keras_learning-seq-cnn

December 5, 2022

1 Images & Neural Networks

The goal of this notebook is to gain expereince with image classification using Keras, a popular deep learning library.

1.1 Loading and Processing

I'm importing a simple image data set from kaggle that has quite simple images of 5 different types of rice: Arborio, Basmati, Ipsala, Jasimine, and Karacadag. They are quite plain images of a single grain of rice on a black background, the simplest identification case for our models. The model should be able to learn how to classify a given grain of rice to one of those 5 labels just based on an image.

The file is imported as a zip, unzipped, and each classifcation is in it's own subdirectory. This has a direct guide in the keras api.

[ ]:
```
# Import keras at the start!
import tensorflow as tf
from tensorflow import keras
```

[ ]:
```
# Load the zip file downloaded from kaggle straight from local memory
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving rice.zip to rice.zip
```

[ ]:
```
!unzip rice.zip
print("Hope that didn't take too long!")
```

```
Streaming output truncated to the last 5000 lines.
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (550).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5500).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5501).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5502).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5503).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5504).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5505).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5506).jpg
```

```
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9958).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9959).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (996).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9960).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9961).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9962).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9963).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9964).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9965).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9966).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9967).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9968).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9969).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (997).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9970).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9971).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9972).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9973).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9974).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9975).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9976).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9977).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9978).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9979).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (998).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9980).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9981).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9982).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9983).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9984).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9985).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9986).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9987).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9988).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9989).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (999).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9990).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9991).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9992).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9993).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9994).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9995).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9996).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9997).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9998).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9999).jpg
  inflating: Rice_Image_Dataset/Rice_Citation_Request.txt
Hope that didn't take too long!
```

Keras has a lot of great utility functions for processing functions, and thankfully I can just load the data from file directories right into a `tf.data.Dataset`. It's more like a generator for accessing the data. We can then retrieve batches of data from the dataset as we need them. I ended up using this simply because the utils function outputs a dataset.

`image_dataset_from_directory()` has a lot of default settings we are just going to use:

```
def image_dataset_from_directory(directory,
                                 labels='inferred',
                                 label_mode='int',
                                 class_names=None,
                                 color_mode='rgb',
                                 batch_size=32,
                                 image_size=(256, 256),
                                 shuffle=True,
                                 seed=None,
                                 validation_split=None,
                                 subset=None,
                                 interpolation='bilinear',
                                 follow_links=False,
                                 crop_to_aspect_ratio=False,
                                 **kwargs):
```

[ ]:
```
data = keras.utils.image_dataset_from_directory("Rice_Image_Dataset",␣
 ↪seed=1234, image_size=(128, 128))
```

```
Found 75000 files belonging to 5 classes.
```

[ ]:
```
# We then iterate through the dataset using numpy
import numpy as np
from matplotlib import pyplot as plt
data_iterator = data.as_numpy_iterator()
```

[ ]:
```
batch = data_iterator.next()
```

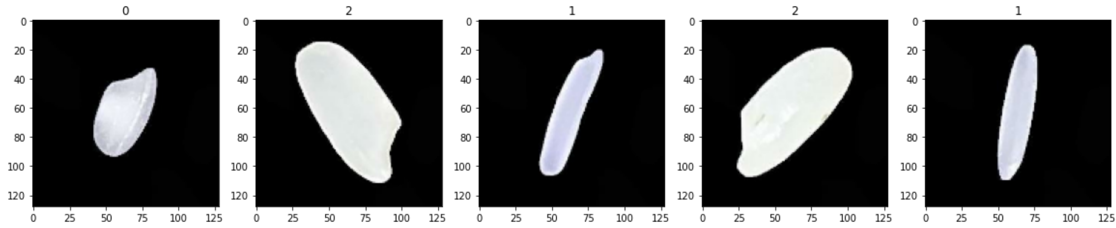
[ ]:
```
# Each batch contains
#   1: the images loaded in as numpy arrays
#   2: the label of the image
# Note the batch size was set to 32 by default
len(batch)
batch[0].shape
```

[ ]:
```
(32, 128, 128, 3)
```

1.1.1 Checking our Work

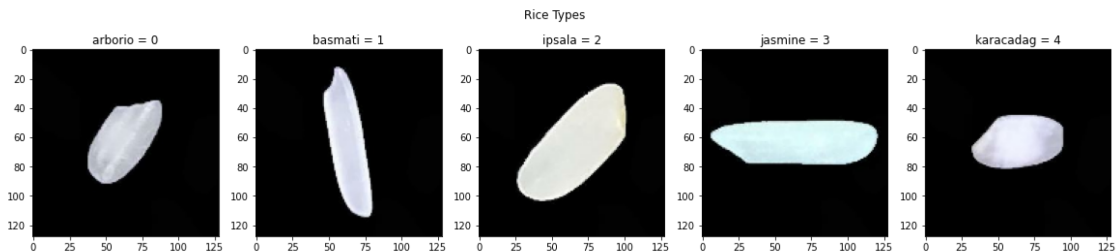
Using the matplotlib function (because I saw this in a lot of examples online, we are going to look at our rice to make sure it is loaded in properly!

[ ]:
```
fig, ax = plt.subplots(ncols=5, figsize=(20,20))
for idx, img in enumerate(batch[0][:5]):
  ax[idx].imshow(img.astype(int))
  ax[idx].title.set_text(batch[1][idx])
```



Lets create a reference chart just so we can identify the different types of rice.

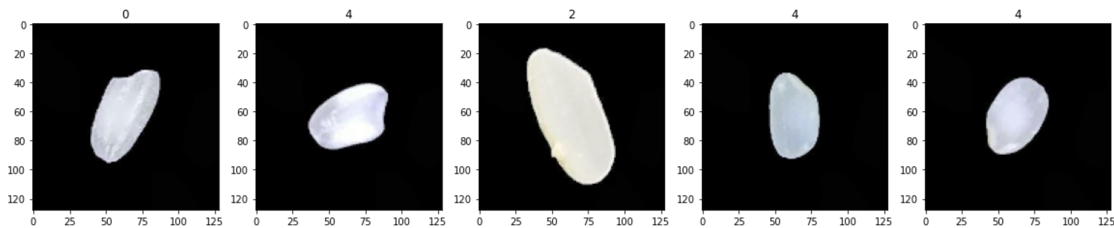
[ ]:
```
fig, ax = plt.subplots(ncols=5, figsize=(20,4))
fig.suptitle('Rice Types')
ax[0].set_title('arborio = 0')
ax[1].set_title('basmati = 1')
ax[2].set_title('ipsala = 2')
ax[3].set_title('jasmine = 3')
ax[4].set_title('karacadag = 4')

# The images are already shuffled so I have to search for an image of each type
# I'm really hoping there is a function to do this, but then again how often
# do you look through a list of tuples as if they are key value pairs...
labels = [0, 1, 2, 3, 4];
while(True):
  batch = data_iterator.next()
  for i, img in enumerate(batch[0]):
    for l in labels:
      if batch[1][i] == l:
        labels.remove(batch[1][i])
        ax[batch[1][i]].imshow(img.astype(int))
  if not labels:
    break;
```



1.1.2 Preprocessing

Now we are going to: 1. Scale the data into values between 0 and 1 for the efficiency of the model 1. Note that our images are stored in rgb 2. We can apply a transformation to the whole dataset pipeiline 2. Partition the data into train and test

[ ]:
```
scaled = data.map(lambda x, y: (x/255, y))
# We can verify the data is scaled by taking a batch, and verifying the max
# value in an image is 1
scaled_it = scaled.as_numpy_iterator()
batch = scaled_it.next()
batch[0].max()

# We can also look at the data, just to ensure we didn't break it. Just note
# the image values are floats now, not ints
fig, ax = plt.subplots(ncols=5, figsize=(20,20))
for idx, img in enumerate(batch[0][:5]):
  ax[idx].imshow(img)
  ax[idx].title.set_text(batch[1][idx])
```



[ ]:
```
# The data has already been shuffled, so we can split using tfds.split
# This function makes up for the lack of a good way to split a df.data.dataset
def get_dataset_partitions_tf(ds, ds_size, train_split=0.8, val_split=0.1,␣
 ↪test_split=0.1, shuffle=True, shuffle_size=10000):
    assert (train_split + test_split + val_split) == 1

    if shuffle:
        # Specify seed to always have the same split distribution between runs
        ds = ds.shuffle(shuffle_size, seed=1234)

    train_size = int(train_split * ds_size)
    val_size = int(val_split * ds_size)

    train_ds = ds.take(train_size)
    val_ds = ds.skip(train_size).take(val_size)
    test_ds = ds.skip(train_size).skip(val_size)

    return train_ds, val_ds, test_ds
```

```
[ ]: print("Original datasize:\t",len(scaled))
train, val, test = get_dataset_partitions_tf(scaled, len(scaled), shuffle=False)
print("Training datasize:\t", len(train))
print("Validation datasize:\t",len(val))
print("Testing datasize:\t",len(test))
print("Total should be close:\t",len(train)+len(val)+len(test))
```

```
Original datasize:      2344
Training datasize:      1875
Validation datasize:    234
Testing datasize:       235
Total should be close:  2344
```

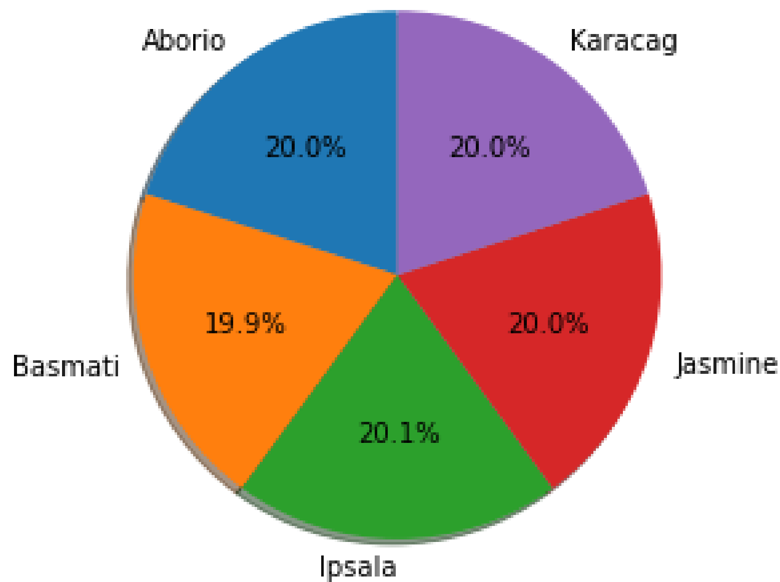
1.1.3 Viewing Class Distribution

As one final look into our data, we want to check to make sure the classes our evenly distributed. The issue is we are retrieving our data in batches at this point. We'll have to retrieve the whole dataset at once and count the label amounts. I'm referencing this stackoverflow.

```
[ ]: # I'm just going to keep imorting libraries as I go to make this text
# easy to use out of context
import numpy as np
num_classes = 5

def count_class(counts, batch):
    # y is tensor with every unique element from batch[1]
    # while c is the count of each of those corresponding elements
    y, _, c = tf.unique_with_counts(batch[1])
    # This function updates the given tensor (counts) with the count of the
    # given batch. As for expanding the dimensions of
    return tf.tensor_scatter_nd_add(counts, tf.expand_dims(y, axis=1), c)

# The reduce function calls a function on every element of a dataset,
# aggregating the result of the reduce function (our counting method), with an
#
counts = train.reduce(
    initial_state=tf.zeros(num_classes, tf.int32),
    reduce_func=count_class)
```

```
[ ]: # Lets import matplotlib again to avoid errors
import matplotlib.pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = 'Aborio', 'Basmati', 'Ipsala', 'Jasmine', "Karacag"
sizes = counts

fig1, ax1 = plt.subplots()
ax1.pie(sizes, labels=labels, autopct='%1.1f%%',
```

```
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()

# Well wasn't that a fun exercise
```



We can see that our taret classes are evenly distributed, and we have a great jumping off point to test some different neural networks!

1.2 Sequential Model

Now we create a basic feed-forward sequential model with dense layers. It's good if we just have 1 input (our image) and we just want to learn a given classification.

[ ]:
```
from keras.models import Sequential
from keras.layers import Dense, Flatten, Dropout
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(128, 128, 3)),
  tf.keras.layers.Dense(16, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(16, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(num_classes, activation='softmax'),
])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 49152)             0

 dense (Dense)               (None, 16)                786448

 dropout (Dropout)           (None, 16)                0

 dense_1 (Dense)             (None, 16)                272

 dropout_1 (Dropout)         (None, 16)                0

 dense_2 (Dense)             (None, 5)                 85

=================================================================
Total params: 786,805
Trainable params: 786,805
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

Now that we have compiled the network, I did have a bit of an issue with the loss function. This was because I didn't store the labels categorically, so I had to read into other loss functions. Easy solution used from this forum

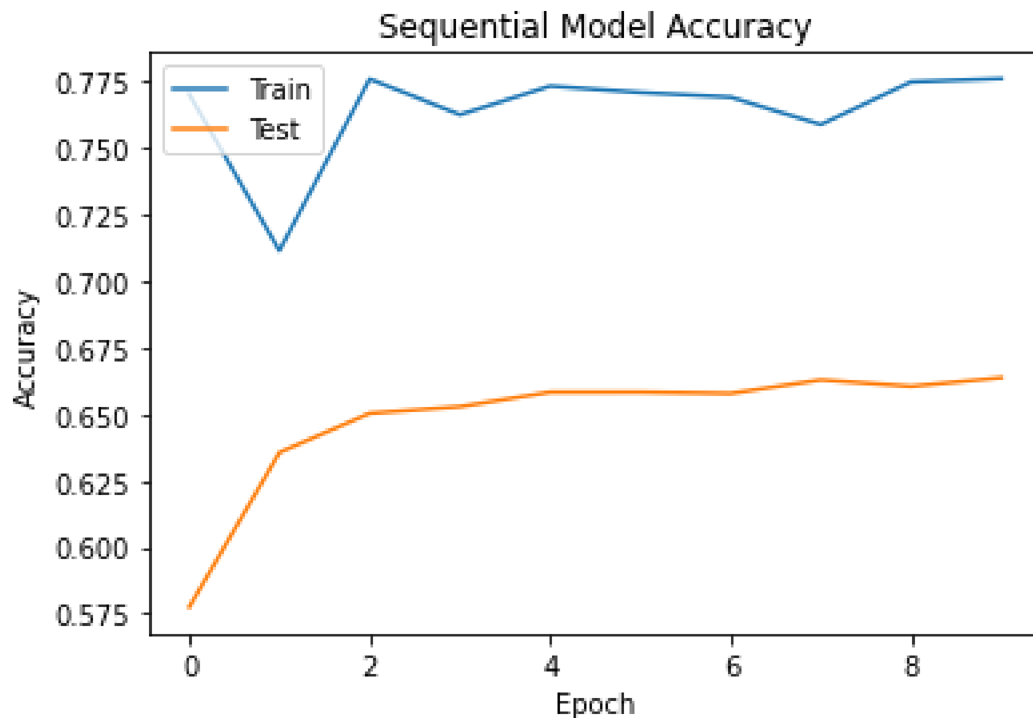
[ ]:
```
history = model.fit(train, epochs=10, validation_data=val, verbose=1)
```

```
Epoch 1/10
1875/1875 [==============================] - 116s 61ms/step - loss: 0.9801 -
accuracy: 0.5775 - val_loss: 0.5450 - val_accuracy: 0.7702
Epoch 2/10
1875/1875 [==============================] - 106s 56ms/step - loss: 0.8501 -
accuracy: 0.6356 - val_loss: 0.6103 - val_accuracy: 0.7114
Epoch 3/10
1875/1875 [==============================] - 110s 58ms/step - loss: 0.8097 -
accuracy: 0.6503 - val_loss: 0.4811 - val_accuracy: 0.7759
Epoch 4/10
1875/1875 [==============================] - 115s 61ms/step - loss: 0.7934 -
accuracy: 0.6528 - val_loss: 0.5024 - val_accuracy: 0.7626
Epoch 5/10
1875/1875 [==============================] - 117s 63ms/step - loss: 0.7775 -
accuracy: 0.6582 - val_loss: 0.4699 - val_accuracy: 0.7732
Epoch 6/10
1875/1875 [==============================] - 112s 60ms/step - loss: 0.7769 -
```

```
accuracy: 0.6582 - val_loss: 0.4812 - val_accuracy: 0.7708
Epoch 7/10
1875/1875 [==============================] - 112s 60ms/step - loss: 0.7738 -
accuracy: 0.6578 - val_loss: 0.4765 - val_accuracy: 0.7691
Epoch 8/10
1875/1875 [==============================] - 111s 59ms/step - loss: 0.7634 -
accuracy: 0.6629 - val_loss: 0.4999 - val_accuracy: 0.7588
Epoch 9/10
1875/1875 [==============================] - 106s 57ms/step - loss: 0.7734 -
accuracy: 0.6604 - val_loss: 0.4782 - val_accuracy: 0.7748
Epoch 10/10
1875/1875 [==============================] - 113s 60ms/step - loss: 0.7673 -
accuracy: 0.6637 - val_loss: 0.4623 - val_accuracy: 0.7760
```

[ ]:

```
import matplotlib.pyplot as plt

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



This is a sign we need some more dense layers. However I suspect the biggest loss in accuracy was from the downgrade in resolution in the model. Running the model with a resolution of 256 yielded better results but would result in a very long training time in more complicated convolutional or reccurent networks. It's worth noting a past run of this model did preform better, with double the resolution and layer size.

[ ]:
```
score = model.evaluate(test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.4600178599357605
Test accuracy: 0.7760915756225586
```

1.3 CNN: Convolutional Neural Networks

Now this is the *bees knees* when it comes to image processing. In a stroke of luck 3Blue1Brown released a youtube video on convolution right as this project started. Convolution (when used in ML) allows us to regularize a neural network by focusing on large scale patterns in something like an image, and scaling down to smaller and smaller details as a network adjusts weights. Kernels or Filters *convolve* throughout the data, and with each forward pass adjusts for finding a feature that benefits the network. I'd suggest watching the video for understanding the math behind how a convolution layer might compress data onto a feature map.

I actually found that the convolutional network didn't get all that much better running for 20 epochs then it did 10. The average increase in accuracy on the validation set was only ~.003, so I went ahead and cut it down to 5 epochs, a runtime of about an hour.

[ ]:
```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout
model_CNN = Sequential([
  Conv2D(8, (3,3), 1, activation='relu', input_shape=(128,128,3)),
  MaxPooling2D(),
  Conv2D(16, (3,3), 1, activation='relu'),
  MaxPooling2D(),
  Conv2D(16, (3,3), 1, activation='relu'),
  MaxPooling2D(),
  Flatten(),
  Dense(32, activation='relu'),
  Dense(5, activation='softmax')
])
model_CNN.summary()
```

```
Model: "sequential_1"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 126, 126, 8)       224
```

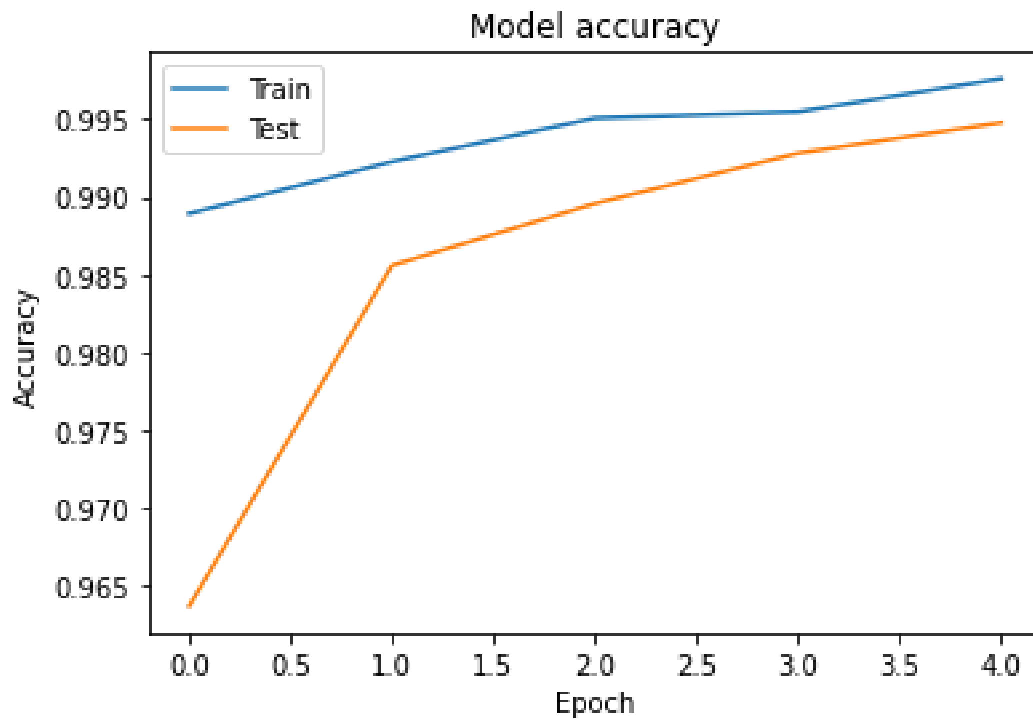
```
 max_pooling2d (MaxPooling2D  (None, 63, 63, 8)        0
 )

 conv2d_1 (Conv2D)           (None, 61, 61, 16)        1168

 max_pooling2d_1 (MaxPooling  (None, 30, 30, 16)       0
 2D)

 conv2d_2 (Conv2D)           (None, 28, 28, 16)        2320

 max_pooling2d_2 (MaxPooling  (None, 14, 14, 16)       0
 2D)

 flatten_1 (Flatten)         (None, 3136)              0

 dense_3 (Dense)             (None, 32)                100384

 dense_4 (Dense)             (None, 5)                 165

=================================================================
Total params: 104,261
Trainable params: 104,261
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
model_CNN.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

[ ]:
```
history_CNN = model_CNN.fit(train, batch_size=32, epochs=5, verbose=1,␣
 ↪validation_data=val)
```

```
Epoch 1/5
1875/1875 [==============================] - 698s 372ms/step - loss: 0.1049 -
accuracy: 0.9637 - val_loss: 0.0372 - val_accuracy: 0.9889
Epoch 2/5
1875/1875 [==============================] - 660s 352ms/step - loss: 0.0435 -
accuracy: 0.9856 - val_loss: 0.0234 - val_accuracy: 0.9923
Epoch 3/5
1875/1875 [==============================] - 655s 349ms/step - loss: 0.0320 -
accuracy: 0.9896 - val_loss: 0.0177 - val_accuracy: 0.9951
Epoch 4/5
1875/1875 [==============================] - 651s 347ms/step - loss: 0.0227 -
accuracy: 0.9928 - val_loss: 0.0137 - val_accuracy: 0.9955
Epoch 5/5
1875/1875 [==============================] - 649s 346ms/step - loss: 0.0163 -
accuracy: 0.9948 - val_loss: 0.0087 - val_accuracy: 0.9976
```

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

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



Looking at that graph it's clear we could get this to a higher accuracy with some more epochs, but for a short experimentation this is quite alright!

[ ]:
```
score = model_CNN.evaluate(test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.009167242795228958
Test accuracy: 0.9969382286071777
```

---

keras_learning_part_2ipynb

December 5, 2022

0.1 Reloading the Data

Google Colab isn't very fun to have to keep running, so I split the assignment into two google colab sessions. So I am importing the data here.

```
[ ]: import tensorflow as tf
     from tensorflow import keras
     import numpy as np
     from matplotlib import pyplot as plt

     # These are constants that are referenced often
     BATCH_SIZE = 32
     IMG_SIZE = (255, 255)
```

```
[ ]: from google.colab import files
     uploaded = files.upload()
     !unzip rice.zip
     print("Hope that didn't take too long!")
```

```
<IPython.core.display.HTML object>
```

```
Streaming output truncated to the last 5000 lines.
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (550).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5500).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5501).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5502).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5503).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5504).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5505).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5506).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5507).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5508).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5509).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (551).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5510).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5511).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5512).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5513).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5514).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (5515).jpg
```

```
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9967).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9968).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9969).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (997).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9970).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9971).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9972).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9973).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9974).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9975).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9976).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9977).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9978).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9979).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (998).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9980).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9981).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9982).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9983).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9984).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9985).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9986).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9987).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9988).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9989).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (999).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9990).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9991).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9992).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9993).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9994).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9995).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9996).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9997).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9998).jpg
  inflating: Rice_Image_Dataset/Karacadag/Karacadag (9999).jpg
  inflating: Rice_Image_Dataset/Rice_Citation_Request.txt
Hope that didn't take too long!
```

[ ]:
```
# I would like to use the GPU on Google Colab
# Lets check if we have access for free...
tf.test.gpu_device_name()
```

[ ]:
```
'/device:GPU:0'
```

[ ]:
```
data = keras.utils.image_dataset_from_directory("Rice_Image_Dataset",␣
↪seed=1234, batch_size=BATCH_SIZE, image_size=IMG_SIZE)
```

```
Found 75000 files belonging to 5 classes.
```

I'm now using the preprocessing guidlenes useful for transfering our images onto MobileNetV2. I'm referencing this to solve issues I had with figuring out exactly what the preprocessing function actually wants as parameters.

```
[ ]: # This function can be used later with the functional api to preprocess the data
     # on the fly!
     preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input
```

I'll still use a 0,1 scaling for the RNN network, so I'll just set up another scaling pipeline

```
[ ]: scaled_data = data.map(lambda x, y: (x/255, y))
```

```
[ ]: # The data has already been shuffled, so we can split using tfds.split
     # This function makes up for the lack of a good way to split a df.data.dataset
     def get_dataset_partitions_tf(ds, ds_size, train_split=0.8, val_split=0.1,␣
      ↪test_split=0.1, shuffle=True, shuffle_size=10000):
         assert (train_split + test_split + val_split) == 1

         if shuffle:
             # Specify seed to always have the same split distribution between runs
             ds = ds.shuffle(shuffle_size, seed=1234)

         train_size = int(train_split * ds_size)
         val_size = int(val_split * ds_size)

         train_ds = ds.take(train_size)
         val_ds = ds.skip(train_size).take(val_size)
         test_ds = ds.skip(train_size).skip(val_size)

         return train_ds, val_ds, test_ds
```

```
[ ]: rnn_train, rnn_val, rnn_test = get_dataset_partitions_tf(scaled_data,␣
      ↪len(scaled_data), shuffle=False)
```

```
[ ]: mobile_train, mobile_val, mobile_test = get_dataset_partitions_tf(data,␣
      ↪len(scaled_data), shuffle=False)
```

0.2 RNN: Recurrent Neural Network

Now, Recurrent Neural Networks use a memory state in order to accurately creating sequential pattern recognition. In our case, that isn't very applicable. However, just to get used to the concept, lets make a simple one! Because we are using an image, I went ahead and flattened the input into a single vector that is then embedded

[ ]:
```
model_RNN = keras.models.Sequential()
model_RNN.add(keras.layers.Flatten(input_shape=(IMG_SIZE+(3,))))
model_RNN.add(keras.layers.Embedding(1000, 32))
model_RNN.add(keras.layers.SimpleRNN(32))
model_RNN.add(keras.layers.Dense(1, activation='sigmoid'))
```

[ ]:
```
model_RNN.summary()
```

```
Model: "sequential_2"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 195075)            0

 embedding_2 (Embedding)     (None, 195075, 32)        320000

 simple_rnn_2 (SimpleRNN)    (None, 32)                2080

 dense_3 (Dense)             (None, 1)                 33

=================================================================
Total params: 322,113
Trainable params: 322,113
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
model_RNN.compile(loss='sparse_categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

[ ]:
```

```

Now I just couldn't get this poor model to train in a manageable time frame with good results. I think that this archiecture isn't really suited to the classification being done here anyway, so I'm safe saying the result here wouldn't have been that impactful on my understanding of the model architecture.

0.3 Pretrained Model Transfer

We will be using the the guide for transfer learning from TensorFlow. It thankfully already uses the the data pipeline I was using for previous models, and showed me how to do a lot of things, like displaying class names much easier.

Following the guide, we configure the data a bit for performance using autotune:

[ ]:
```
AUTOTUNE = tf.data.AUTOTUNE

mobile_train = mobile_train.prefetch(buffer_size=AUTOTUNE)
```

```
mobile_val = mobile_val.prefetch(buffer_size=AUTOTUNE)
mobile_test = mobile_test.prefetch(buffer_size=AUTOTUNE)
```

[ ]:
```
print('Number of training batches: %d' % tf.data.experimental.
 ↪cardinality(mobile_train))
print('Number of validation batches: %d' % tf.data.experimental.
 ↪cardinality(mobile_val))
print('Number of test batches: %d' % tf.data.experimental.
 ↪cardinality(mobile_test))
```

```
Number of training batches: 1875
Number of validation batches: 234
Number of test batches: 235
```

We then load in every layer of the pretrained model except for the last one, which we plan to replace with our own classification layer

[ ]:
```
# Create the base model from the pre-trained model MobileNet V2
IMG_SHAPE = IMG_SIZE + (3,)
# Imports images from the data set in a tensor.shape = (255, 255, 3)
# with the pixels on a scale from -1 to 1
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                               include_top=False,
                                               weights='imagenet')
```

```
WARNING:tensorflow:`input_shape` is undefined or non-square, or `rows` is not in
[96, 128, 160, 192, 224]. Weights for input shape (224, 224) will be loaded as
the default.
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applicatio
ns/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h
5
9406464/9406464 [==============================] - 0s 0us/step
```

Below is an example of a feature extractor…

> "This feature extractor converts each [255]x[255]x3 image into a 5x5x1280 block of features. Let's see what it does to an example batch of images:"

I reckon that this is for making each image have smaller subcomponents easier for feature detection, and this exactly what happens in the model as our inputs are convoluted through the model.

[ ]:
```
image_batch, label_batch = next(iter(mobile_train))
print(image_batch.shape)
feature_batch = base_model(image_batch)
print(feature_batch.shape)
```

```
(32, 255, 255, 3)
(32, 8, 8, 1280)
```

Now we have the data in the right form, we want to freeze the base of the model (a convolutional model by the way), and add our own categorical layer at the top.

[ ]:
```
base_model.trainable = False
# Now this is a complicated model!
base_model.summary()
```

To add a classification layer/head, we want to average the 8x8 sections of each image we divied out in the last step. This means we will have a layer of only one 1280 element vector that represents an average of the previous layer

[ ]:
```
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
feature_batch_average = global_average_layer(feature_batch)
print(feature_batch_average.shape)
```

```
(32, 1280)
```

After creating a layer to find the average of the previous layer, we want to convert the layers to prediction per image. We have 5 different classes, so we have a Dense layer with 5 nodes, and I'm picking an activation function of `softmax` since it will probably fit the categorical distribution the best.

[ ]:
```
prediction_layer = tf.keras.layers.Dense(5, activation="softmax")
prediction_batch = prediction_layer(feature_batch_average)
print(prediction_batch.shape)
```

```
(32, 5)
```

We can now build a model by chaining all of these elements together. The guide uses the functional method of building a model rather then the sequential so might have to get a bit confused here. In the functionaly API, you continuously call layers on an input object. The kind of obtuse part of this API is that you can apply all these function mappings and layers with all the same syntax! I reccomend reading the Functional API if you want to know what happens under the hood in this code block

[ ]:
```
# So we would define the input
inputs = tf.keras.Input(shape=(255, 255, 3))
# We can then preprocess the data by just calling the preprocess function
# this allows us to use more of a data pipeline
x = preprocess_input(inputs)
# Then we can apply all of the layers from the base model
x = base_model(x, training=False)
# Add our previously created average layer that will average the output of the
# last layer in the base_model, which should be of the shape (8, 8, 1280)
x = global_average_layer(x)
# We add a dropout here... because we can!
x = tf.keras.layers.Dropout(0.2)(x)
outputs = prediction_layer(x)
model = tf.keras.Model(inputs, outputs)
```

We then must compile this model before training it. Note that we will be using `tf.keras.losses.sparse_categorical_crossentroypy` since by default our labels are ints and changing them to a one hot encoding of categorical data makes the data a bit harder to explore.

The one hot encoding would train better, but it *shouldn't* effect the actual accuracy of the final model.

[ ]:
```
base_learning_rate = 0.0001
model.compile(optimizer=tf.keras.optimizers.
 ↪Adam(learning_rate=base_learning_rate),
              loss=tf.keras.losses.sparse_categorical_crossentropy,
              metrics=['accuracy'])
```

[ ]:
```
model.summary()
```

```
Model: "model"

_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_2 (InputLayer)        [(None, 255, 255, 3)]     0         
                                                                 
 tf.math.truediv (TFOpLambda  (None, 255, 255, 3)      0         
 )                                                               
                                                                 
 tf.math.subtract (TFOpLambd  (None, 255, 255, 3)      0         
 a)                                                              
                                                                 
 mobilenetv2_1.00_224 (Funct  (None, 8, 8, 1280)       2257984   
 ional)                                                          
                                                                 
 global_average_pooling2d (G  (None, 1280)             0         
 lobalAveragePooling2D)                                          
                                                                 
 dropout (Dropout)           (None, 1280)              0         
                                                                 
 dense (Dense)               (None, 5)                 6405      
                                                                 
=================================================================
Total params: 2,264,389
Trainable params: 6,405
Non-trainable params: 2,257,984
_________________________________________________________________
```

We can see above how all of our layers ended up in the final model, finally verifying that convultion did indeed give us a image shape of (8, 8, 1280) before our average pooling layer.

Lets train! The guide actually has us set a baseline loss and accuracy before we fit the model, nifty! Unfortunately we don't really have the time for a baseline comparison…

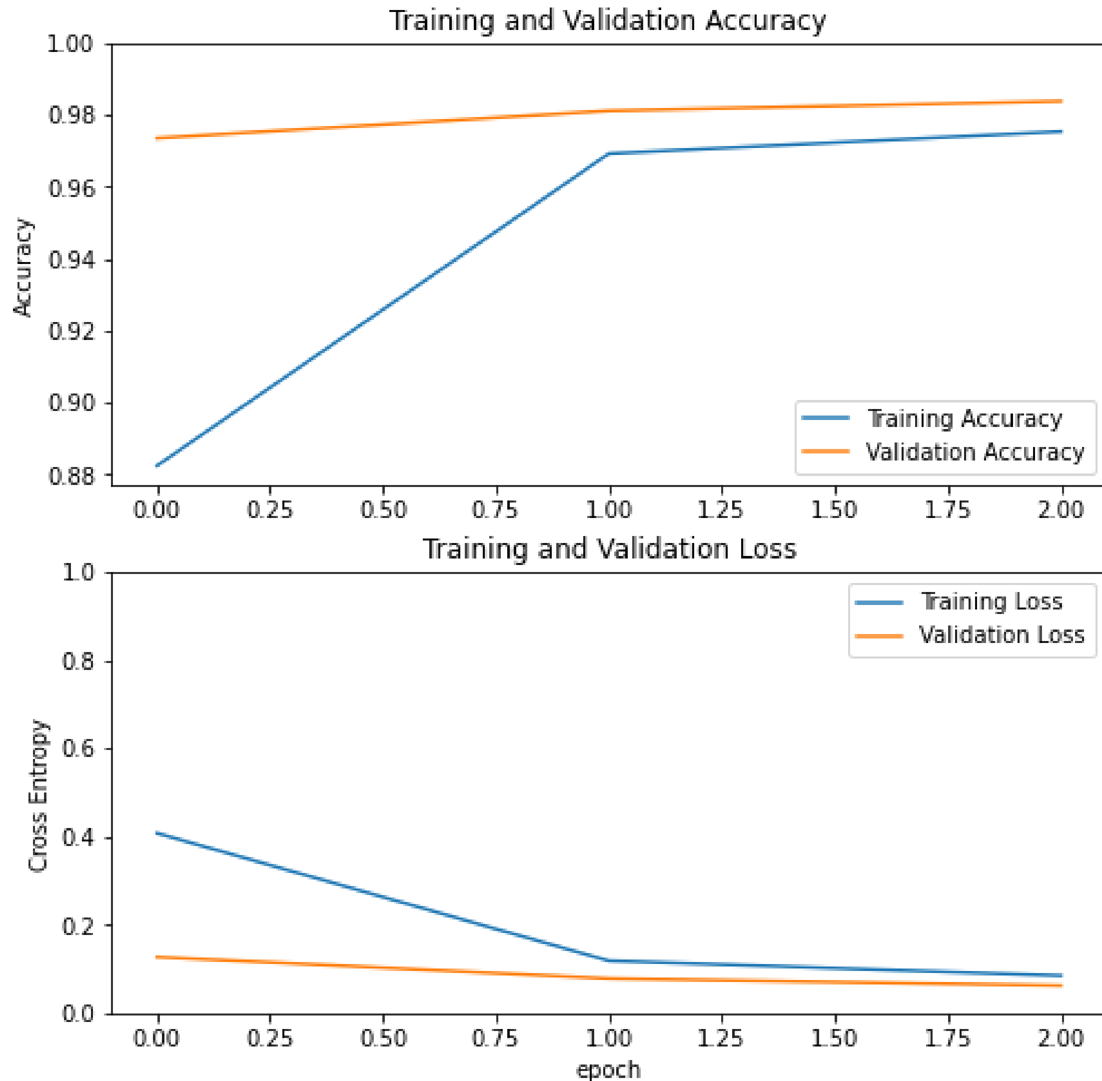
[ ]:
```
initial_epochs = 3
# loss0, accuracy0 = model.evaluate(mobile_val)
# print("initial loss: {:.2f}".format(loss0))
# print("initial accuracy: {:.2f}".format(accuracy0))
```

[ ]:
```
# We have the epoch set so low because... life is short
history = model.fit(mobile_train,
                    epochs=initial_epochs,
                    validation_data=mobile_val)
```

```
Epoch 1/3
1875/1875 [==============================] - 196s 103ms/step - loss: 0.4076 -
accuracy: 0.8825 - val_loss: 0.1265 - val_accuracy: 0.9736
Epoch 2/3
1875/1875 [==============================] - 186s 99ms/step - loss: 0.1179 -
accuracy: 0.9693 - val_loss: 0.0779 - val_accuracy: 0.9812
Epoch 3/3
1875/1875 [==============================] - 185s 99ms/step - loss: 0.0844 -
accuracy: 0.9754 - val_loss: 0.0620 - val_accuracy: 0.9838
```

[ ]:
```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
print("Accuracy over epochs: ", acc)


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

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

plt.subplot(2, 1, 2)
plt.plot(loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.ylabel('Cross Entropy')
plt.ylim([0,1.0])
plt.title('Training and Validation Loss')
plt.xlabel('epoch')
plt.show()
```

```
Accuracy over epochs:  [0.8824833035469055, 0.9692999720573425,
0.9753666520118713]
```



Note that the validation metrics are better than the training metrics because layers that prevent overfitting like dropout and batch normalization are turned off when calculating validation loss (sense that doesn't benefit the model)

---

1 Analysis

The first thing I want to state is that I definitely learned how this api works by the end of this project, and I regret not having thoroughly reviewing the API at the start. With my learning as I went, as well as trying to understand how each architcture works, there wasn't much time to really train my models. However, I've gained some insights into the 4 approaches used.

1. A vanilla sequential model

2. A basic CNN model
3. A basic RNN model
4. A CNN model made from Google's MobileNetV2

Really the trend shown in my experiment is that better convolution networks result in better performance in the final model. While a basic dense sequential model mimics pattern recognition to an easily understandable degree, it only *approximates* a capacity for feature selection. Our convolutional networks instead manually add a feature selection process through convolution. We can see that this directly results in higher accuracy in our models. This is of course directly opposed to an RNN model where the new memory state doesn't do much to help with image recognition.

The convolutional model taken from MobileNetV2 takes this all a step further, with a large model already tuned for feature selection in images. We basically are given a model that can recognize features embedded on a manifold of our 2D image space, that space being more grounded in what images actually look like. By starting there our model can more generaly expand to classify outliers, as well as expect a touch more accuracy. We get that reflected improvement, however, its dampened by how clinically clean our rice images are.

> Note: For a easy comparison between the efficiency of these methods, the transfer network probably had a much higher ceiling for how well it could identify rice given the training data, but because there was so little noise in the data, it was much easier to get a quickly trained model from a simple CNN then go through all the trouble transfering performance from google. If I were using a much more complex dataset with hope for expansion, I would most likely use MobileNetworkV2. Especially if I had more training time then I do now (about 2 hours til the deadline)

This was all still a worthwhile exploration into image recognition, and I'm now a bit curious as to how we can use this as a jumping off point into image stitching and general manipulation. Oh and also, as convenient as Google Colab is, I have a 3080 graphics card in my laptop, which could train the model well over 10x faster then what this cloud comput option gives me. Time to set up a local envrionoment!