image_classification_sequential

December 4, 2022

1 Image Classification with a Sequential Model

I built an image classifier that identifies what breed a dog is based off of an image of the dog. I followed code samples from Dr. Karen Mazidi's *Machine Learning Handbook*.

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
```

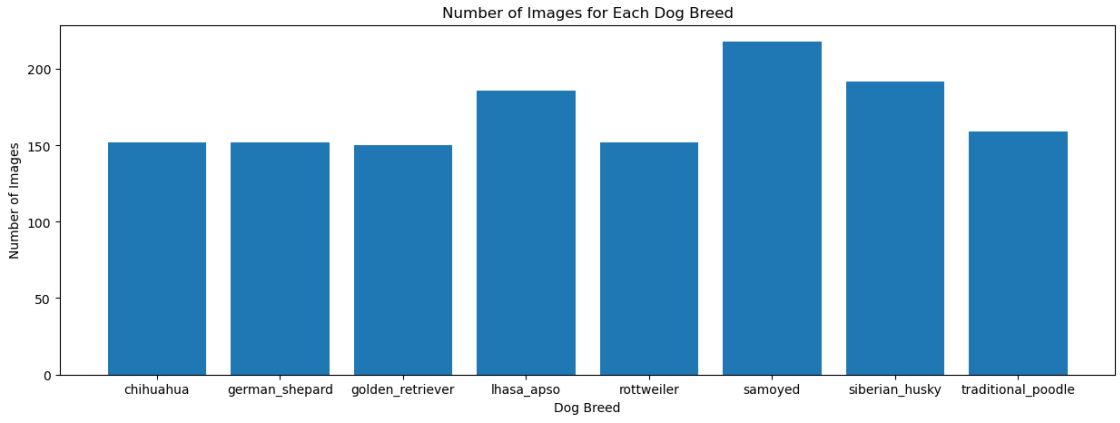
1.3 Graphing the Distribution of Target Classes

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()
```

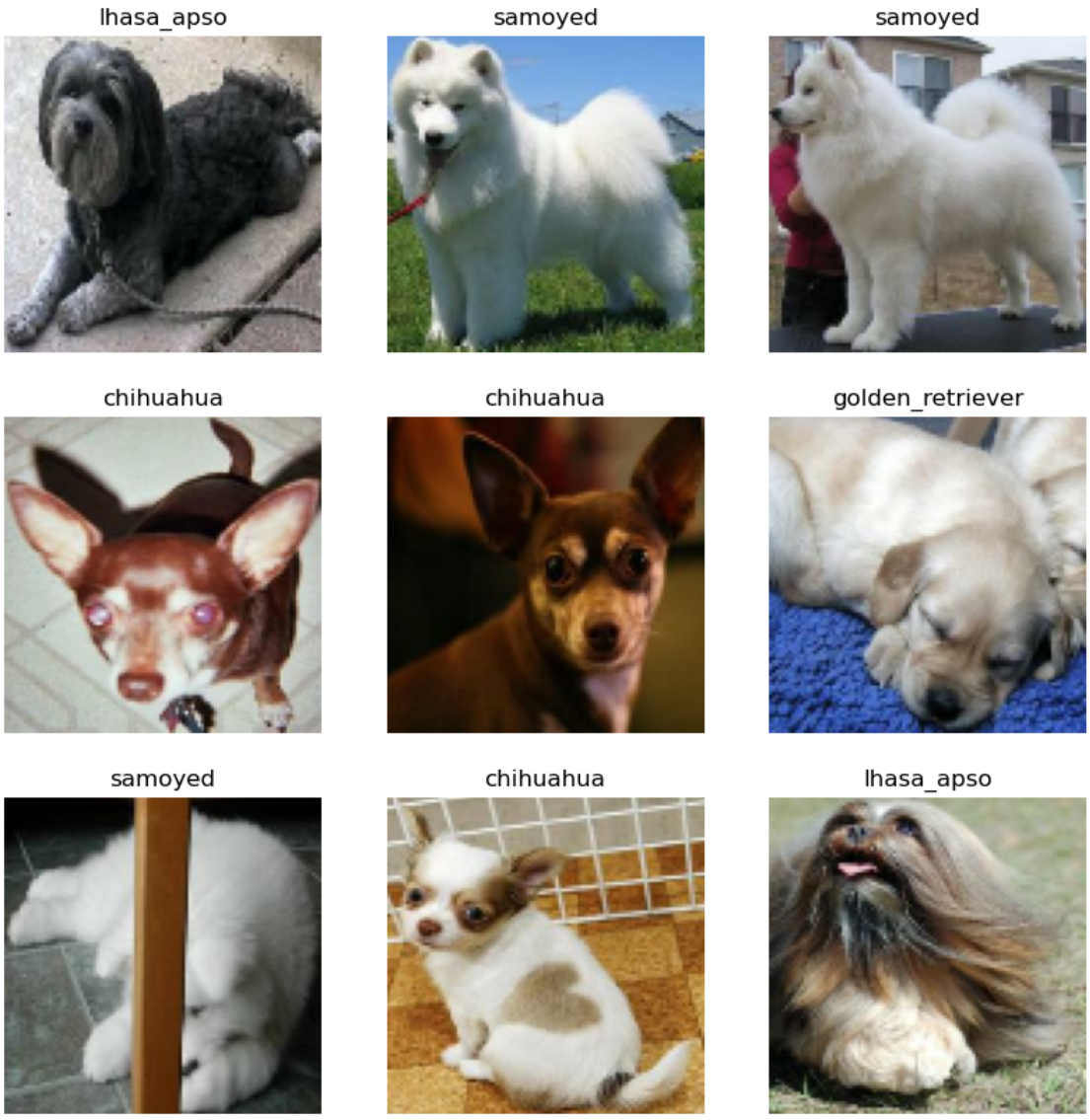
```
[152, 152, 150, 186, 152, 218, 192, 159]
```



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]

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
0.017221853 0.9235199
```

1.7 Building the Model

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

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(img_height, img_width, 3)),

  tf.keras.layers.Dense(512, activation="relu"),
  tf.keras.layers.Dropout(0.2),

  tf.keras.layers.Dense(512, activation="relu"),
  tf.keras.layers.Dropout(0.2),

  tf.keras.layers.Dense(num_classes, activation="softmax"),
```

```
])
```

1.8 Compiling the Model

The Root Mean Squared Propagation 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(loss="sparse_categorical_crossentropy",
optimizer="rmsprop",
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 16% and a validation accuracy of about 15%.

[ ]:
```
epochs = 20

history = model.fit(
    training_data,
    validation_data=validation_data,
    epochs=epochs
)
```

```
Epoch 1/20
35/35 [==============================] - 9s 251ms/step - loss: 6839.8345 -
accuracy: 0.1451 - val_loss: 222.5212 - val_accuracy: 0.1208
Epoch 2/20
35/35 [==============================] - 8s 226ms/step - loss: 124.9533 -
accuracy: 0.1423 - val_loss: 2.1340 - val_accuracy: 0.0750
Epoch 3/20
35/35 [==============================] - 7s 201ms/step - loss: 25.6826 -
accuracy: 0.1387 - val_loss: 2.1692 - val_accuracy: 0.1375
Epoch 4/20
35/35 [==============================] - 7s 194ms/step - loss: 2.1391 -
accuracy: 0.1433 - val_loss: 2.1689 - val_accuracy: 0.1375
Epoch 5/20
35/35 [==============================] - 6s 185ms/step - loss: 5.5681 -
accuracy: 0.1377 - val_loss: 2.1361 - val_accuracy: 0.1375
Epoch 6/20
35/35 [==============================] - 6s 186ms/step - loss: 2.0696 -
accuracy: 0.1387 - val_loss: 2.1362 - val_accuracy: 0.1375
Epoch 7/20
35/35 [==============================] - 7s 193ms/step - loss: 2.0804 -
accuracy: 0.1579 - val_loss: 2.0868 - val_accuracy: 0.1625
Epoch 8/20
35/35 [==============================] - 7s 191ms/step - loss: 2.0681 -
```

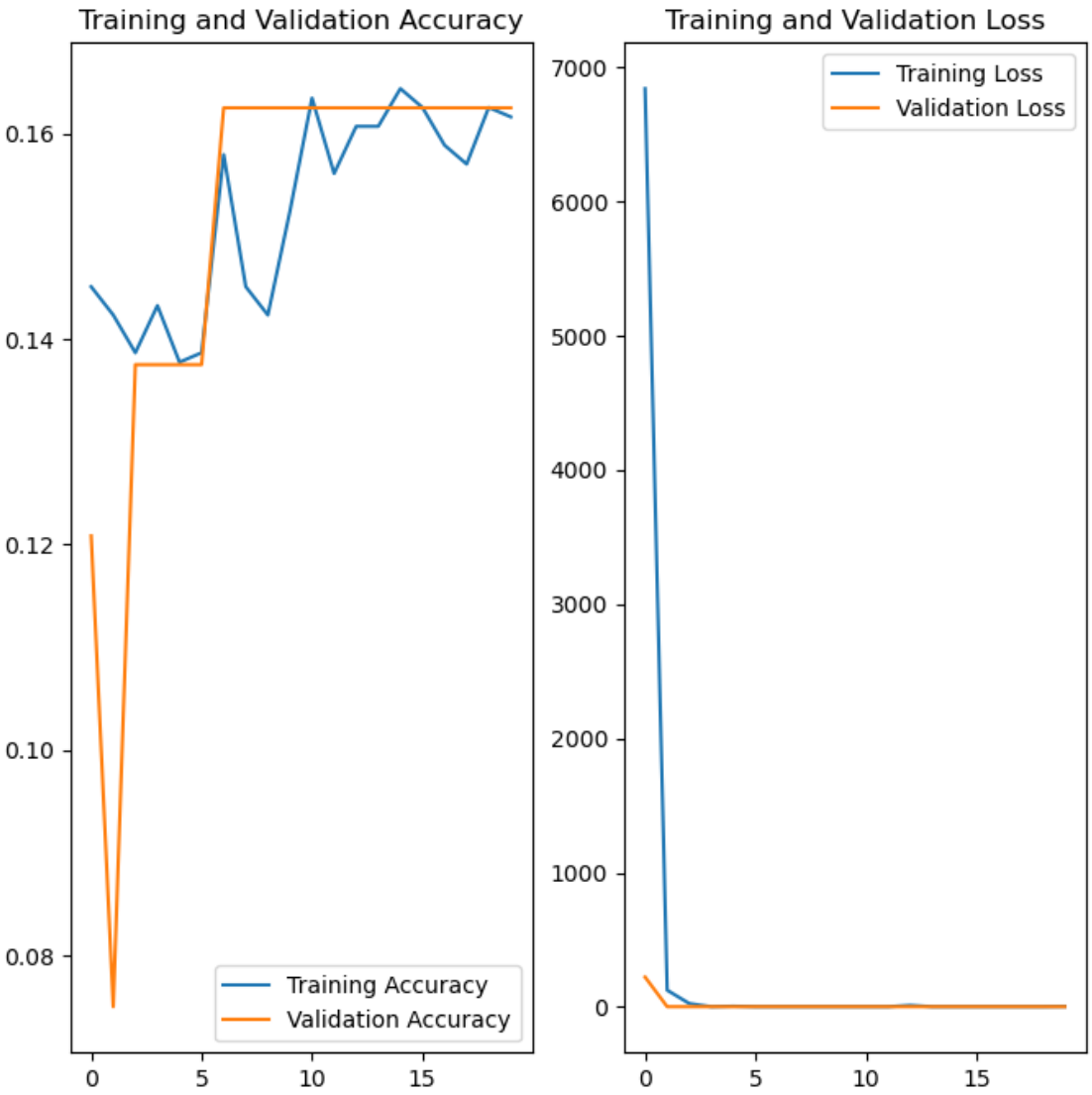
```
accuracy: 0.1451 - val_loss: 2.0906 - val_accuracy: 0.1625
Epoch 9/20
35/35 [==============================] - 6s 184ms/step - loss: 2.0693 -
accuracy: 0.1423 - val_loss: 2.0937 - val_accuracy: 0.1625
Epoch 10/20
35/35 [==============================] - 7s 197ms/step - loss: 2.0682 -
accuracy: 0.1524 - val_loss: 2.0983 - val_accuracy: 0.1625
Epoch 11/20
35/35 [==============================] - 7s 200ms/step - loss: 2.0678 -
accuracy: 0.1635 - val_loss: 2.0976 - val_accuracy: 0.1625
Epoch 12/20
35/35 [==============================] - 7s 193ms/step - loss: 2.0682 -
accuracy: 0.1561 - val_loss: 2.1029 - val_accuracy: 0.1625
Epoch 13/20
35/35 [==============================] - 7s 192ms/step - loss: 12.0227 -
accuracy: 0.1607 - val_loss: 2.1042 - val_accuracy: 0.1625
Epoch 14/20
35/35 [==============================] - 6s 185ms/step - loss: 2.0719 -
accuracy: 0.1607 - val_loss: 2.1041 - val_accuracy: 0.1625
Epoch 15/20
35/35 [==============================] - 7s 197ms/step - loss: 2.0719 -
accuracy: 0.1644 - val_loss: 2.1029 - val_accuracy: 0.1625
Epoch 16/20
35/35 [==============================] - 7s 190ms/step - loss: 2.0700 -
accuracy: 0.1625 - val_loss: 2.1020 - val_accuracy: 0.1625
Epoch 17/20
35/35 [==============================] - 7s 190ms/step - loss: 2.0700 -
accuracy: 0.1589 - val_loss: 2.1016 - val_accuracy: 0.1625
Epoch 18/20
35/35 [==============================] - 7s 198ms/step - loss: 2.0706 -
accuracy: 0.1570 - val_loss: 2.1023 - val_accuracy: 0.1625
Epoch 19/20
35/35 [==============================] - 7s 206ms/step - loss: 2.0722 -
accuracy: 0.1625 - val_loss: 2.1016 - val_accuracy: 0.1625
Epoch 20/20
35/35 [==============================] - 7s 195ms/step - loss: 3.2561 -
accuracy: 0.1616 - val_loss: 2.0962 - val_accuracy: 0.1625
```

1.10 Visualizing the Results of Training

Both the training accuracy and the validation accuracy are similar, which suggests that there is little overfitting. However, both the training data and the validation data have quite a low accuracy rate.

[ ]:
```
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 Evaluating the Model

When evaluated on the test data, our model performed okay with an accuracy of 16%.

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

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

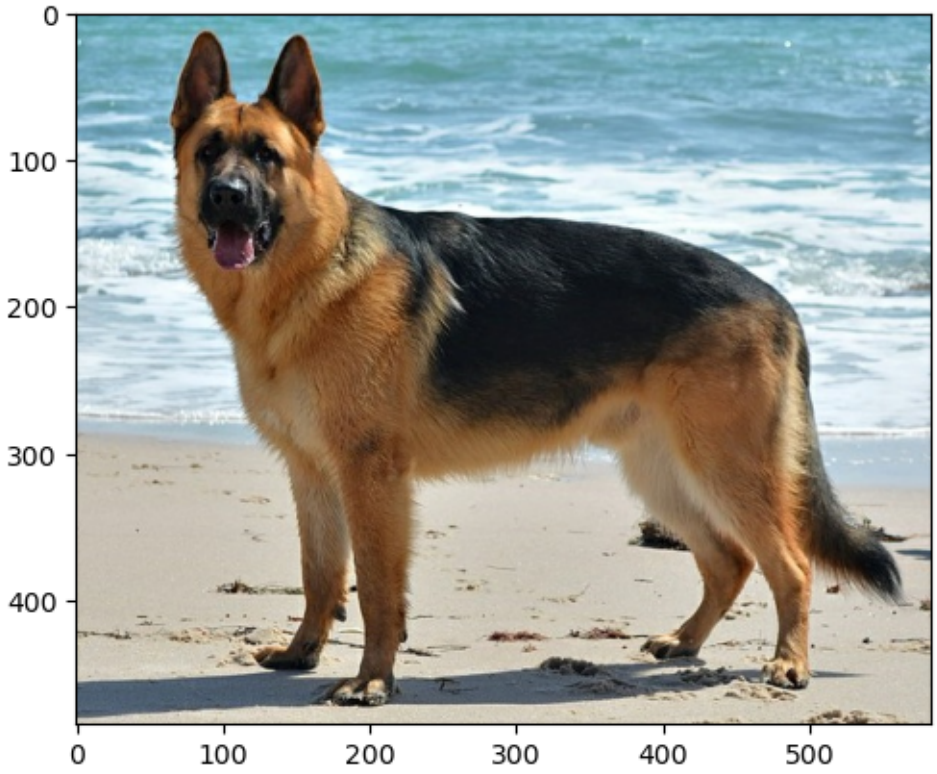
```
Test loss: 2.031654119491577
Test accuracy: 0.15625
```

1.12 Predicting on New Data

Let's classify an image that the neural network hasn't seen before. The neural network incorrectly classified the dog as a samoyed with 12.93% 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_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 121ms/step
This image most likely belongs to samoyed with a 12.93 percent confidence.
```


0
100
200
300
400
0
100
200
300
400
500