image_classification_transfer_learning

December 4, 2022

1 Image Classification Using Transfer Learning

I built an image classifier that identifies what breed a dog is based off of an image of the dog. I followed concepts from this tutorial: https://www.tensorflow.org/tutorials/images/transfer_learning

1.1 Setup

Import TensorFlow and other libraries that are necessary.

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import PIL
import tensorflow as tf
import tensorflow_hub as hub
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"

from tensorflow import io
from tensorflow import keras
from tensorflow import image
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential


from PIL import Image
```

1.2 Reading the Data and Splitting it into Training/Validation Sets

The data was taken from the Stanford Dogs Dataset (https://www.kaggle.com/datasets/jessicali9530/stanford-dogs-dataset). This image data set can be used to train models to identify the breed of a dog based off an image. There are over 20,000 different images of 120 dog breeds in this data set, but I narrowed it down to only include 8 of my favorite dog breeds.

After reading in the data, we can see that there are 1361 files belonging to 8 classes. 80% of these files (1089 images) will be used for training the model, while 20% (272 images) will be used for validation.

[ ]:
```
batch_size = 32
img_size = (100,100)
```

```
path = "dog_breeds"

training_data = tf.keras.utils.image_dataset_from_directory(
    path,
    validation_split=0.2,
    subset="training",
    seed=123,
    image_size=img_size,
    batch_size=batch_size
)

validation_data = tf.keras.preprocessing.image_dataset_from_directory(
    path,
    validation_split=0.2,
    subset="validation",
    seed=123,
    image_size=img_size,
    batch_size=batch_size
)

class_names = training_data.class_names
print("\nClass Names:", class_names)
```

```
Found 1361 files belonging to 8 classes.
Using 1089 files for training.
Found 1361 files belonging to 8 classes.
Using 272 files for validation.

Class Names: ['chihuahua', 'german_shepard', 'golden_retriever', 'lhasa_apso',
'rottweiler', 'samoyed', 'siberian_husky', 'traditional_poodle']
```

We will also use some data from the validation batch to create a test batch.

[ ]:
```
val_batches = tf.data.experimental.cardinality(validation_data)
test_data = validation_data.take(val_batches // 5)
validation_data = validation_data.skip(val_batches // 5)

print('Number of validation batches: %d' % tf.data.experimental.
 ↪cardinality(validation_data))
print('Number of test batches: %d' % tf.data.experimental.
 ↪cardinality(test_data))
```

```
Number of validation batches: 8
Number of test batches: 1
```

1.3 Configuring the Dataset for Performance

This segment of code allows for buffered prefetching, while yields data from the disk without having I/O be blocking. Overall, it allows for better performance while building the model.

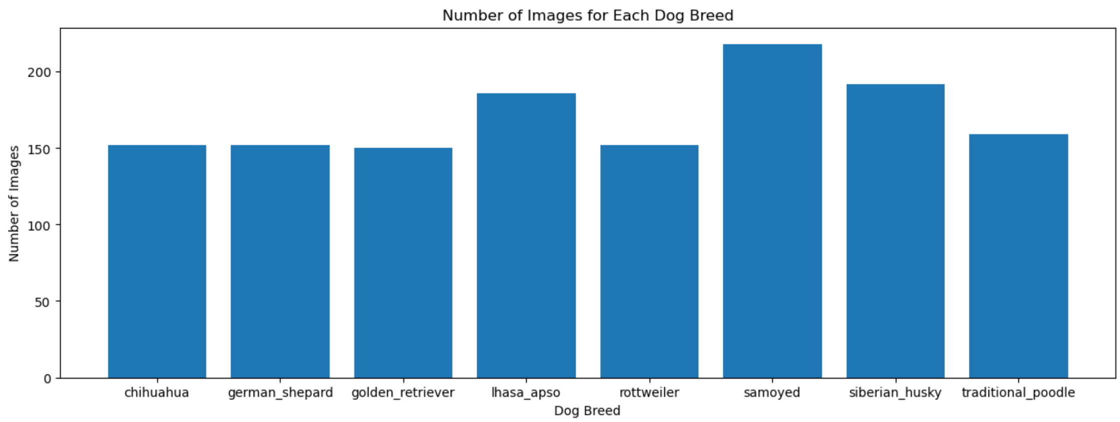
[ ]:
```
AUTOTUNE = tf.data.AUTOTUNE

training_data = training_data.prefetch(buffer_size=AUTOTUNE)
validation_data = validation_data.prefetch(buffer_size=AUTOTUNE)
test_data = test_data.prefetch(buffer_size=AUTOTUNE)
```

1.4 Graphing the Distribution of Target Classes

We can build a plot to see the distribution of data that we have available to us. The Samoyed class has the most number of images available, while the Golden Retriever class has the least number of images available.

[ ]:
```
import fnmatch

class_distribution = []

for classification in class_names:
    dir_path = os.path.join(path, classification)
    count = len(fnmatch.filter(os.listdir(dir_path), '*.*'))
    class_distribution.append(count)

print(class_distribution)

x = np.array(class_names)
y = np.array(class_distribution)

plt.figure(figsize=(15, 5))

plt.bar(x,y)

plt.title("Number of Images for Each Dog Breed")
plt.xlabel("Dog Breed")
plt.ylabel("Number of Images")

plt.show()
```

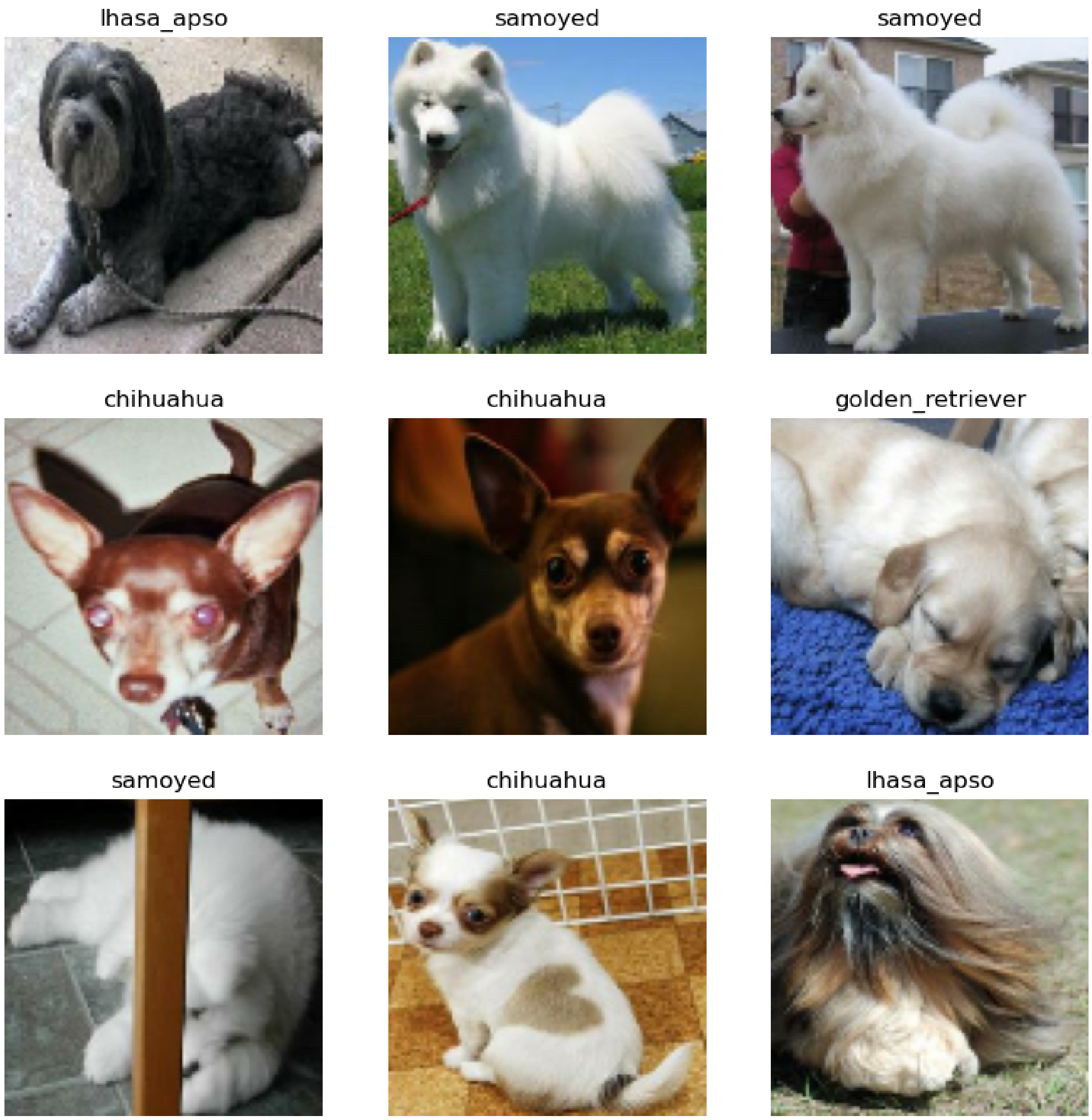
```
[152, 152, 150, 186, 152, 218, 192, 159]
```



1.5 Visualizing the Data

Let's take a look at a couple of images from the training set.

[ ]:
```
plt.figure(figsize=(10,10))

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



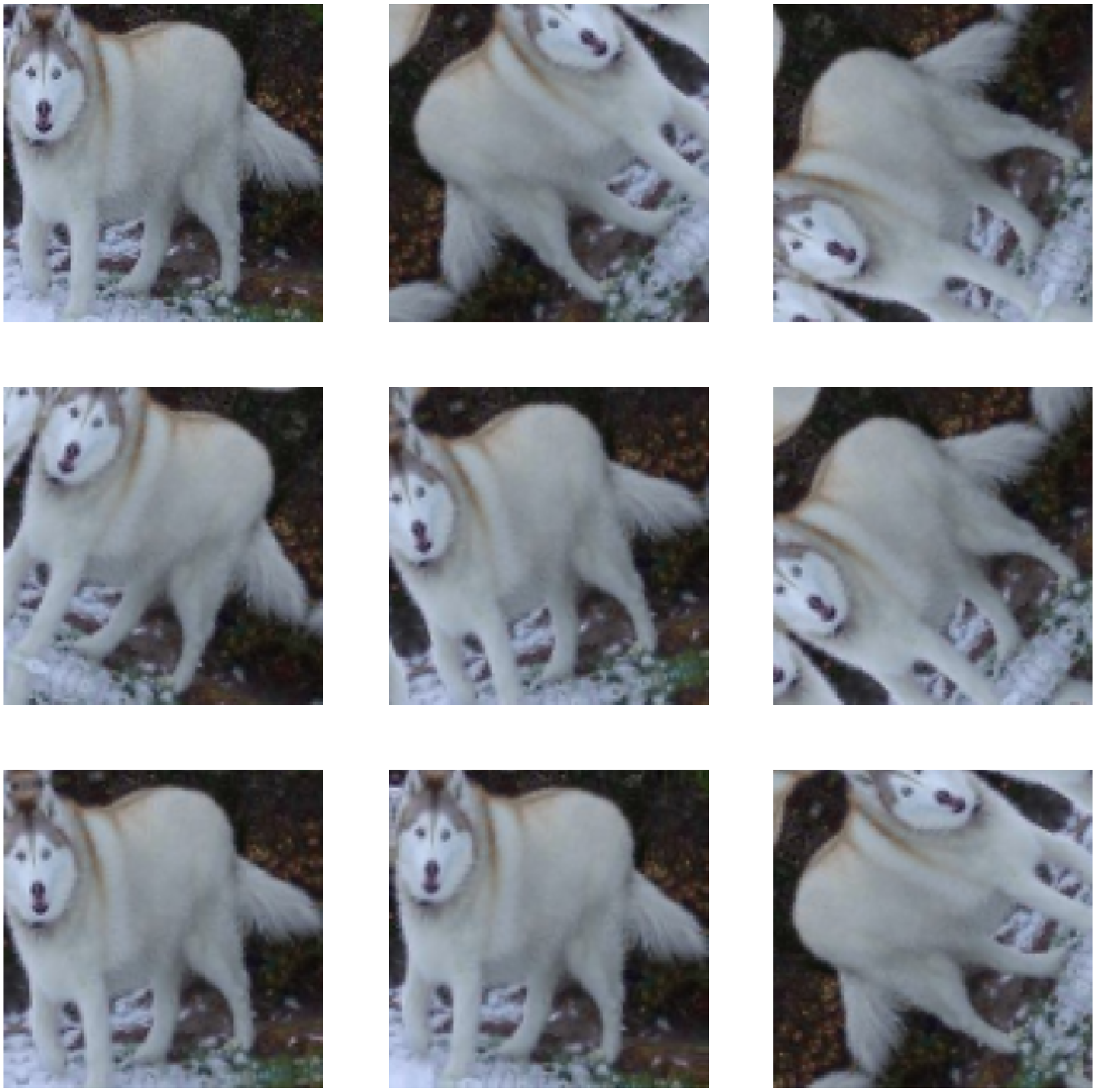
1.6 Data Augmentation

Because this data set has a small number of training examples, there could be overfitting. Data augmentation tackles overfitting by generating more training data from the existing data through using random transformations that create images which look believable. An example of an image that has underwent data augmentation is shown below.

[ ]:
```
data_augmentation = tf.keras.Sequential([
  tf.keras.layers.RandomFlip('horizontal'),
  tf.keras.layers.RandomRotation(0.2),
])
```

[ ]:
```
for image, _ in training_data.take(1):
  plt.figure(figsize=(10, 10))
  first_image = image[0]
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    augmented_image = data_augmentation(tf.expand_dims(first_image, 0))
    plt.imshow(augmented_image[0] / 255)
    plt.axis('off')
```

```
WARNING:tensorflow:5 out of the last 5 calls to <function pfor.<locals>.f at
0x0000021770AB1940> triggered tf.function retracing. Tracing is expensive and
the excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
WARNING:tensorflow:6 out of the last 6 calls to <function pfor.<locals>.f at
0x0000021770B34F70> triggered tf.function retracing. Tracing is expensive and
the excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
```



1.7 Rescaling the Pixel Values

We will be using tf.keras.applications.MobileNetV2 as a base model, which requires pixel values to be between [-1, 1]. Currently, our pixel values are between [0, 255]. We can use a built-in preprocessing method included with the model to rescale the pixel values.

[ ]:
```
preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input
```

[ ]:
```
rescale = tf.keras.layers.Rescaling(1./127.5, offset=-1)
```

1.8 Creating a Base Model from a Pre-trained CovNet

We will be creating our base model using MobileNet V2, which was developed at Google. It is trained on the ImageNet dataset, which is a very large dataset that has a wide variety of categories.

This base of knowledge will assist us in identifying what breed a dog is.

After creating our base model, we need to pick which layer of MobileNet V2 we will use for feature extraction. As per common practice, we will choose the "bottleneck layer," which is the very last layer before the flatten operation.

[ ]:
```
# Create the base model from the pre-trained model MobileNet V2
IMG_SHAPE = img_size + (3,)
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.
```

The feature extraction feature converts each 100x100x3 sized image into a 5x5x1280.

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

```
(32, 4, 4, 1280)
```

1.9 Freezing the Convolutional Base

Before compiling the model, it is important to freeze the convolutional base. This prevents weights in a given layer from being updated during training.

[ ]:
```
base_model.trainable = False
```

1.10 Adding a Classification Head

We can use a tf.keras.layers.GlobalAveragePooling2D layer to convert the features into a single 1280-element vector per image.

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

```
(32, 1280)
```

By adding a tf.keras.layers.Dense layer, we can convert these features into a single prediction per image.

[ ]:
```
prediction_layer = tf.keras.layers.Dense(8, activation='softmax')
prediction_batch = prediction_layer(feature_batch_average)
print(prediction_batch.shape)
```

1.11 Building the Model

Using data augmentation, rescaling, our base model, and our feature extraction layer, we can make a build a new model. Because training=false was used previously, this creates a batch normalization layer.

[ ]:
```
inputs = tf.keras.Input(shape=(100, 100, 3))
x = data_augmentation(inputs)
x = preprocess_input(x)
x = base_model(x, training=False)
x = global_average_layer(x)
x = tf.keras.layers.Dropout(0.2)(x)
outputs = prediction_layer(x)
model = tf.keras.Model(inputs, outputs)
```

1.12 Compiling the Model

The Adam algorithm is used for the optimizer and the Sparse Categorical Cross Entropy function is used for the loss function. By passing accuracy to the metrics parameter, we can see the training and validation accuracy for each epoch.

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

1.13 Training the Model

Using the model.fit() function, we can train the model for 20 epochs. At the end of the 20th epoch, our model has a training accuracy of 91% and a validation accuracy of about 97%.

[ ]:
```
epochs=20
history = model.fit(
  training_data,
  validation_data=validation_data,
  epochs=epochs
)
```

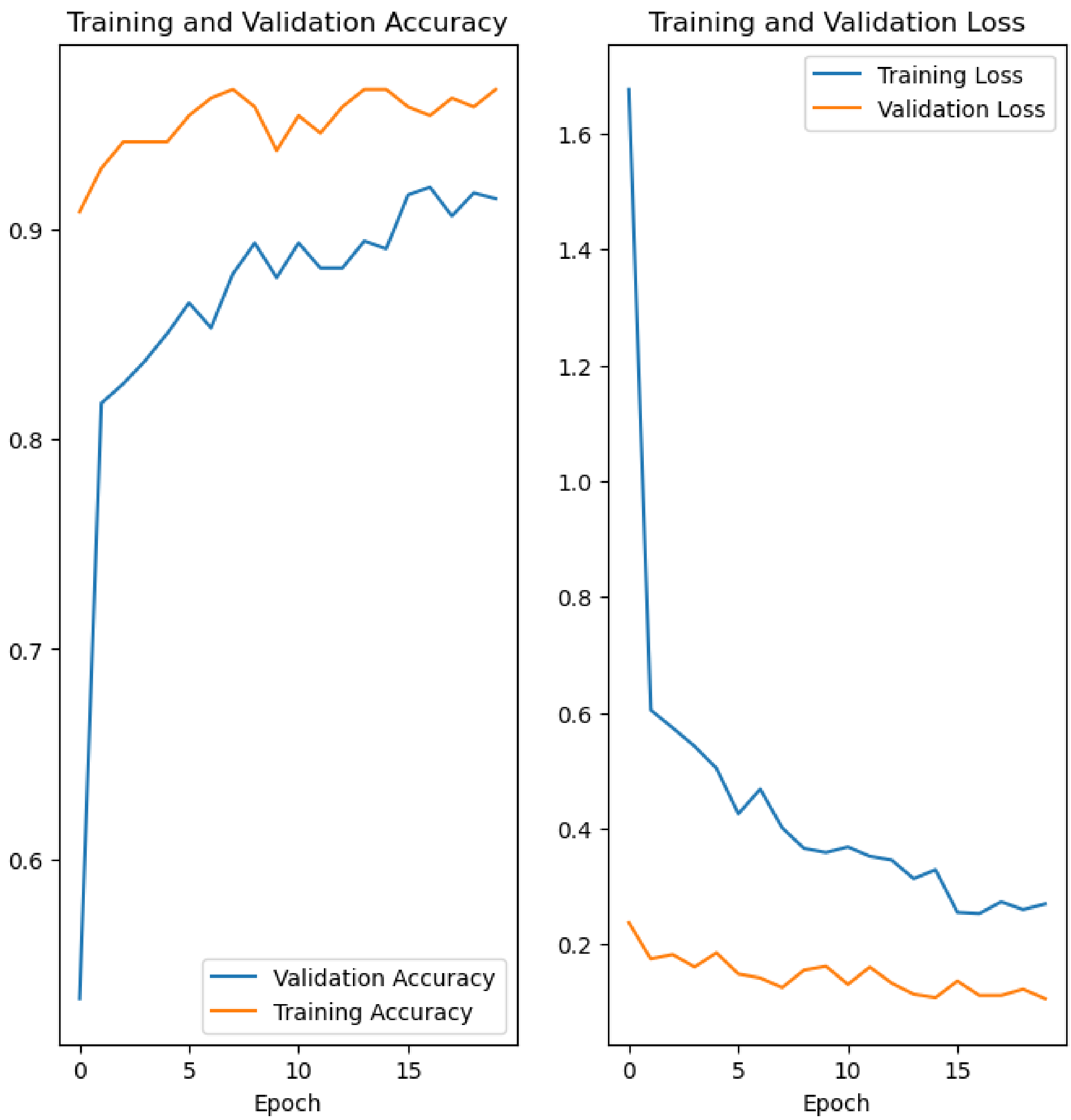
```
Epoch 1/20
35/35 [==============================] - 19s 396ms/step - loss: 1.6761 -
accuracy: 0.5335 - val_loss: 0.2373 - val_accuracy: 0.9083
Epoch 2/20
35/35 [==============================] - 18s 506ms/step - loss: 0.6043 -
accuracy: 0.8173 - val_loss: 0.1747 - val_accuracy: 0.9292
Epoch 3/20
35/35 [==============================] - 16s 452ms/step - loss: 0.5736 -
accuracy: 0.8264 - val_loss: 0.1822 - val_accuracy: 0.9417
Epoch 4/20
35/35 [==============================] - 16s 437ms/step - loss: 0.5418 -
accuracy: 0.8375 - val_loss: 0.1606 - val_accuracy: 0.9417
Epoch 5/20
```

```
35/35 [==============================] - 15s 426ms/step - loss: 0.5037 -
accuracy: 0.8503 - val_loss: 0.1851 - val_accuracy: 0.9417
Epoch 6/20
35/35 [==============================] - 15s 435ms/step - loss: 0.4252 -
accuracy: 0.8650 - val_loss: 0.1486 - val_accuracy: 0.9542
Epoch 7/20
35/35 [==============================] - 18s 505ms/step - loss: 0.4678 -
accuracy: 0.8531 - val_loss: 0.1416 - val_accuracy: 0.9625
Epoch 8/20
35/35 [==============================] - 21s 609ms/step - loss: 0.4008 -
accuracy: 0.8788 - val_loss: 0.1252 - val_accuracy: 0.9667
Epoch 9/20
35/35 [==============================] - 17s 476ms/step - loss: 0.3656 -
accuracy: 0.8935 - val_loss: 0.1552 - val_accuracy: 0.9583
Epoch 10/20
35/35 [==============================] - 16s 438ms/step - loss: 0.3584 -
accuracy: 0.8770 - val_loss: 0.1619 - val_accuracy: 0.9375
Epoch 11/20
35/35 [==============================] - 17s 473ms/step - loss: 0.3681 -
accuracy: 0.8935 - val_loss: 0.1306 - val_accuracy: 0.9542
Epoch 12/20
35/35 [==============================] - 16s 448ms/step - loss: 0.3519 -
accuracy: 0.8815 - val_loss: 0.1604 - val_accuracy: 0.9458
Epoch 13/20
35/35 [==============================] - 18s 518ms/step - loss: 0.3453 -
accuracy: 0.8815 - val_loss: 0.1328 - val_accuracy: 0.9583
Epoch 14/20
35/35 [==============================] - 15s 432ms/step - loss: 0.3132 -
accuracy: 0.8944 - val_loss: 0.1136 - val_accuracy: 0.9667
Epoch 15/20
35/35 [==============================] - 16s 462ms/step - loss: 0.3283 -
accuracy: 0.8907 - val_loss: 0.1078 - val_accuracy: 0.9667
Epoch 16/20
35/35 [==============================] - 21s 582ms/step - loss: 0.2551 -
accuracy: 0.9164 - val_loss: 0.1365 - val_accuracy: 0.9583
Epoch 17/20
35/35 [==============================] - 19s 523ms/step - loss: 0.2530 -
accuracy: 0.9201 - val_loss: 0.1115 - val_accuracy: 0.9542
Epoch 18/20
35/35 [==============================] - 19s 527ms/step - loss: 0.2730 -
accuracy: 0.9063 - val_loss: 0.1115 - val_accuracy: 0.9625
Epoch 19/20
35/35 [==============================] - 17s 475ms/step - loss: 0.2598 -
accuracy: 0.9174 - val_loss: 0.1224 - val_accuracy: 0.9583
Epoch 20/20
35/35 [==============================] - 17s 457ms/step - loss: 0.2694 -
accuracy: 0.9146 - val_loss: 0.1059 - val_accuracy: 0.9667
```

1.14 Visualizing the Results of Training

Using the accuracy and loss of the training and validation sets, we can create plots. As more epochs pass, the validation accuracy increases and is more similar to the training accuracy.

[ ]:
```
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='Validation Accuracy')
plt.plot(epochs_range, val_acc, label='Training Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
plt.xlabel("Epoch")

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.xlabel("Epoch")
plt.show()
```



1.15 Fine Tuning

So far, we have only trained a few layers on top of the MobileNetV2 base model. However, the weights of the pre-trained network were not updated.

Using fine-tuning, we can train the weights of the top layer of the pre-trained layer alongside the training of the classifier we added. This will make our model more accurate.

First, we un-freeze the top layers of the model and set the bottem layers to be un-trainable.

[ ]:
```
base_model.trainable = True
```

[ ]:
```python
print("Number of layers in the base model: ", len(base_model.layers))

fine_tune_at = 100

for layer in base_model.layers[:fine_tune_at]:
  layer.trainable = False
```

```
Number of layers in the base model:  154
```

Next, we can recompile the model and continue training.

[ ]:
```python
base_learning_rate = 0.0001
model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              optimizer = tf.keras.optimizers.
 ↪RMSprop(learning_rate=base_learning_rate/10),
              metrics=['accuracy'])
```

[ ]:
```python
fine_tune_epochs = 10
total_epochs =  epochs + fine_tune_epochs

history_fine = model.fit(training_data,
                         epochs=total_epochs,
                         initial_epoch=history.epoch[-1],
                         validation_data=validation_data)
```

```
Epoch 20/30
35/35 [==============================] - 54s 1s/step - loss: 0.3067 - accuracy:
0.9036 - val_loss: 0.1289 - val_accuracy: 0.9583
Epoch 21/30
35/35 [==============================] - 54s 2s/step - loss: 0.2415 - accuracy:
0.9192 - val_loss: 0.1166 - val_accuracy: 0.9500
Epoch 22/30
35/35 [==============================] - 41s 1s/step - loss: 0.2169 - accuracy:
0.9339 - val_loss: 0.1200 - val_accuracy: 0.9542
Epoch 23/30
35/35 [==============================] - 48s 1s/step - loss: 0.2149 - accuracy:
0.9256 - val_loss: 0.1271 - val_accuracy: 0.9583
Epoch 24/30
35/35 [==============================] - 40s 1s/step - loss: 0.1811 - accuracy:
0.9412 - val_loss: 0.1195 - val_accuracy: 0.9583
Epoch 25/30
35/35 [==============================] - 40s 1s/step - loss: 0.1769 - accuracy:
0.9339 - val_loss: 0.1317 - val_accuracy: 0.9583
Epoch 26/30
35/35 [==============================] - 40s 1s/step - loss: 0.1946 - accuracy:
0.9339 - val_loss: 0.1406 - val_accuracy: 0.9542
Epoch 27/30
35/35 [==============================] - 42s 1s/step - loss: 0.1736 - accuracy:
0.9339 - val_loss: 0.1180 - val_accuracy: 0.9500
```

```
Epoch 28/30
35/35 [==============================] - 40s 1s/step - loss: 0.1707 - accuracy:
0.9522 - val_loss: 0.1093 - val_accuracy: 0.9625
Epoch 29/30
35/35 [==============================] - 42s 1s/step - loss: 0.1438 - accuracy:
0.9431 - val_loss: 0.1465 - val_accuracy: 0.9500
Epoch 30/30
35/35 [==============================] - 44s 1s/step - loss: 0.1397 - accuracy:
0.9467 - val_loss: 0.1340 - val_accuracy: 0.9542
```

1.16 Evaluating the Model

When evaluated on the test data, our model performed quite well with an accuracy of 94%.

[ ]:
```
score = model.evaluate(test_data, verbose=0)

print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
Test loss: 0.215651735663414
Test accuracy: 0.9375
```

1.17 Predicting on New Data

Let's classify an image that the neural network hasn't seen before. The neural network was able to correctly classify the dog as a german shepard with 27.93% confidence.

[ ]:
```
im_path = 'test1.jpg'

im = io.read_file(im_path)
im = image.decode_jpeg(im, channels=3)
plt.imshow(im)


img = tf.keras.utils.load_img(
    im_path, target_size=img_size
)

img_array = tf.keras.utils.img_to_array(img)
img_array = tf.expand_dims(img_array, 0)

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

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

```
1/1 [==============================] - 0s 78ms/step
This image most likely belongs to german_shepard with a 27.93 percent
confidence.
```