image_classification_cnn

December 4, 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']
```

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
```

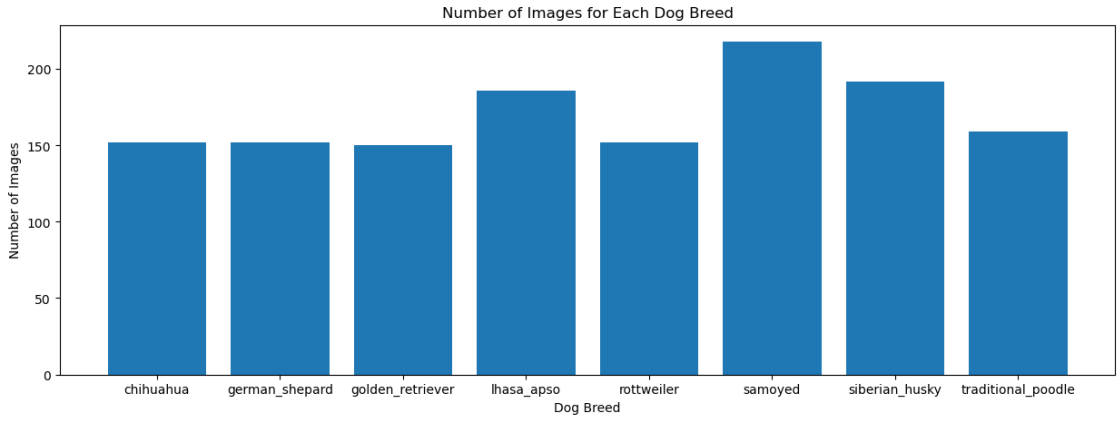
1.3 Graphing the Distribution of Target Classes

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()
```

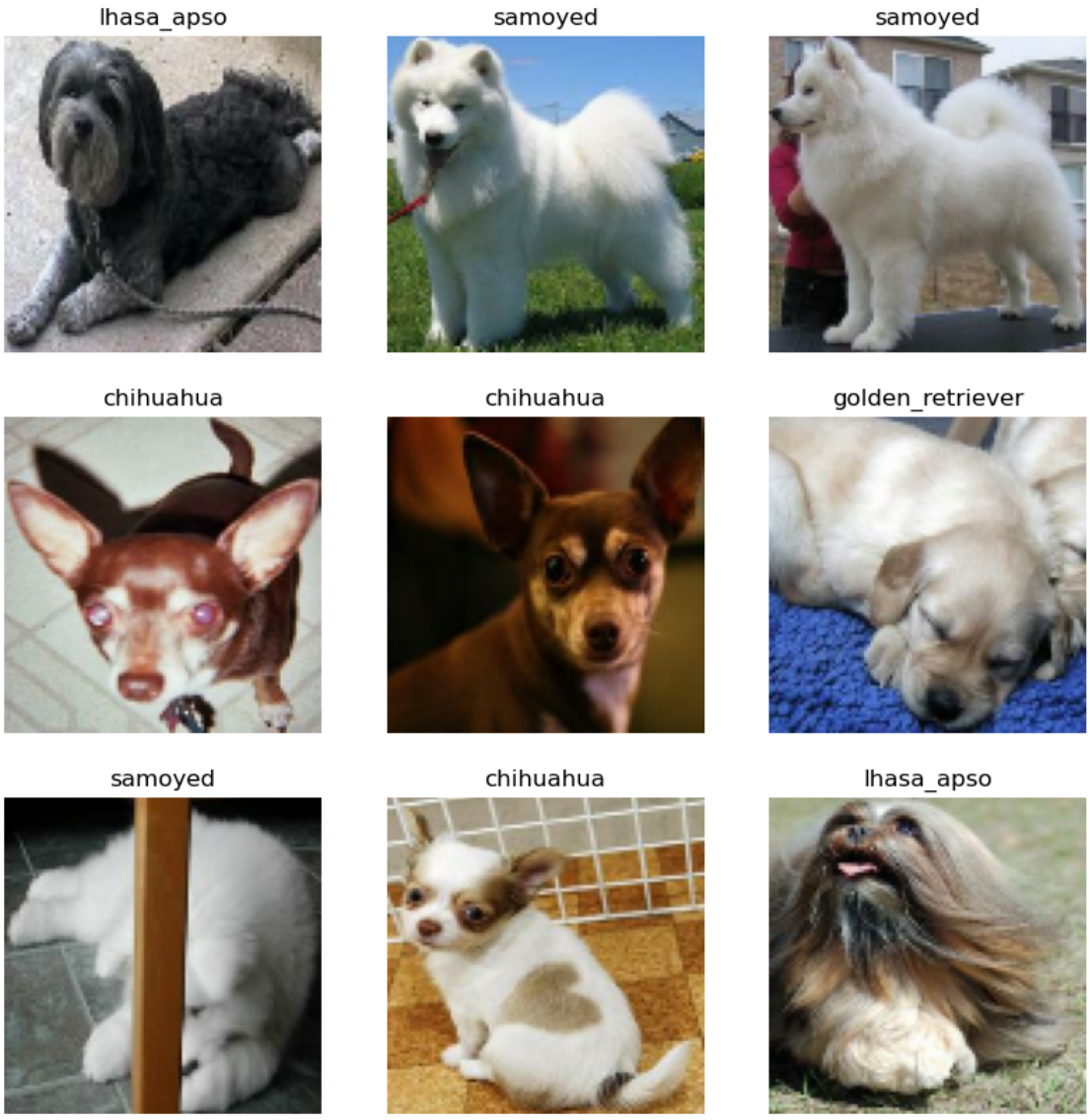
```
[152, 152, 150, 186, 152, 218, 192, 159]
```



1.4 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")
```



1.5 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.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))
```

```
WARNING:tensorflow:From c:\Users\iadit\anaconda3\envs\tf\lib\site-
packages\tensorflow\python\autograph\pyct\static_analysis\liveness.py:83:
Analyzer.lamba_check (from
tensorflow.python.autograph.pyct.static_analysis.liveness) is deprecated and
will be removed after 2023-09-23.
Instructions for updating:
Lambda fuctions will be no more assumed to be used in the statement where they
are used, or at least in the same block.
https://github.com/tensorflow/tensorflow/issues/56089
0.016923936 0.9860392
```

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
)
```

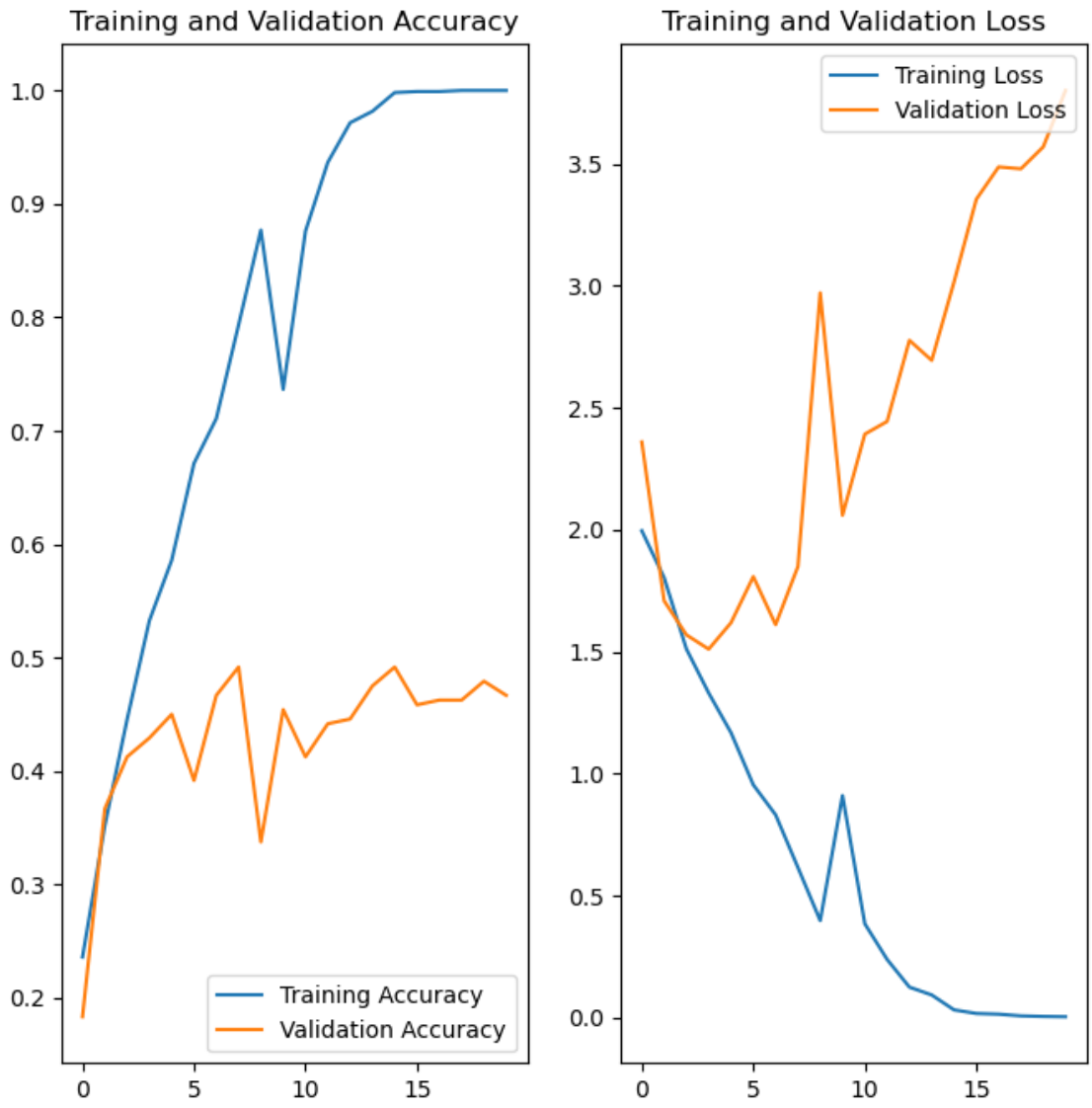
```
Epoch 1/20
35/35 [==============================] - 16s 421ms/step - loss: 1.9955 -
accuracy: 0.2360 - val_loss: 2.3593 - val_accuracy: 0.1833
Epoch 2/20
35/35 [==============================] - 12s 346ms/step - loss: 1.8033 -
accuracy: 0.3517 - val_loss: 1.7085 - val_accuracy: 0.3667
Epoch 3/20
35/35 [==============================] - 12s 348ms/step - loss: 1.5111 -
accuracy: 0.4454 - val_loss: 1.5689 - val_accuracy: 0.4125
Epoch 4/20
35/35 [==============================] - 12s 338ms/step - loss: 1.3305 -
accuracy: 0.5326 - val_loss: 1.5104 - val_accuracy: 0.4292
Epoch 5/20
```

```
35/35 [==============================] - 13s 354ms/step - loss: 1.1679 -
accuracy: 0.5859 - val_loss: 1.6187 - val_accuracy: 0.4500
Epoch 6/20
35/35 [==============================] - 12s 347ms/step - loss: 0.9542 -
accuracy: 0.6713 - val_loss: 1.8076 - val_accuracy: 0.3917
Epoch 7/20
35/35 [==============================] - 12s 342ms/step - loss: 0.8314 -
accuracy: 0.7107 - val_loss: 1.6107 - val_accuracy: 0.4667
Epoch 8/20
35/35 [==============================] - 12s 336ms/step - loss: 0.6156 -
accuracy: 0.7934 - val_loss: 1.8489 - val_accuracy: 0.4917
Epoch 9/20
35/35 [==============================] - 12s 340ms/step - loss: 0.3978 -
accuracy: 0.8770 - val_loss: 2.9704 - val_accuracy: 0.3375
Epoch 10/20
35/35 [==============================] - 13s 354ms/step - loss: 0.9105 -
accuracy: 0.7365 - val_loss: 2.0589 - val_accuracy: 0.4542
Epoch 11/20
35/35 [==============================] - 12s 342ms/step - loss: 0.3849 -
accuracy: 0.8760 - val_loss: 2.3915 - val_accuracy: 0.4125
Epoch 12/20
35/35 [==============================] - 12s 340ms/step - loss: 0.2387 -
accuracy: 0.9366 - val_loss: 2.4436 - val_accuracy: 0.4417
Epoch 13/20
35/35 [==============================] - 12s 341ms/step - loss: 0.1253 -
accuracy: 0.9715 - val_loss: 2.7759 - val_accuracy: 0.4458
Epoch 14/20
35/35 [==============================] - 13s 357ms/step - loss: 0.0932 -
accuracy: 0.9816 - val_loss: 2.6939 - val_accuracy: 0.4750
Epoch 15/20
35/35 [==============================] - 13s 353ms/step - loss: 0.0320 -
accuracy: 0.9982 - val_loss: 3.0151 - val_accuracy: 0.4917
Epoch 16/20
35/35 [==============================] - 13s 357ms/step - loss: 0.0174 -
accuracy: 0.9991 - val_loss: 3.3541 - val_accuracy: 0.4583
Epoch 17/20
35/35 [==============================] - 12s 351ms/step - loss: 0.0145 -
accuracy: 0.9991 - val_loss: 3.4866 - val_accuracy: 0.4625
Epoch 18/20
35/35 [==============================] - 13s 353ms/step - loss: 0.0074 -
accuracy: 1.0000 - val_loss: 3.4789 - val_accuracy: 0.4625
Epoch 19/20
35/35 [==============================] - 13s 372ms/step - loss: 0.0051 -
accuracy: 1.0000 - val_loss: 3.5687 - val_accuracy: 0.4792
Epoch 20/20
35/35 [==============================] - 13s 354ms/step - loss: 0.0039 -
accuracy: 1.0000 - val_loss: 3.7999 - val_accuracy: 0.4667
```

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 45%. 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

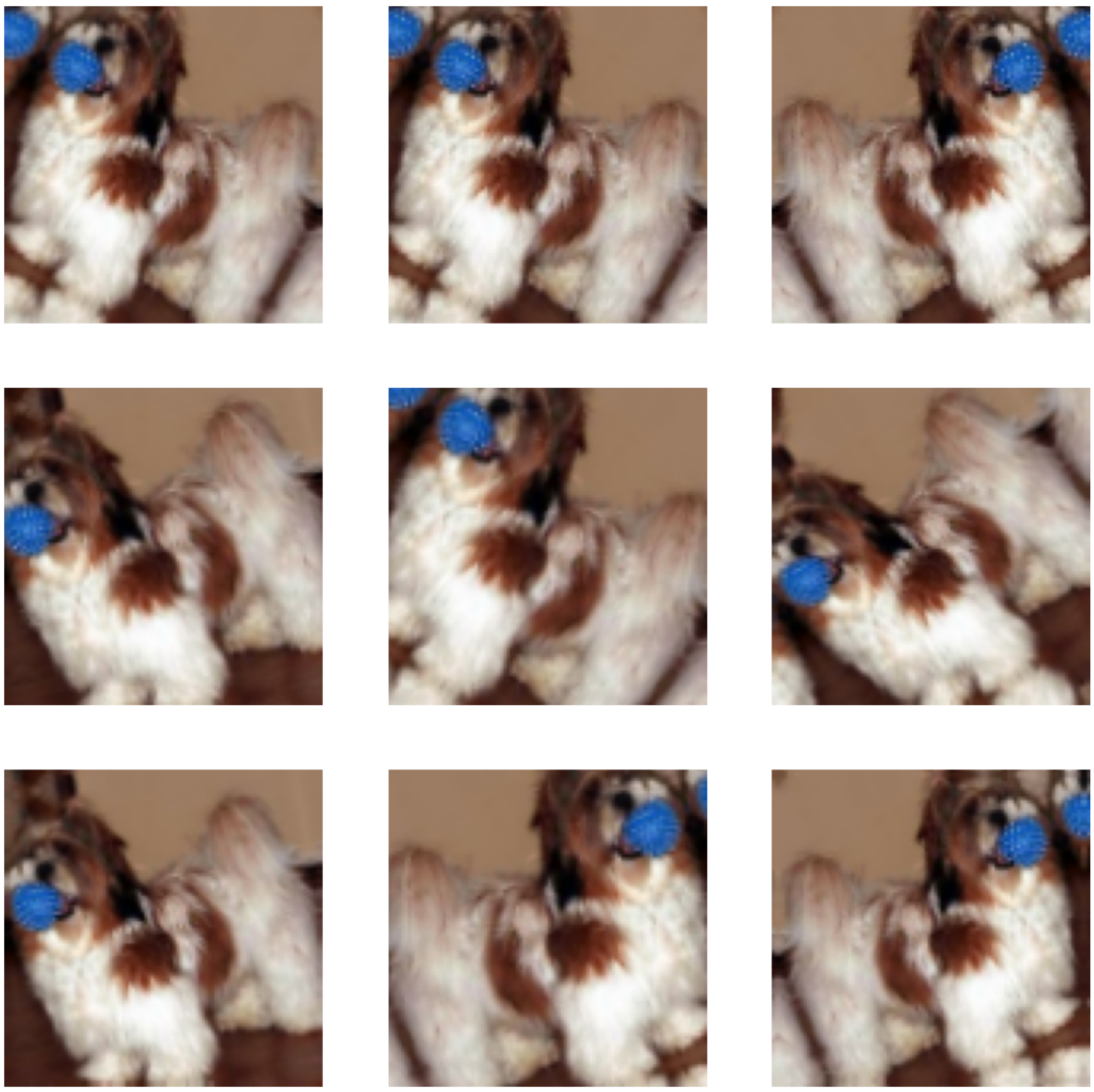
Because this data set has a small number of training examples, there is overfitting. Data augmentation tackles overfitting by generating more training data from the existing data through using random transformations that create images which look believable. An example of an image that has underwent data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

```
WARNING:tensorflow:5 out of the last 5 calls to <function pfor.<locals>.f at
0x0000025D96045F70> 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
0x0000025D96045DC0> 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.
```



1.12 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
)
```

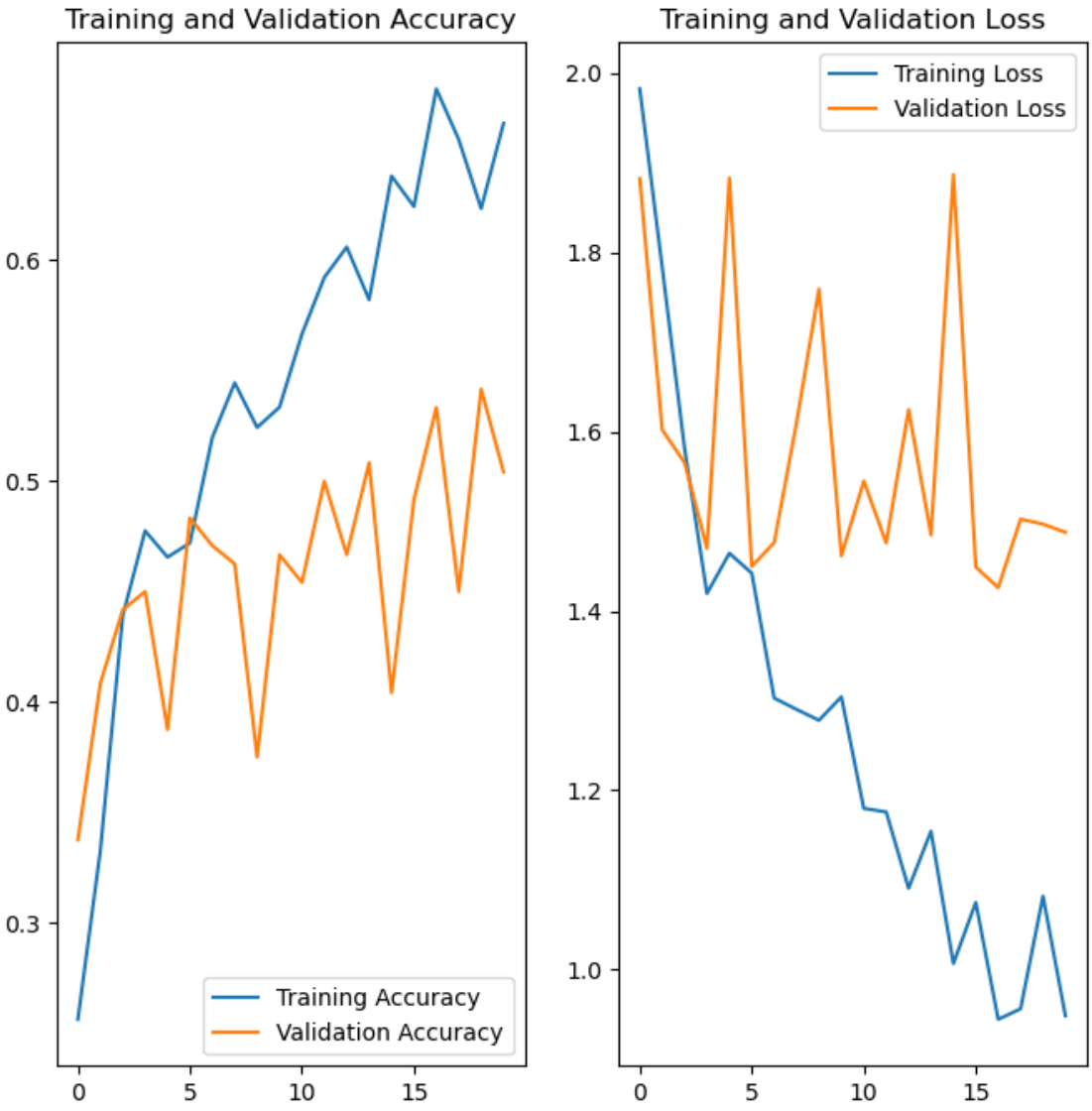
```
Epoch 1/20
35/35 [==============================] - 22s 446ms/step - loss: 1.9825 -
accuracy: 0.2562 - val_loss: 1.8824 - val_accuracy: 0.3375
Epoch 2/20
35/35 [==============================] - 14s 403ms/step - loss: 1.7841 -
accuracy: 0.3324 - val_loss: 1.6024 - val_accuracy: 0.4083
Epoch 3/20
35/35 [==============================] - 14s 392ms/step - loss: 1.5816 -
accuracy: 0.4389 - val_loss: 1.5657 - val_accuracy: 0.4417
Epoch 4/20
35/35 [==============================] - 15s 423ms/step - loss: 1.4197 -
accuracy: 0.4775 - val_loss: 1.4700 - val_accuracy: 0.4500
Epoch 5/20
35/35 [==============================] - 14s 380ms/step - loss: 1.4646 -
accuracy: 0.4656 - val_loss: 1.8831 - val_accuracy: 0.3875
Epoch 6/20
35/35 [==============================] - 14s 400ms/step - loss: 1.4423 -
accuracy: 0.4720 - val_loss: 1.4499 - val_accuracy: 0.4833
Epoch 7/20
```

```
35/35 [==============================] - 14s 394ms/step - loss: 1.3030 -
accuracy: 0.5197 - val_loss: 1.4764 - val_accuracy: 0.4708
Epoch 8/20
35/35 [==============================] - 14s 401ms/step - loss: 1.2903 -
accuracy: 0.5445 - val_loss: 1.6104 - val_accuracy: 0.4625
Epoch 9/20
35/35 [==============================] - 14s 383ms/step - loss: 1.2782 -
accuracy: 0.5243 - val_loss: 1.7594 - val_accuracy: 0.3750
Epoch 10/20
35/35 [==============================] - 13s 377ms/step - loss: 1.3044 -
accuracy: 0.5335 - val_loss: 1.4616 - val_accuracy: 0.4667
Epoch 11/20
35/35 [==============================] - 14s 407ms/step - loss: 1.1799 -
accuracy: 0.5666 - val_loss: 1.5450 - val_accuracy: 0.4542
Epoch 12/20
35/35 [==============================] - 14s 384ms/step - loss: 1.1760 -
accuracy: 0.5923 - val_loss: 1.4761 - val_accuracy: 0.5000
Epoch 13/20
35/35 [==============================] - 16s 454ms/step - loss: 1.0908 -
accuracy: 0.6061 - val_loss: 1.6248 - val_accuracy: 0.4667
Epoch 14/20
35/35 [==============================] - 17s 467ms/step - loss: 1.1544 -
accuracy: 0.5822 - val_loss: 1.4850 - val_accuracy: 0.5083
Epoch 15/20
35/35 [==============================] - 16s 431ms/step - loss: 1.0069 -
accuracy: 0.6382 - val_loss: 1.8869 - val_accuracy: 0.4042
Epoch 16/20
35/35 [==============================] - 15s 429ms/step - loss: 1.0749 -
accuracy: 0.6244 - val_loss: 1.4492 - val_accuracy: 0.4917
Epoch 17/20
35/35 [==============================] - 16s 437ms/step - loss: 0.9446 -
accuracy: 0.6777 - val_loss: 1.4262 - val_accuracy: 0.5333
Epoch 18/20
35/35 [==============================] - 15s 424ms/step - loss: 0.9564 -
accuracy: 0.6547 - val_loss: 1.5026 - val_accuracy: 0.4500
Epoch 19/20
35/35 [==============================] - 17s 481ms/step - loss: 1.0820 -
accuracy: 0.6235 - val_loss: 1.4970 - val_accuracy: 0.5417
Epoch 20/20
35/35 [==============================] - 19s 529ms/step - loss: 0.9486 -
accuracy: 0.6621 - val_loss: 1.4879 - val_accuracy: 0.5042
```

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.

[ ]:
```
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 Evaluating the Model

When evaluated on the test data, our model performed okay with an accuracy of 59%.

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

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

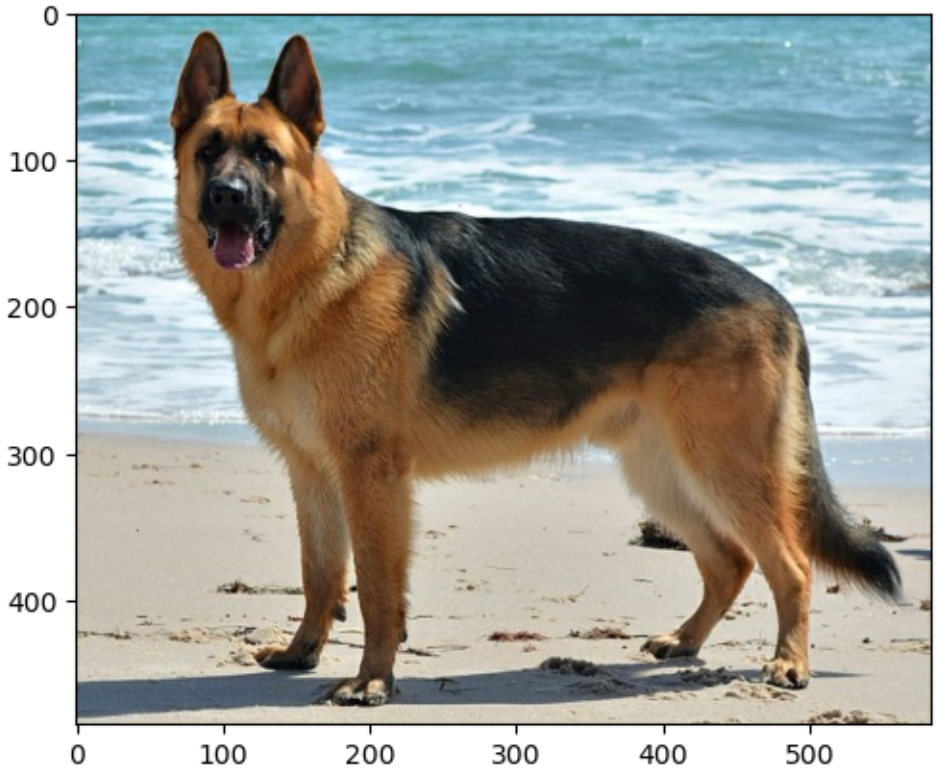
```
Test loss: 1.2461516857147217
Test accuracy: 0.59375
```

1.14 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 50.21% 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_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 144ms/step
This image most likely belongs to german_shepard with a 50.21 percent
confidence.
```


0
100
200
300
400
0
100
200
300
400
500