image_classification_cnn

December 3, 2022

1 Image Classification with a CNN

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/classification

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 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_height = 100
img_width = 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_height, img_width),
    batch_size=batch_size
)

validation_data = tf.keras.utils.image_dataset_from_directory(
    path,
    validation_split=0.2,
    subset="validation",
    seed=123,
    image_size=(img_height, img_width),
    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']
```

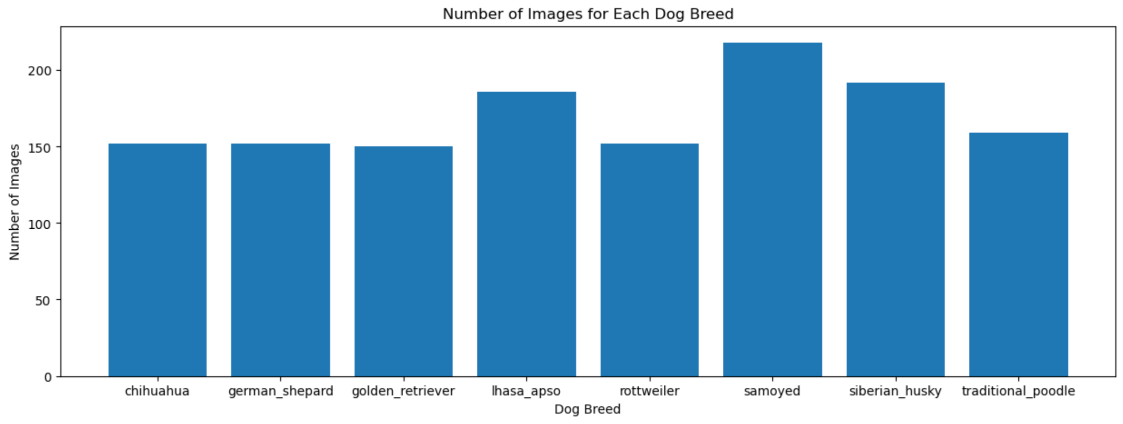
1.3 Graphing the Distribution of Target Classes

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)
```

```python
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()
```

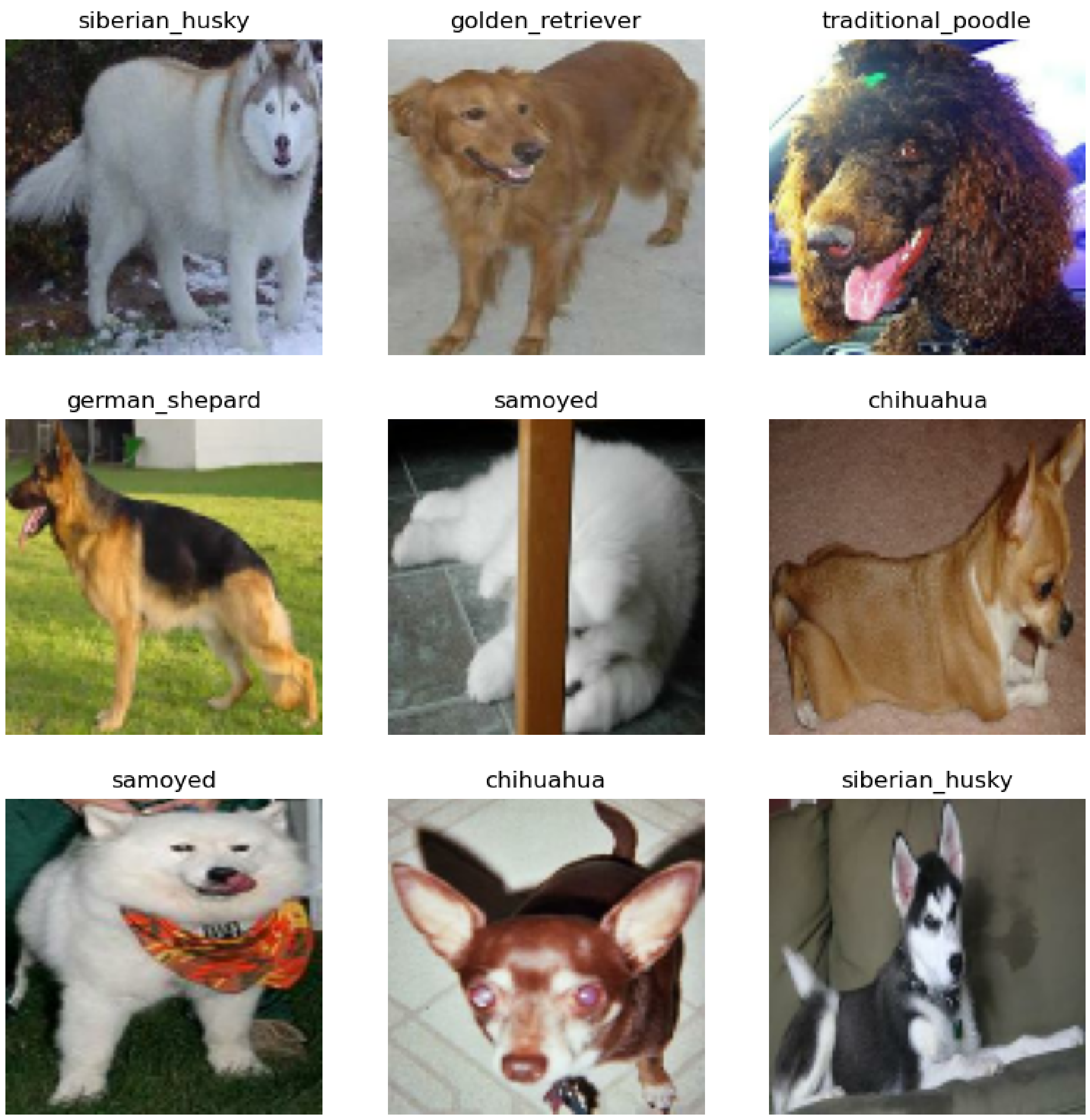
```
[152, 152, 150, 186, 152, 218, 192, 159]
```



1.4 Visualizing the Data

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

```python
[ ]: 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")
```



1.5 Configuting 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.cache().shuffle(1000).
  ↪prefetch(buffer_size=AUTOTUNE)
validation_data = validation_data.cache().prefetch(buffer_size=AUTOTUNE)
```

1.6 Standardizing the Data

As of now, the RGB values in the images are in the [0, 255] range, which isn't ideal for training a neural network. Making our input values smaller would be more ideal.

[ ]:
```
normalization_layer = layers.Rescaling(1./255)

normalized_ds = training_data.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]

# Notice the pixel values are now in `[0,1]`.
print(np.min(first_image), np.max(first_image))
```

```
0.0044721626 0.9765146
```

1.7 Building the Model

This Keras sequential model consists of 3 convolution blocks, each with its own max pooling layer. THe first convolutional layer has 16 filters, the second convolutional layer has 32 filters, and the third convolutional layer has 64 filters. A fully connected layer with 128 layers with a rectifier linear unit (ReLu) activation function sits on top of the convolution blocks.

[ ]:
```
num_classes = len(class_names)

model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])
```

1.8 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=tf.keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

1.9 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 100% and a validation accuracy of about 47%.

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

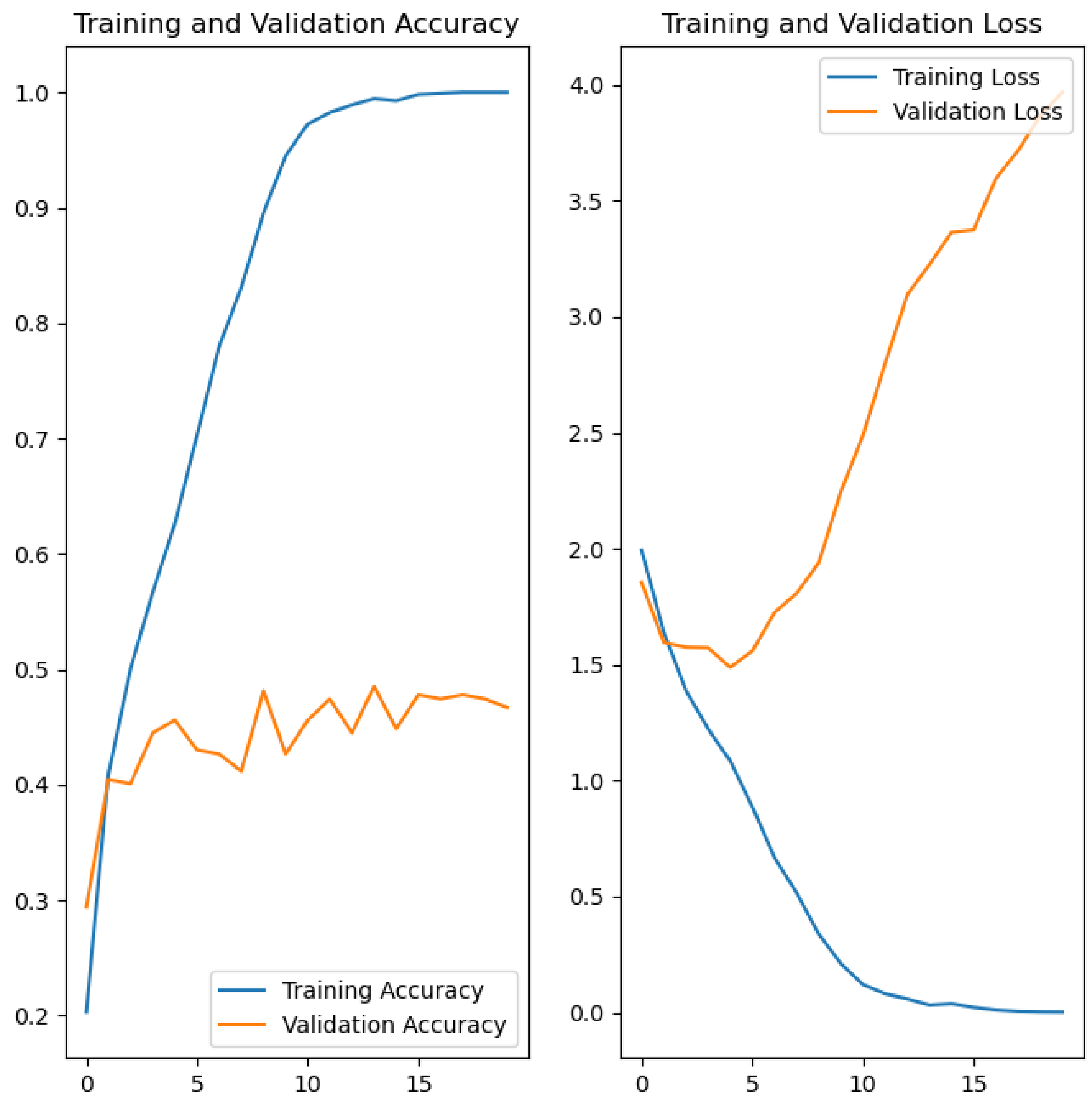
```
Epoch 1/20
35/35 [==============================] - 6s 164ms/step - loss: 1.9936 -
accuracy: 0.2029 - val_loss: 1.8539 - val_accuracy: 0.2941
Epoch 2/20
35/35 [==============================] - 6s 185ms/step - loss: 1.6392 -
accuracy: 0.4096 - val_loss: 1.5951 - val_accuracy: 0.4044
Epoch 3/20
35/35 [==============================] - 6s 161ms/step - loss: 1.3886 -
accuracy: 0.5005 - val_loss: 1.5746 - val_accuracy: 0.4007
Epoch 4/20
35/35 [==============================] - 6s 175ms/step - loss: 1.2230 -
accuracy: 0.5666 - val_loss: 1.5722 - val_accuracy: 0.4449
Epoch 5/20
35/35 [==============================] - 5s 155ms/step - loss: 1.0848 -
accuracy: 0.6263 - val_loss: 1.4882 - val_accuracy: 0.4559
Epoch 6/20
35/35 [==============================] - 6s 158ms/step - loss: 0.8867 -
accuracy: 0.7025 - val_loss: 1.5582 - val_accuracy: 0.4301
Epoch 7/20
35/35 [==============================] - 6s 180ms/step - loss: 0.6679 -
accuracy: 0.7796 - val_loss: 1.7238 - val_accuracy: 0.4265
Epoch 8/20
35/35 [==============================] - 6s 163ms/step - loss: 0.5158 -
accuracy: 0.8310 - val_loss: 1.8069 - val_accuracy: 0.4118
Epoch 9/20
35/35 [==============================] - 7s 194ms/step - loss: 0.3386 -
accuracy: 0.8953 - val_loss: 1.9386 - val_accuracy: 0.4816
Epoch 10/20
35/35 [==============================] - 6s 171ms/step - loss: 0.2107 -
accuracy: 0.9449 - val_loss: 2.2459 - val_accuracy: 0.4265
Epoch 11/20
```

```
35/35 [==============================] - 5s 148ms/step - loss: 0.1199 -
accuracy: 0.9725 - val_loss: 2.4893 - val_accuracy: 0.4559
Epoch 12/20
35/35 [==============================] - 7s 190ms/step - loss: 0.0802 -
accuracy: 0.9826 - val_loss: 2.7993 - val_accuracy: 0.4743
Epoch 13/20
35/35 [==============================] - 6s 159ms/step - loss: 0.0585 -
accuracy: 0.9890 - val_loss: 3.0956 - val_accuracy: 0.4449
Epoch 14/20
35/35 [==============================] - 5s 156ms/step - loss: 0.0319 -
accuracy: 0.9945 - val_loss: 3.2257 - val_accuracy: 0.4853
Epoch 15/20
35/35 [==============================] - 6s 168ms/step - loss: 0.0378 -
accuracy: 0.9927 - val_loss: 3.3643 - val_accuracy: 0.4485
Epoch 16/20
35/35 [==============================] - 6s 161ms/step - loss: 0.0218 -
accuracy: 0.9982 - val_loss: 3.3742 - val_accuracy: 0.4779
Epoch 17/20
35/35 [==============================] - 5s 151ms/step - loss: 0.0105 -
accuracy: 0.9991 - val_loss: 3.5955 - val_accuracy: 0.4743
Epoch 18/20
35/35 [==============================] - 6s 178ms/step - loss: 0.0042 -
accuracy: 1.0000 - val_loss: 3.7154 - val_accuracy: 0.4779
Epoch 19/20
35/35 [==============================] - 6s 158ms/step - loss: 0.0023 -
accuracy: 1.0000 - val_loss: 3.8657 - val_accuracy: 0.4743
Epoch 20/20
35/35 [==============================] - 6s 162ms/step - loss: 0.0017 -
accuracy: 1.0000 - val_loss: 3.9678 - val_accuracy: 0.4669
```

1.10 Visualizing the Results of Training

Using the accuracy and loss of the training and validation sets, we can create plots. The training accuracy increases linearly over time, whereas the validation accuracy stalls around 40%. The difference between the training accuracy and the validation accuracy is large, which is a sign of overfitting. We will address overfitting in the next two sections.

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

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

epochs_range = range(epochs)

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

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



1.11 Data Augmentation

Because this data set has a small number of training examples, there is 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 = keras.Sequential(
       [
         layers.RandomFlip("horizontal",
                           input_shape=(img_height,
                                        img_width,
                                        3)),
         layers.RandomRotation(0.1),
         layers.RandomZoom(0.1),
       ]
     )
```

```
[ ]: plt.figure(figsize=(10, 10))
     for images, _ in training_data.take(1):
       for i in range(9):
         augmented_images = data_augmentation(images)
         ax = plt.subplot(3, 3, i + 1)
         plt.imshow(augmented_images[0].numpy().astype("uint8"))
         plt.axis("off")
```

1.12 Dropout

Another technique that reduces overfitting is to introduce dropout regularization, which randomly drops out a number of output units from the layer during the training process. We built another model that utilizes both data augmentation and drops 20% of the output units randomly from the applied layer.

[ ]:
```
model = Sequential([
  data_augmentation,
  layers.Rescaling(1./255),

  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Dropout(0.2),

  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes, name="outputs")
])
```

The model can then be compiled.

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

Our model can now be trained.

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

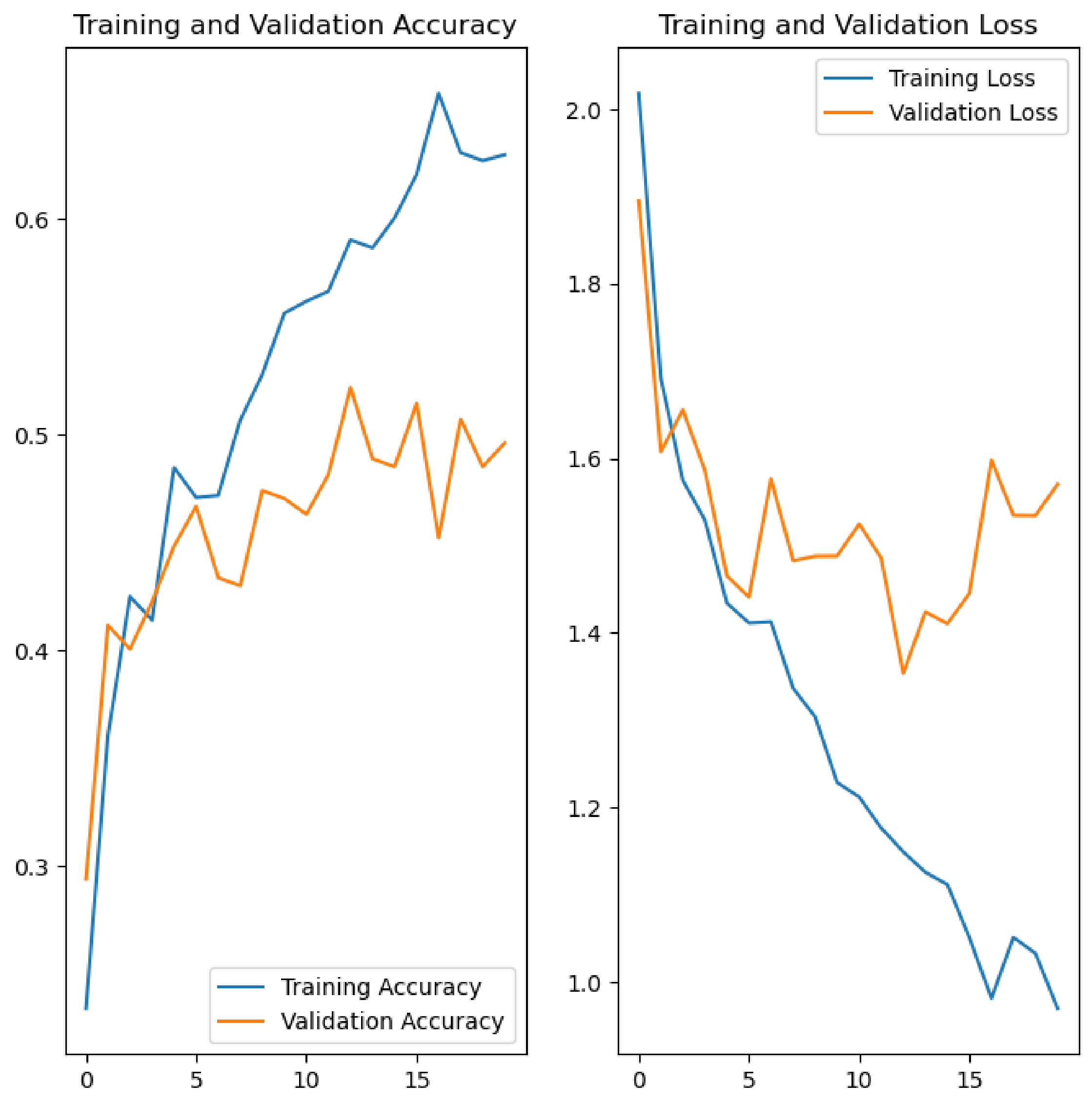
```
Epoch 1/20
35/35 [==============================] - 8s 186ms/step - loss: 2.0183 -
accuracy: 0.2342 - val_loss: 1.8952 - val_accuracy: 0.2941
Epoch 2/20
35/35 [==============================] - 9s 269ms/step - loss: 1.6919 -
accuracy: 0.3609 - val_loss: 1.6076 - val_accuracy: 0.4118
Epoch 3/20
35/35 [==============================] - 8s 221ms/step - loss: 1.5750 -
accuracy: 0.4252 - val_loss: 1.6557 - val_accuracy: 0.4007
Epoch 4/20
35/35 [==============================] - 8s 231ms/step - loss: 1.5293 -
accuracy: 0.4141 - val_loss: 1.5862 - val_accuracy: 0.4228
Epoch 5/20
35/35 [==============================] - 8s 224ms/step - loss: 1.4341 -
accuracy: 0.4848 - val_loss: 1.4653 - val_accuracy: 0.4485
Epoch 6/20
35/35 [==============================] - 8s 236ms/step - loss: 1.4113 -
accuracy: 0.4711 - val_loss: 1.4410 - val_accuracy: 0.4669
Epoch 7/20
```

```
35/35 [==============================] - 7s 209ms/step - loss: 1.4126 -
accuracy: 0.4720 - val_loss: 1.5766 - val_accuracy: 0.4338
Epoch 8/20
35/35 [==============================] - 7s 212ms/step - loss: 1.3366 -
accuracy: 0.5069 - val_loss: 1.4828 - val_accuracy: 0.4301
Epoch 9/20
35/35 [==============================] - 7s 192ms/step - loss: 1.3040 -
accuracy: 0.5280 - val_loss: 1.4879 - val_accuracy: 0.4743
Epoch 10/20
35/35 [==============================] - 8s 231ms/step - loss: 1.2287 -
accuracy: 0.5565 - val_loss: 1.4881 - val_accuracy: 0.4706
Epoch 11/20
35/35 [==============================] - 8s 216ms/step - loss: 1.2119 -
accuracy: 0.5620 - val_loss: 1.5249 - val_accuracy: 0.4632
Epoch 12/20
35/35 [==============================] - 7s 205ms/step - loss: 1.1763 -
accuracy: 0.5666 - val_loss: 1.4855 - val_accuracy: 0.4816
Epoch 13/20
35/35 [==============================] - 6s 184ms/step - loss: 1.1492 -
accuracy: 0.5904 - val_loss: 1.3533 - val_accuracy: 0.5221
Epoch 14/20
35/35 [==============================] - 7s 193ms/step - loss: 1.1257 -
accuracy: 0.5868 - val_loss: 1.4241 - val_accuracy: 0.4890
Epoch 15/20
35/35 [==============================] - 8s 217ms/step - loss: 1.1118 -
accuracy: 0.6006 - val_loss: 1.4105 - val_accuracy: 0.4853
Epoch 16/20
35/35 [==============================] - 7s 188ms/step - loss: 1.0509 -
accuracy: 0.6208 - val_loss: 1.4453 - val_accuracy: 0.5147
Epoch 17/20
35/35 [==============================] - 7s 190ms/step - loss: 0.9813 -
accuracy: 0.6584 - val_loss: 1.5979 - val_accuracy: 0.4522
Epoch 18/20
35/35 [==============================] - 6s 183ms/step - loss: 1.0509 -
accuracy: 0.6309 - val_loss: 1.5348 - val_accuracy: 0.5074
Epoch 19/20
35/35 [==============================] - 7s 202ms/step - loss: 1.0327 -
accuracy: 0.6272 - val_loss: 1.5345 - val_accuracy: 0.4853
Epoch 20/20
35/35 [==============================] - 6s 185ms/step - loss: 0.9696 -
accuracy: 0.6299 - val_loss: 1.5706 - val_accuracy: 0.4963
```

Now, we can create plots that show the accuracy and loss for the training and validation data. There is a smaller gap in the accuracy between the training and validation data, which means that less overfitting is present.

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

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

epochs_range = range(epochs)

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

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



1.13 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 97.5% accuracy.

[ ]:
```
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_height, img_width)
)

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 19ms/step
This image most likely belongs to german_shepard with a 97.51 percent
confidence.
```