.0265 - val_accuracy: 0.9927
Epoch 5/5
469/469 [==============================] - 288s 614ms/step - loss: 0.0362 - accuracy: 0.9880 - val_loss: 0.0333 - val_accuracy: 0.9893
```

Evaluate the Model

I evaluated the CNN model below. Similar to the sequential model, I outputted the loss and the accuracy of the CNN model. The graph of the CNN model is also shown, and it includes the model's accuracy on the train and validation sets.

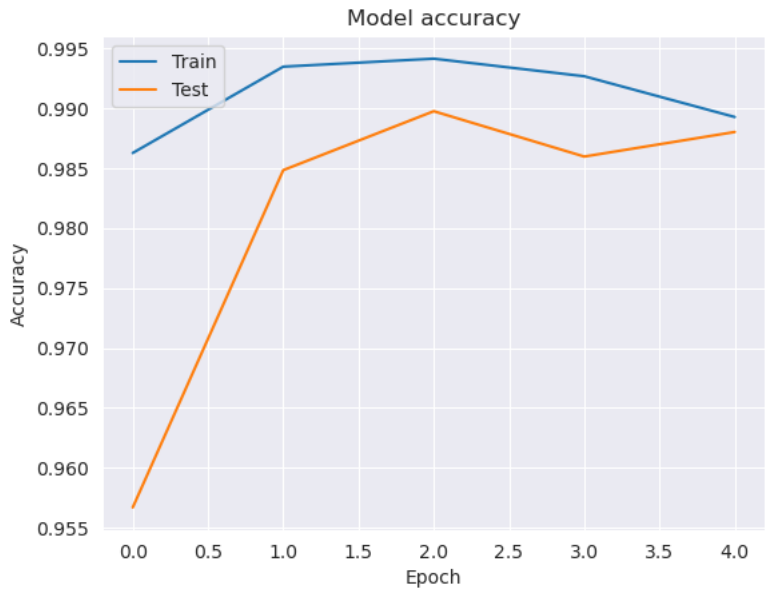
```
score2 = model2.evaluate(test_x, verbose=0)
print('Test loss:', score2[0])
print('Test accuracy:', score2[1])
```

```
Test loss: 0.033269692212343216
Test accuracy: 0.9892666935920715
```

```
history2.history.keys()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
# Plot training & validation accuracy values
plt.plot(history2.history['val_accuracy'])
plt.plot(history2.history['accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



Pre-trained Model and Transfer Learning

For my last model, I read about using a pre-trained model and using transfer learning. For my pre-trained model, I used the MobileNetV2 model. For this pre-trained model, I needed to pick a layer for the model to depend on. I used the last layer, which is also referred to the "bottleneck" layer. Below, I created the base layer, which is the pre-trained model. Then, I outputted the summary of that model. Then, I extracted the last

layer from the pre-trained model. Finally, I created the transfer learning model using the base model relying on the last layer for training. I outputted the summaries for both the pre-trained model and the transfer learning model below as well.

```
preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input
rescale = tf.keras.layers.Rescaling(1./127.5, offset=-1)


base_model3 = tf.keras.applications.MobileNetV2(input_shape=(224, 224, 3),
                                                include_top=False,
                                                weights='imagenet')

base_model3.trainable = False

base_model3.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_t
9406464/9406464 [==============================] - 0s 0us/step
Model: "mobilenetv2_1.00_224"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 224, 224, 3  0           []
                                )]

 Conv1 (Conv2D)                 (None, 112, 112, 32  864         ['input_1[0][0]']
                                )

 bn_Conv1 (BatchNormalization)  (None, 112, 112, 32  128         ['Conv1[0][0]']
                                )

 Conv1_relu (ReLU)              (None, 112, 112, 32  0           ['bn_Conv1[0][0]']
                                )

 expanded_conv_depthwise (Depth  (None, 112, 112, 32  288        ['Conv1_relu[0][0]']
 wiseConv2D)                    )

 expanded_conv_depthwise_BN (Ba  (None, 112, 112, 32  128        ['expanded_conv_depthwise[0][0]']
 tchNormalization)              )

 expanded_conv_depthwise_relu (  (None, 112, 112, 32  0          ['expanded_conv_depthwise_BN[0][0
 ReLU)                          )                                 ]']

 expanded_conv_project (Conv2D)  (None, 112, 112, 16  512        ['expanded_conv_depthwise_relu[0]
                                )                                 [0]']

 expanded_conv_project_BN (Batc  (None, 112, 112, 16  64         ['expanded_conv_project[0][0]']
 hNormalization)                )

 block_1_expand (Conv2D)        (None, 112, 112, 96  1536        ['expanded_conv_project_BN[0][0]'
                                )                                 ]

 block_1_expand_BN (BatchNormal  (None, 112, 112, 96  384        ['block_1_expand[0][0]']
 ization)                       )

 block_1_expand_relu (ReLU)     (None, 112, 112, 96  0           ['block_1_expand_BN[0][0]']
                                )

 block_1_pad (ZeroPadding2D)    (None, 113, 113, 96  0           ['block_1_expand_relu[0][0]']
                                )

 block_1_depthwise (DepthwiseCo  (None, 56, 56, 96)  864         ['block_1_pad[0][0]']
 nv2D)

 block_1_depthwise_BN (BatchNor  (None, 56, 56, 96)  384         ['block_1_depthwise[0][0]']
 malization)

 block_1_depthwise_relu (ReLU)  (None, 56, 56, 96)   0           ['block_1_depthwise_BN[0][0]']

 block_1_project (Conv2D)       (None, 56, 56, 24)   2304        ['block_1_depthwise_relu[0][0]']

 block_1_project_BN (BatchNorma  (None, 56, 56, 24)  96          ['block_1_project[0][0]']
 lization)
```

Use the last layer to depend on for the transfer learning

```
last_layer = base_model3.output

x = Flatten()(last_layer)
pred = Dense(5, activation='softmax', name='softmax')(x)
```

```
transfer_model3 = Model(inputs=base_model3.input, outputs=pred)


transfer_model3.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 224, 224, 3  0           []
                                )]

 Conv1 (Conv2D)                 (None, 112, 112, 32  864         ['input_1[0][0]']
                                )

 bn_Conv1 (BatchNormalization)  (None, 112, 112, 32  128         ['Conv1[0][0]']
                                )

 Conv1_relu (ReLU)              (None, 112, 112, 32  0           ['bn_Conv1[0][0]']
                                )

 expanded_conv_depthwise (Depth  (None, 112, 112, 32  288        ['Conv1_relu[0][0]']
 wiseConv2D)                    )

 expanded_conv_depthwise_BN (Ba  (None, 112, 112, 32  128        ['expanded_conv_depthwise[0][0]']
 tchNormalization)              )

 expanded_conv_depthwise_relu (  (None, 112, 112, 32  0          ['expanded_conv_depthwise_BN[0][0
 ReLU)                          )                                 ]']

 expanded_conv_project (Conv2D)  (None, 112, 112, 16  512        ['expanded_conv_depthwise_relu[0]
                                )                                 [0]']

 expanded_conv_project_BN (Batc  (None, 112, 112, 16  64         ['expanded_conv_project[0][0]']
 hNormalization)                )

 block_1_expand (Conv2D)        (None, 112, 112, 96  1536        ['expanded_conv_project_BN[0][0]'
                                )                                 ]

 block_1_expand_BN (BatchNormal  (None, 112, 112, 96  384        ['block_1_expand[0][0]']
 ization)                       )

 block_1_expand_relu (ReLU)     (None, 112, 112, 96  0           ['block_1_expand_BN[0][0]']
                                )

 block_1_pad (ZeroPadding2D)    (None, 113, 113, 96  0           ['block_1_expand_relu[0][0]']
                                )

 block_1_depthwise (DepthwiseCo  (None, 56, 56, 96)  864         ['block_1_pad[0][0]']
 nv2D)

 block_1_depthwise_BN (BatchNor  (None, 56, 56, 96)  384         ['block_1_depthwise[0][0]']
 malization)

 block_1_depthwise_relu (ReLU)  (None, 56, 56, 96)   0           ['block_1_depthwise_BN[0][0]']

 block_1_project (Conv2D)       (None, 56, 56, 24)   2304        ['block_1_depthwise_relu[0][0]']

 block_1_project_BN (BatchNorma  (None, 56, 56, 24)  96          ['block_1_project[0][0]']
 lization)

 block_2_expand (Conv2D)        (None, 56, 56, 144)  3456        ['block_1_project_BN[0][0]']
```

```
base_learning_rate = 0.0001

transfer_model3.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=base_learning_rate),
              loss=tf.keras.losses.categorical_crossentropy,
              metrics=['accuracy'])

history_transfer_model3 = transfer_model3.fit(train_x,
                                              epochs=epochs,
                                              verbose=1,
                                              validation_data=test_x)
```

```
Epoch 1/5
469/469 [==============================] - 407s 859ms/step - loss: 0.0421 - accuracy: 0.9863 - val_loss: 0.0293 - val_accuracy: 0.9902
Epoch 2/5
469/469 [==============================] - 246s 524ms/step - loss: 0.0208 - accuracy: 0.9933 - val_loss: 0.0319 - val_accuracy: 0.9913
Epoch 3/5
469/469 [==============================] - 244s 521ms/step - loss: 0.0135 - accuracy: 0.9955 - val_loss: 0.0268 - val_accuracy: 0.9923
Epoch 4/5
```

```
469/469 [==============================] - 249s 532ms/step - loss: 0.0103 - accuracy: 0.9967 - val_loss: 0.0264 - val_accuracy: 0.9921
Epoch 5/5
469/469 [==============================] - 250s 532ms/step - loss: 0.0090 - accuracy: 0.9968 - val_loss: 0.0253 - val_accuracy: 0.9928
```

Evaluate the Model

Below, I evaluated the transfer learning model. Similar to the previous two models, I outputted both the loss and the accuracy of the model, and I includled a graph which displays the model's accuracy on the train and validation sets.

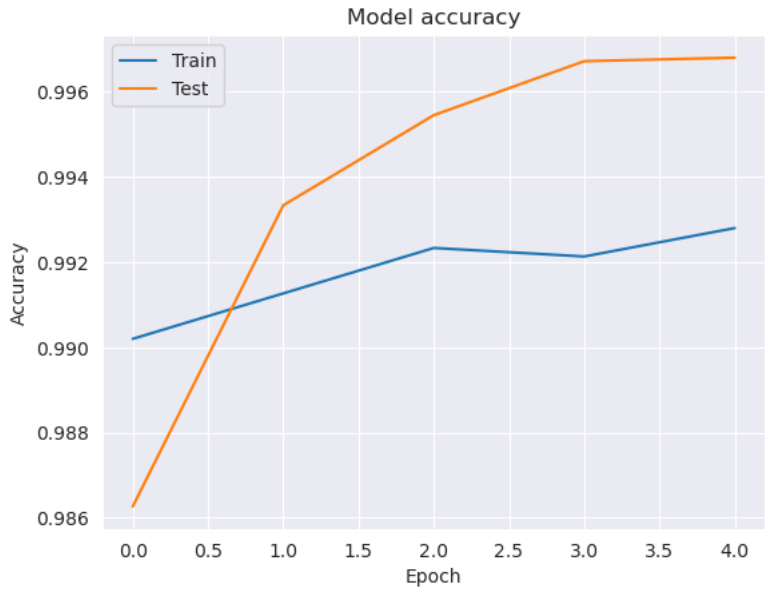
```
score3 = transfer_model3.evaluate(test_x, verbose=0)
print('Test loss:', score3[0])
print('Test accuracy:', score3[1])
```

```
Test loss: 0.0253437627106905
Test accuracy: 0.9927999973297119
```

```
history_transfer_model3.history.keys()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
# Plot training & validation accuracy values
plt.plot(history_transfer_model3.history['val_accuracy'])
plt.plot(history_transfer_model3.history['accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



Analysis of Models

Sequential Model:

For the sequential model, the accuracy on the test set was 0.98. This is quite a high accuracy score, and it indicates that the model was able to accurately predict the type of rice in the image to a the correct label. Furthermore, the test loss was extremely low, which indicates that the model was a good fit for predicting a single example. The low loss means that there was not much error when the model was training on the training set, which applies into the validation set. In the graph, we can see that the training set had a higher accuracy, but there was a dip in the accuracy in the second epoch. However, the testing set consistently performed well, meaning that it had no dips.

CNN Model:

For the CNN model, the accuracy was slightly higher than the sequential model's accuracy. The CNN model's accuracy was 0.989. Furthermore, the test loss was 0.333, which was also lower than the sequential model's test loss. This means that the CNN model was a better fit for predicting the labels for each rice image. Through the test loss, we can tell that the model is a better fit because there was less error when predicting on the validation set. Also, we can tell that the model is a better fit through the accuracy, which was higher than the sequential's. Another thing to note is that, in the graph for the CNN model, the training set had a higher accuracy than the validatdion set. The test set was

not as consistent as the training set as well, meaning that there was a dip in the accuracy.

Transfer Learning:

For the transfer-learning and pre-trained model, the accuracy was the highest of the three models, which was at 0.99. The test loss of the model was also the lowest at 0.0253 compared to the two other models. This indicates that the transfer-learning model was the best for the dataset. The test loss was the lowest, meaning that there was the least amount of error in contrast to the other two models (sequential and CNN). Furthermore, when looking at the graph that I created for this model, the validation set performed much better than the training set. The testing set was also much more consistent in comparison to the training set.

Summary: From the three models, we can see that the transfer learning model was the best at predicting the type of rice in the images. However, all of the models performed extremely well, with all having an accuracy of more than 0.95 for all three models. After having chosen to do 5 epochs for each of the models, I think that this was a good choice because the testing data never performed significantly better than the training data. This means that the training data did not overfit the dataset. However, in the last model, there was a chance of overfitting because, according to the graph, the validation set did perform significantly better than the training set. It's also important to note that the first sequential model may not have performed as well because I chose a different optimizer. Overall, though, I would choose the CNN model as the best choice for this dataset because there was no chance of overfitting according to the graph, and it still had an extremely high accuracy and extremely low test loss.

References:

[1] For help with the transfer learning model: https://www.tensorflow.org/tutorials/images/transfer_learning#rescale_pixel_values
[2] For importing the data, I got help from this notebook: https://www.kaggle.com/code/ayessa/rice-classification-99-accuracy
[3] For viewing the class distributions, I got help from this notebook: https://www.kaggle.com/code/mushfirat/rice-classification-99-2-
[4] For the models, I used Dr. Mazidi's examples on her GitHub for assistance.