Dogs and Cats Dataset Model

Josh Cassada, Colin Kligge,
Parker Ray

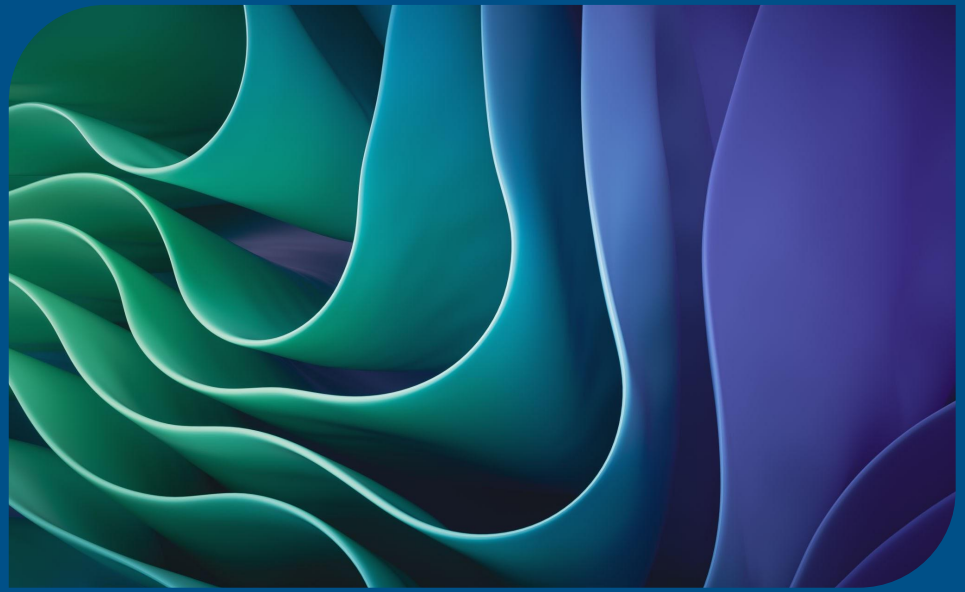
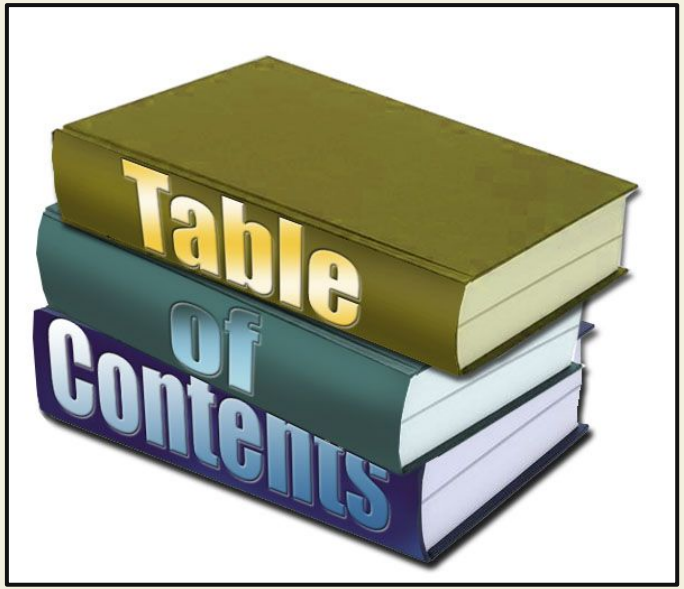


Table of Contents

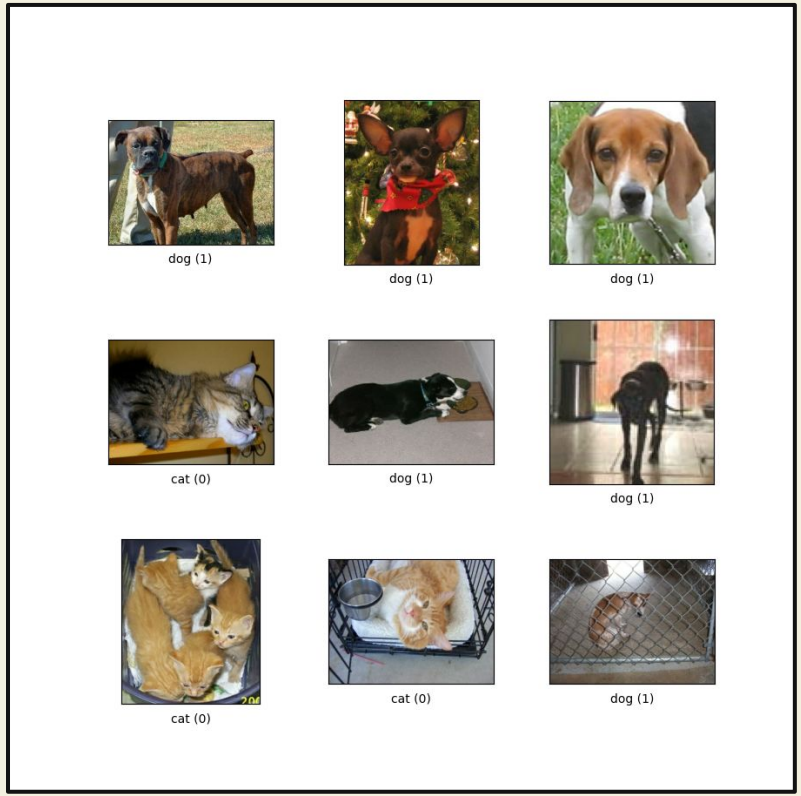
- Dataset
- Problem Statement
- Importing dataset
- Removing images
- Original code
- Innovations
- Contributions
- Questions



Dataset

Cats vs Dogs

- 25,000 labeled images in one training split
    - 1783 images were corrupted and removed
- 2 classes, Cat or Dog
- Varying resolutions
- Image shape is (Xres, Yres, 3)
    - 3 indicates 3 color channels (rgb)



Problem Statement

Adjust model hyperparameters such as batch size and number of epochs to improve the accuracy of the MobileNetV2 CNN model on the validation data while also considering time.

Importing Modules and Dataset

```
import tensorflow as tf
import matplotlib.pyplot as plt
import os
import numpy as np
```

Tf - used for training the models and is used through a majority of the dataset

Plt - used for making visualizations with the dataset

Os - used for file pathing and creating directories

Np - used for array making and features

```
import tensorflow_datasets as tfds

data, ds_info = tfds.load(name = "cats_vs_dogs",
                                          data_dir = "/content/drive2/MyDrive/food-101_data",
                                         shuffle_files = True,
                                         as_supervised = True, # returns data set in tuple form
                                          with_info = True) # returns metadata

def preprocess_image(image, label):
  image = tf.image.resize(image, [128,128], preserve_aspect_ratio=True) # scales images down or up to 128x128 while preserving the aspect ratio
  image = tf.image.resize_with_crop_or_pad(image, 128, 128) # adds padding to any image that is not 128x128 so it fits the dimensionality requirements
  image = tf.cast(image, tf.float32) / 255 # normalizes images so color value pixel range is between 0 and 1
  label = tf.one_hot(label, 2) # One hot encodes the label, 1 indicates the image belongs to the class, the depth is 2 whis is the number of classes
  return image, label

dataset = data['train'].map(preprocess_image, num_parallel_calls=tf.data.AUTOTUNE).cache()
```

Autotune helps speed up data processing by determining the optimal parallel calls

Removing Images

```
WARNING:absl:1738 images were corrupted and were skipped
```

```
Subsequent calls will reuse this data.
```

The corrupted images were removed from the dataset to prevent errors in the model, improving the training and efficiency of the model, and overall create better quality. The importing of the dataset was modified to be able to be recalled once and not be downloaded every time.

Visualizing the Data and Image Features

```
for image, label in train_data.take(1):
    print(f"Image shape: {image.shape}, Label: {label}")

Image shape: (8, 128, 128, 3), Label: [[1. 0.]
```

The data was checked to see if the labels and size were working correctly.

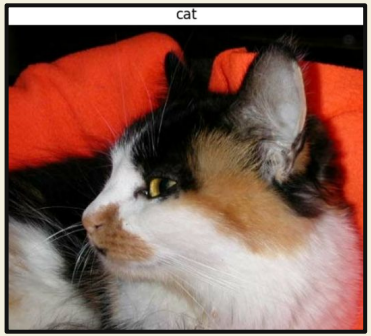
```
# confirm uniform color channels
check_color_channels = data['train'].take(1000)
for image in check_color_channels:
  if image[0].shape[2] != 3:
    print("Image has different amount of color channels")
```

Then from the dataset, a batch will be created with 1000 images and checks if each of those images have red, blue and green color channels. If it does not, it is most likely on a grayscale.

Visualizing the Data and Image Features

```
# View an image
import matplotlib.pyplot as plt

my_image = data['train'].take(1) # retrieves the first sample from a random class
for aspect in my_image:
  img = aspect[0]
  label = aspect[1]

# View image class
if label == 1:
  label = 'dog'
else:
  label = 'cat'
print(label)
```

```
# show the image and the class it belongs to
plt.imshow(img)
plt.title(label)
plt.axis("off")
```



It takes one image from the sample and displays it.

- aspect[0] refers to the image data itself (the array)

- aspect[1] refers to the label associated with the image (the class label)

The image is then classified and given a label.

Finally, the image is checked by showing its classification.

Displaying Random Image

```
def rand_image(dataset):
  my_image = dataset.take(1) # retrieves the first sample from a random class
  for image, label in my_image:
    img = image[0]
    label = label[0]

  index = np.argmax(label)


  if index == 0:
    label = 'cat'
  else:
    label = 'dog'

  plt.imshow(img)
  plt.title(label)
  plt.axis("off")
```

```
rand_image(train_data)
```

One image is retrieved from the dataset and the loop iterates the sample.

Image[0] and label[0] then retrieve the first image and label in the batch.

Then argmax retrieves the index used for the if-else statement to determine if the label is either cat or dog.

Finally, using the matplotlib function, the image is shown, the label is added, and the axis points are removed from the sample.

Each time the function is ran, a new image is chosen from the dataset.

Original Code

```
from tensorflow.keras.applications import MobileNetV2

base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
```

```
# Create model
from tensorflow.keras import layers, models
model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(128, activation='relu'),
    layers.Dense(2, activation='softmax') # 2 is one for each class
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])


# Fit the model
history = model.fit(train_data,
                    validation_data = test_data,
                      epochs = 5)
```

The model used is MobileNetv2, being used as a neural network, which runs efficiently and provides high accuracy, especially for image classification. Weights help load pre-trained weight from ImageNet, a visual database. It also inputs the images shape, 128x128.

The model then is turned into a sequence (layers stacked upon each other) and GlobalAveragePooling2D reduced dimensionality. Relu helps the model learn more complex pattern and introduces non-linearity. Softmax turns the output into a probability distribution

Model compile then ensures the training and deep learning of the model, being measured by its accuracy.

Finally in fitting the model, the amount of epochs can be changed where more usually leads to better performance. However too many can cause problems and make the model not be able to learn any patterns.

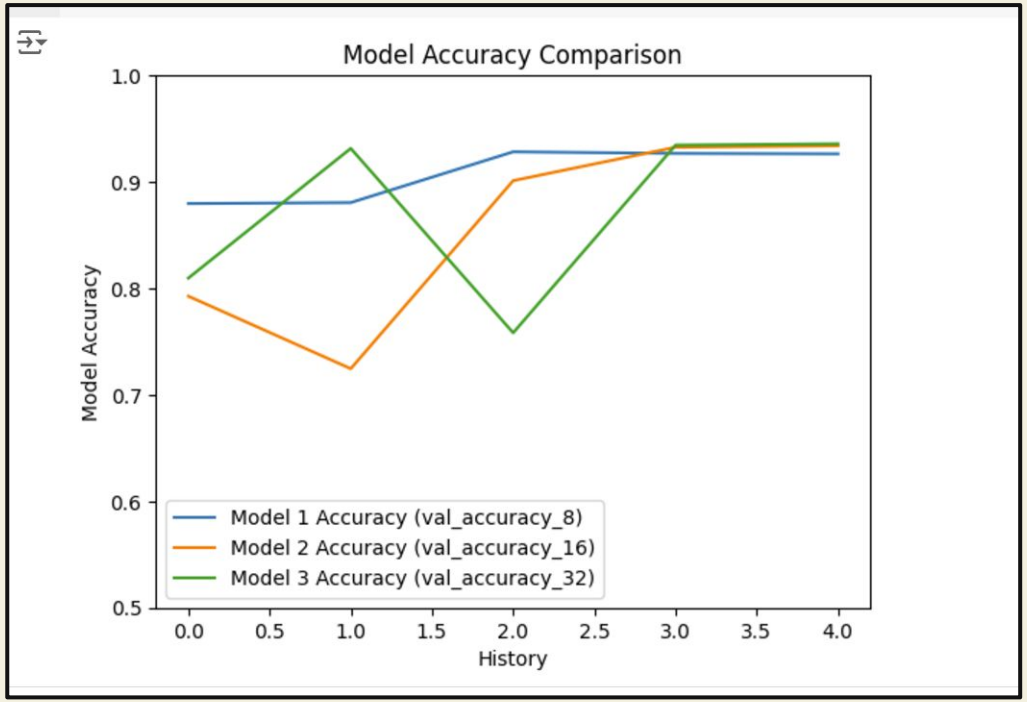
Changing the batch sizes

```python
import matplotlib.pyplot as plt

val_accuracy_8 = [0.8796, 0.8805, 0.9282, 0.9267, 0.9263]
val_accuracy_16 = [0.7926, 0.7245, 0.9011, 0.9325, 0.9338]
val_accuracy_32 = [0.8096, 0.9314, 0.7582, 0.9345, 0.9357]

# Plot final accuracy
def compare_results(val_accuracy_8, val_accuracy_16, val_accuracy_32):




  # Plot accuracy comparison
  plt.plot(val_accuracy_8, label="Model 1 Accuracy (val_accuracy_8)")
  plt.plot(val_accuracy_16, label="Model 2 Accuracy (val_accuracy_16)")
  plt.plot(val_accuracy_32, label="Model 3 Accuracy (val_accuracy_32)")
  plt.title("Model Accuracy Comparison")
  plt.xlabel("History")
  plt.ylabel("Model Accuracy")
  plt.ylim(0.5, 1)
  plt.legend()
  plt.show()


compare_results(val_accuracy_8, val_accuracy_16, val_accuracy_32)
```

This table displays the validation accuracies for the model trained over 5 epochs with different batch sizes: 8, 16, and 32.

- When looking at batch size 8 you can see that the validation accuracy starts relatively high (0.8796) and improves steadily until stabilizing at around 0.9263 by the 5th epoch.
- When looking at batch size 16 the validation accuracy starts lower (0.7926) compared to batch size 8 but catches up and even exceeds it by the 4th and 5th epochs (reaching 0.9338).
- When looking at batch size 32 the validation accuracy fluctuates more initially (0.8096 → 0.9314 → 0.7582), indicating some instability during training.

Changing the batch sizes

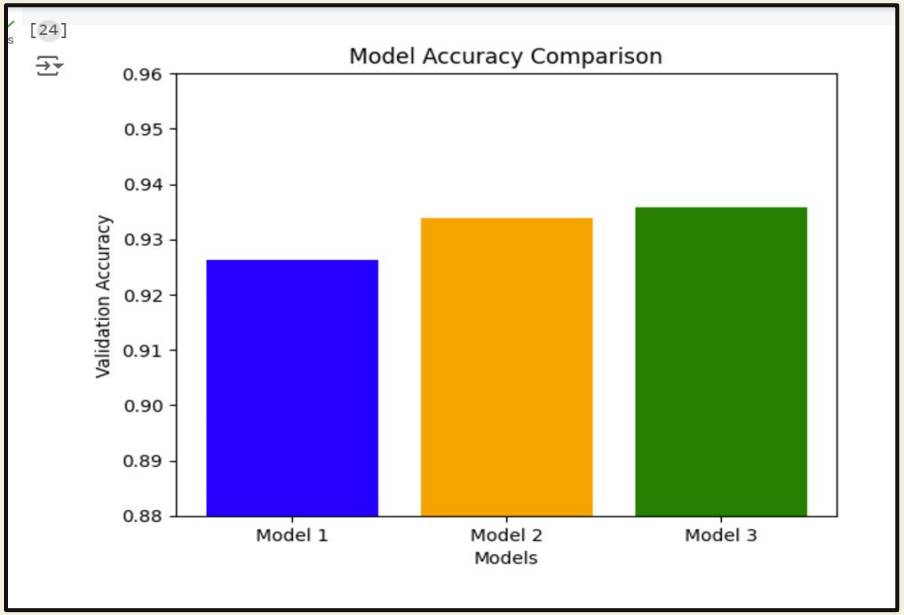


Key takeaways from looking at this graph and comparing batch sizes:

Batch Size Trade-offs:

- Smaller batch sizes tend to perform better initially because of finer updates to model weights, but they take longer to train.
- Larger batch sizes are computationally faster but may struggle with convergence or stability, as seen in the initial fluctuations for batch size 32.

```python
[24] import matplotlib.pyplot as plt

val_accuracy_8 = [0.9263]
val_accuracy_16 = [0.9338]
val_accuracy_32 = [0.9357]

def compare_results(val_accuracy_8, val_accuracy_16, val_accuracy_32):
    x_labels = ["Model 1", "Model 2", "Model 3"]
    accuracies = [val_accuracy_8[0], val_accuracy_16[0], val_accuracy_32[0]]

    plt.bar(x_labels, accuracies, color=['blue', 'orange', 'green'])
    plt.title("Model Accuracy Comparison")
    plt.xlabel("Models")
    plt.ylabel("Validation Accuracy")
    plt.ylim(0.88, .96)
    plt.show()

# Call the function
compare_results(val_accuracy_8, val_accuracy_16, val_accuracy_32)
```



Overview

- This slide compares the validation accuracy achieved by three models with batch sizes 8, 16, and 32 at the 5th epoch.
- The bar chart shows the final validation accuracy for each batch size after training for 5 epochs.

Performance Comparison:

- Validation accuracy improves slightly as the batch size increases, with batch size **32** achieving the best result.

What is the point of testing different batch sizes?

```
# Fit the model
history = model.fit(train_data,
                    validation_data = test_data,
                      epochs = 5)
```

```
Epoch 1/5
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 182s 51ms/step - a
Epoch 2/5
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 46s 20ms/step - ac
Epoch 3/5
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 44s 19ms/step - ac
Epoch 4/5
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 45s 19ms/step - ac
Epoch 5/5
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 46s 20ms/step - ac
```

```
[ ] # Fit the model
    history = model.fit(train_data,
                        validation_data = test_data,
                          epochs = 5)
```

```
Epoch 1/5
1164/1164 ━━━━━━━━━━━━━━━━━━━━ 151s 90ms/step -
Epoch 2/5
1164/1164 ━━━━━━━━━━━━━━━━━━━━ 34s 30ms/step - a
Epoch 3/5
1164/1164 ━━━━━━━━━━━━━━━━━━━━ 34s 29ms/step - a
Epoch 4/5
1164/1164 ━━━━━━━━━━━━━━━━━━━━ 35s 30ms/step - a
Epoch 5/5
1164/1164 ━━━━━━━━━━━━━━━━━━━━ 34s 29ms/step - a
```

```
[ ] # Fit the model
    history = model.fit(train_data,
                        validation_data = test_data
                          epochs = 5)
```

```
Epoch 1/5
582/582 ━━━━━━━━━━━━━━━━━━━━ 167s 176ms/step -
Epoch 2/5
582/582 ━━━━━━━━━━━━━━━━━━━━ 26s 45ms/step - ac
Epoch 3/5
582/582 ━━━━━━━━━━━━━━━━━━━━ 26s 45ms/step - ac
Epoch 4/5
582/582 ━━━━━━━━━━━━━━━━━━━━ 42s 46ms/step - ac
Epoch 5/5
582/582 ━━━━━━━━━━━━━━━━━━━━ 25s 43ms/step - ac
```

Testing different batch sizes is a crucial part of optimizing machine learning models because the batch size can significantly impact training speed, model performance, and stability.

Smaller batch size gives more frequent updates to the weight, leading to faster convergence and less use of the memory.

Larger batch sizes are generally faster because of less updates to the model.

Changing the Number of Epochs

```
# Fit the model
history = model.fit(train_data,
                    validation_data = test_data,
                      epochs = 1)
```

```
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 190s 52ms/step - accuracy: 0.8553 - loss: 0.3734 - val_accuracy: 0.8470 - val_loss: 0.8801
```

```
# Train longer
history2 = model2.fit(train_data,
                      validation_data = test_data,
                       epochs = 5)
```

```
Epoch 1/5
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 108s 29ms/step - accuracy: 0.9022 - loss: 0.2649 - val_accuracy: 0.6462 - val_loss: 1.1113
Epoch 2/5
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 47s 20ms/step - accuracy: 0.9284 - loss: 0.1834 - val_accuracy: 0.8364 - val_loss: 1.2654
Epoch 3/5
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 48s 21ms/step - accuracy: 0.9440 - loss: 0.1478 - val_accuracy: 0.9263 - val_loss: 0.4635
Epoch 4/5
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 46s 20ms/step - accuracy: 0.9575 - loss: 0.1119 - val_accuracy: 0.9329 - val_loss: 0.2385
Epoch 5/5
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 45s 20ms/step - accuracy: 0.9598 - loss: 0.1040 - val_accuracy: 0.8390 - val_loss: 0.8804
```

```
# Train even longer
history3 = model3.fit(train_data,
                      validation_data = test_data,
                       epochs = 10) # Increase to 10
```

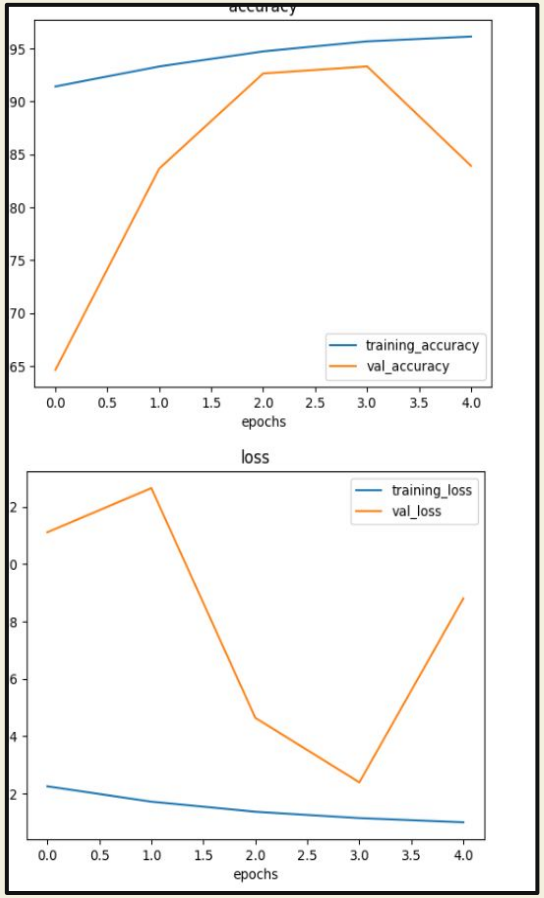
```
Epoch 1/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 156s 31ms/step - accuracy: 0.9664 - loss: 0.0898 - val_accuracy: 0.9381 - val_loss: 0.2256
Epoch 2/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 46s 20ms/step - accuracy: 0.9741 - loss: 0.0666 - val_accuracy: 0.9448 - val_loss: 0.1986
Epoch 3/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 47s 20ms/step - accuracy: 0.9729 - loss: 0.0709 - val_accuracy: 0.9310 - val_loss: 0.2735
Epoch 4/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 46s 20ms/step - accuracy: 0.9753 - loss: 0.0633 - val_accuracy: 0.9325 - val_loss: 0.2697
Epoch 5/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 46s 20ms/step - accuracy: 0.9810 - loss: 0.0547 - val_accuracy: 0.9001 - val_loss: 0.4724
Epoch 6/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 49s 21ms/step - accuracy: 0.9820 - loss: 0.0491 - val_accuracy: 0.9430 - val_loss: 0.2637
Epoch 7/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 49s 21ms/step - accuracy: 0.9836 - loss: 0.0486 - val_accuracy: 0.9480 - val_loss: 0.2808
Epoch 8/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 48s 20ms/step - accuracy: 0.9811 - loss: 0.0483 - val_accuracy: 0.9095 - val_loss: 0.3329
Epoch 9/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 46s 20ms/step - accuracy: 0.9838 - loss: 0.0437 - val_accuracy: 0.9486 - val_loss: 0.2279
Epoch 10/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 48s 21ms/step - accuracy: 0.9856 - loss: 0.0405 - val_accuracy: 0.9469 - val_loss: 0.1854
```

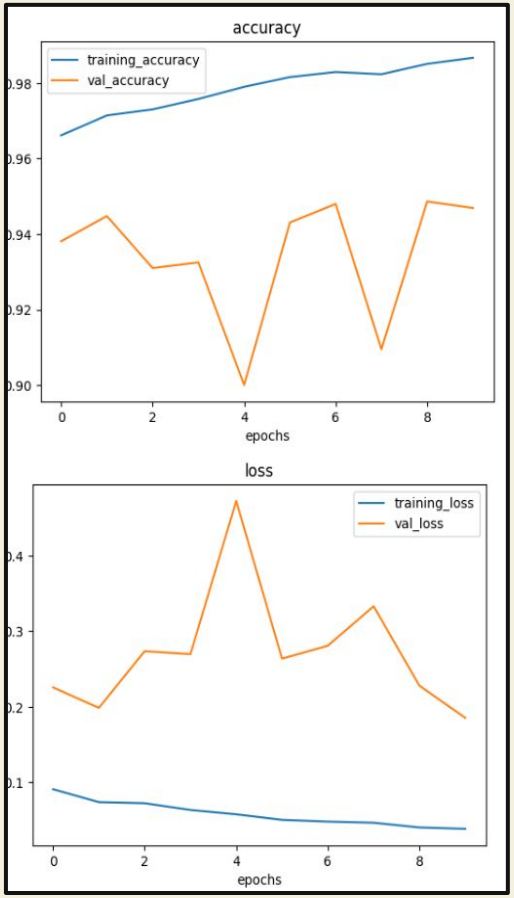
This function displays the validation accuracies for the model trained for 1, 5, and 10 epochs.

- The number of epochs is how many times the model looks at the dataset.
- Common issues with model accuracy is overfitting or underfitting the training data. This can be a result of the model not looking at the data enough times or too many times.
- Finding the ideal number of epochs to train our model for will prevent overfitting and underfitting while keeping the training time low.

Analyzing Accuracy and Loss Curves

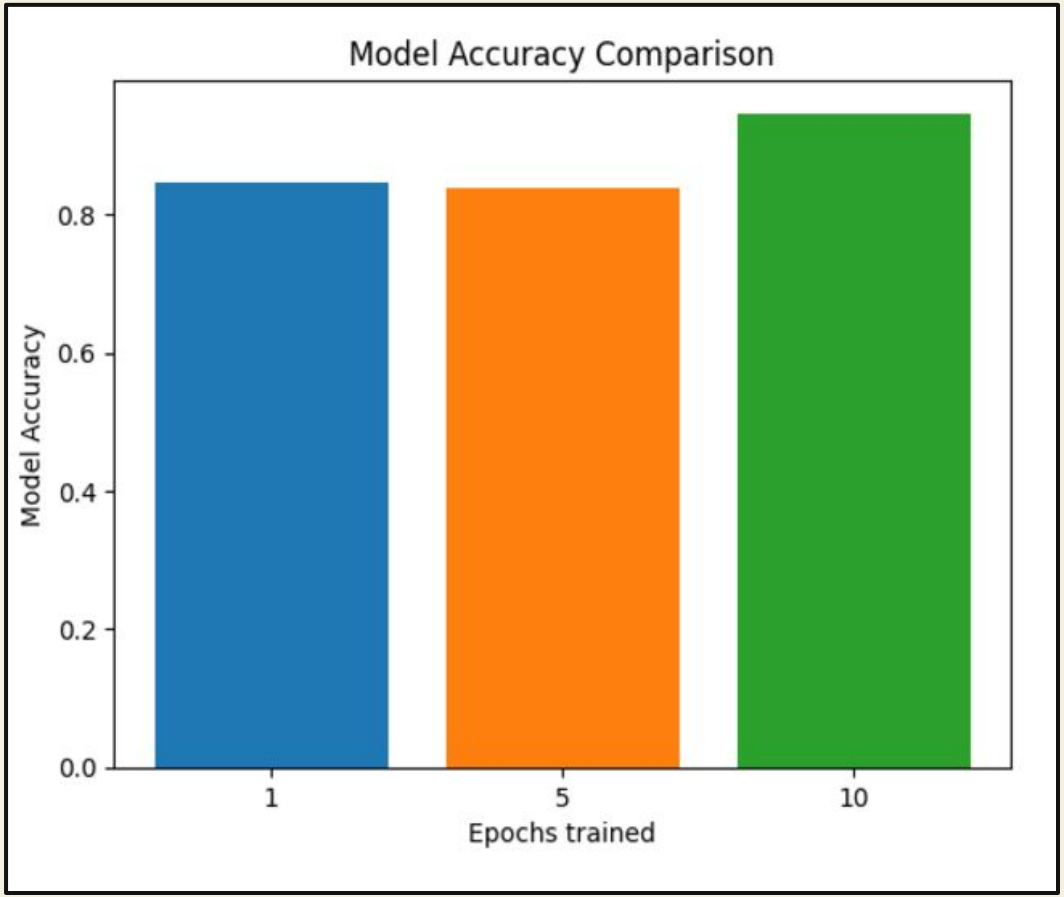
```python
# Check the loss curves
import matplotlib.pyplot as plt
def plot_loss_curves(history):
  loss = history.history["loss"]
  val_loss = history.history["val_loss"]

  accuracy = history.history["accuracy"]
  val_accuracy = history.history["val_accuracy"]

  epochs = range(len(history.history["loss"])) # How many epochs did we run for

  # Plot accuracy
  plt.plot(epochs, accuracy, label="training_accuracy")
  plt.plot(epochs, val_accuracy, label="val_accuracy")
  plt.title("accuracy")
  plt.xlabel("epochs")
  plt.legend()

  # Plot loss
  plt.figure()
  plt.plot(epochs, loss, label="training_loss")
  plt.plot(epochs, val_loss, label="val_loss")
  plt.title("loss")
  plt.xlabel("epochs")
  plt.legend()
```





Comparing Final Epoch Accuracy

```python
# Plot final accuracy
val_accuracy1 = 0.8470
val_accuracy2 = 0.8390
val_accuracy3 = 0.9469


def compare_results():
  val_accuracy1 = 0.8470
  val_accuracy2 = 0.8390
  val_accuracy3 = 0.9469
  accuracy1 = val_accuracy1
  accuracy2 = val_accuracy2
  accuracy3 = val_accuracy3

  # Plot accuracy comparison
  plt.bar("1", val_accuracy1)
  plt.bar("5", val_accuracy2)
  plt.bar("10", val_accuracy3)
  plt.title("Model Accuracy Comparison")
  plt.xlabel("Epochs trained")
  plt.ylabel("Model Accuracy")
```



Ideal Epoch for Accuracy

```python
def find_max_accuracy(my_history):
  current_max = 0.0
  epoch = None
  for index, value in enumerate(my_history.history["val_accuracy"]):
    if value > current_max:
      current_max = value
      epoch = index
  return f"Maximum Accuracy: {current_max}\nEpoch: {epoch}"

print(find_max_accuracy(history3))
```

```
Maximum Accuracy: 0.9486
Epoch: 9
```

```python
# Train even longer
history3 = model3.fit(train_data,
                      validation_data = test_data,
                        epochs = 10) # Increase to 10
```

```
Epoch 1/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 156s 31ms/step - accuracy: 0.9664 - loss: 0.0898 - val_accuracy: 0.9381 - val_loss: 0.2256
Epoch 2/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 46s 20ms/step - accuracy: 0.9741 - loss: 0.0666 - val_accuracy: 0.9448 - val_loss: 0.1986
Epoch 3/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 47s 20ms/step - accuracy: 0.9729 - loss: 0.0709 - val_accuracy: 0.9310 - val_loss: 0.2735
Epoch 4/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 46s 20ms/step - accuracy: 0.9753 - loss: 0.0633 - val_accuracy: 0.9325 - val_loss: 0.2697
Epoch 5/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 46s 20ms/step - accuracy: 0.9810 - loss: 0.0547 - val_accuracy: 0.9001 - val_loss: 0.4724
Epoch 6/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 49s 21ms/step - accuracy: 0.9820 - loss: 0.0491 - val_accuracy: 0.9430 - val_loss: 0.2637
Epoch 7/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 49s 21ms/step - accuracy: 0.9836 - loss: 0.0486 - val_accuracy: 0.9480 - val_loss: 0.2808
Epoch 8/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 48s 20ms/step - accuracy: 0.9811 - loss: 0.0483 - val_accuracy: 0.9095 - val_loss: 0.3329
Epoch 9/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 46s 20ms/step - accuracy: 0.9838 - loss: 0.0437 - val_accuracy: 0.9486 - val_loss: 0.2279
Epoch 10/10
2327/2327 ━━━━━━━━━━━━━━━━━━━━ 48s 21ms/step - accuracy: 0.9856 - loss: 0.0405 - val_accuracy: 0.9469 - val_loss: 0.1854
```

Contributions

Parker:

- Experimented with different epoch training times (1, 5, 10) and their impact on model accuracy.
- Created data visualization functions such as view random image or plot loss curves.
- Worked on data set download and saving to google drive

Colin:

- Looking, analyzing, and developing the original model and ensure it runs properly
- Preprocessing data to improve the quality and analysis of it

Josh:

- Experimented with different batch sizes (8, 16, and 32) to analyze their impact on model performance.
- Trained the model for 5 epochs for each batch size and tracked the validation accuracies to identify trends and trade-offs.

Team:

- Making the slides together
- Forming the proposal
- Writing the final project paper
- Providing assistance to other team members when needed

Questions?