VERDICKT_JULIA-PS6

April 12, 2024

1 Problem Set 6: Neural Networks

Warning! Some of the problems in this problem set require heavy computation - you are encouraged to start early so that you don't get stuck at the last minute.

2 Section 1: Neural Network Architecture

In the first section of this problem set, we'll spend some time examining neural network model architecture. Please type in your answers after each question – this section requires no coding.

2.0.1 Question 1.1

Consider an input image of dimensions 150 X 150 X 3: i.e., height and width of 150 pixels, with 3 channels.

- 1.1.1 Suppose you feed this image into a fully connected (dense) layer with 512 neurons. How many learnable / trainable parameters (or weights) does this layer have?
- 1.1.2 Now, suppose we feed this image into a Conv2D layer with 512 filters, kernel size 3 x 3, and stride 1 (assume 0 padding). How many learnable / trainable parameters (or weights) does this layer have?

Your Answer Here 1.1.1: A fully connected (dense) layer connects each input to each output in a layer, meaning every input neuron is connected to every output neuron. The total number of learnable parameters in a fully connected layer is $(150 \times 150 \times 3 \times 512) + 512 = 34,560,512$

1.1.2: In a convolutional layer, the parameters are determined by the number of filters, the size of these filters, and the depth of the input image. We have for each filter: $(3 \times 3 \times 3) + 1 = 28$ parameters where the 3's represent the kernel dimesions and the number of channels. The total parameters is thus $(28 \times 512) = 14,336$ as we multiply the number of parameters for each filter by the number of filters.

2.0.2 Question 1.2

Consider the following CNN, and answer the related questions. Assume that your input images are of dimensions (150, 150, 3): i.e., height and width of 150 pixels, with 3 channels.

1.2.1: Complete the table – fill in the missing entries (A, B, C, D). For each missing entry, provide a brief explanation for your answer (No more than 2 brief sentences.)

- Hint: The Keras documentation MaxPooling2D will be of use

	Layer	Output Shape	Number of parameters
1	Conv2D (No. of Filters: 32, Kernel Size: 3 X 3, Stride:1, Padding: 0)	(148, 148, 32)	896
2	Conv2D (No. of Filters: 64, Kernel Size: 3 X 3, Stride:1, Padding: 0)	A	B
3	MaxPooling2D(Pool size: 2 X 2, Padding: "valid", stride = "None")	C	0
4	Conv2D (No. of Filters: 64, Kernel Size: 3 X 3, Stride:1, Padding: 0)	(71, 71, 64)	D
5	MaxPooling2D(Pool size: 2 X 2, Padding: "valid", stride = "None")	(35, 35, 64)	0

1.2.2: Report the total number of parameters in this model?

1.2.3: Consider Layer 2 from above. Suppose we change the stride to 2, how does it affect the output shape (A) that you calculated above? How does it affect the number of parameters (B)?

Your Answer Here **1.2.1**: - **A**: The output height is $\frac{(148-3+2\times0)}{1}+1=146$. The output width is $\frac{(148-3+2\times0)}{1}+1=146$. Where 148 is the input width and height from the previous layer, 3 is the kernel dimeions, 0 is the padding, and the stride is the 1 in the denominator. The number of filters is 64. Thus the solution for **A** is $(146, 146, 64)$. - **B**: The total number of parameters is $((3\times3\times32+1)\times64)=18,496$. This comes from (kernel height * kernel width * number of channels + 1 for bias) * number of filters. - **C**: The new height is the $\lfloor\frac{146}{2}\rfloor=73$ and the new width is $\lfloor\frac{146}{2}\rfloor=73$. This comes from **height or width = floor (height or width)/(pool size)**. Max pooling does not affect the number of channels, so our new output shape is $(73, 73, 64)$. - **D**: The total number of parameters is $((3\times3\times64+1)\times64)=36,928$

1.2.2:

- The total number of parameters in the model is $896+18496+0+36928+0=56,320$

1.2.3: - If we change the stride to 2 for Layer 2, it will affect the output shape (A) but not the number of parameters (B). - The new output height is $\frac{(148-3+2\times0)}{2}+1=\frac{145}{2}+1$. The output width is $\frac{(148-3+2\times0)}{2}+1=\frac{145}{2}+1$. Since $\frac{145}{2}$ isn't an integer, we have to use its floor since we can't have half of a pixel. Thus, the new dimensions shrink to $\lfloor\frac{145}{2}\rfloor+1=72+1=73$ for both height and width. This change does not affect the number of parameters **B**, which is based on the kernel dimensions and number of channels, not the input height and width.

3 Section 2: Truck v/s Cars: Neural Networks and Image Classification

Your goal for this problem set is to train neural network models for image classification. Specifically, your task is to train models that correctly predict where the vehicle in a given image is a truck, or a car / automobile.

It might be useful to start by implementing this entire problem set on a relatively small subset of all of the images first, before using the full dataset.

3.1 2.1. Load Data + Exploratory Analysis

For this problem, we'll load the CIFAR 10 dataset, from the Keras API. This dataset has been widely used in ML and computer vision research – you can read more about the state of the art model performance (and how this has improved over time) here.

The CIFAR 10 dataset originally has 10 classes – we've provided helper code below to load the data, and remove images belonging to unnecessary classes. We will use this dataset for a supervised binary classification problem.

Your tasks: - Extract a validation set from your training data. Keep 70% of the images for training, while the remainder will be used for validation. - Examine a single image in from your training set. Report the dimensions (width, height, number of channels.) Plot each channel. - Select 9 random images from your training set. Plot these images in a 3 X 3 grid, along with the corresponding category / label - Plot the distribution of labels in your training, validation and test sets.

[1]:
```
from keras.datasets import cifar10
import numpy as np
def cifar_2classes():
    """
    Helper code to clean the CIFAR 10 dataset, and remove the unnecessary␣
 ↪classes.
    """
    ## Load data
    label_names = ["airplane",
                   "automobile",
                   "bird",
                   "cat",
                   "deer",
                   "dog",
                   "frog",
                   "horse",
                   "ship",
                   "truck"]


    label_map = {0:99, 1:0, 2:99, 3:99, 4:99, 5:99, 6:99, 7:99, 8:99, 9:1}

    (X_train_val, y_train_val), (X_test, y_test) = cifar10.load_data()

    (X_train_val, y_train_val), (X_test, y_test) = cifar10.load_data()
    y_train_val1 = np.array([[label_map[y[0]]] for y in y_train_val])
    y_test1 = np.array([[label_map[y[0]]] for y in y_test])

    X_train_val_clean = X_train_val[np.where(y_train_val1 != 99)[0]]
    y_train_val_clean =  y_train_val1[np.where(y_train_val1 != 99)]

    X_test_clean = X_test[np.where(y_test1 != 99 )[0]]
    y_test_clean = y_test1[np.where(y_test1 != 99)]
```

```
    return X_train_val_clean, y_train_val_clean, X_test_clean, y_test_clean
```

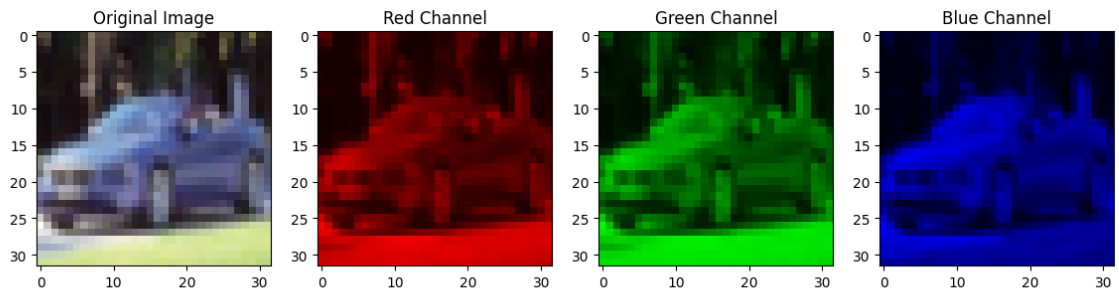
```
2024-04-12 16:12:10.508481: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild
TensorFlow with the appropriate compiler flags.
```

[2]:
```
from sklearn.model_selection import train_test_split

## Load data
X_train_val, y_train_val, X_test, y_test = cifar_2classes()

## Split into train, validation and test.
X_train, X_val, y_train, y_val = train_test_split(X_train_val,
                                                  y_train_val,
                                                  test_size=0.3,
                                                  random_state=0)
```

[3]:
```
### YOUR CODE HERE

import matplotlib.pyplot as plt



# Examine a single image from the training set.
# Report the dimensions (width, height, number of channels.)
# Plot each channel.
single_image = X_train[0]
print("Image dimensions:", single_image.shape)

# Plot the original image
plt.figure(figsize=(14, 6))
plt.subplot(1, 4, 1)
plt.imshow(single_image)
plt.title('Original Image')

# Plot each channel separately
for i, color in enumerate(['Red Channel',
                           'Green Channel',
                           'Blue Channel'], start=1):
    temp_image = np.zeros(single_image.shape, dtype='uint8')
     # Zero out the other two channels
    temp_image[:, :, i-1] = single_image[:, :, i-1]
    plt.subplot(1, 4, i+1)
    plt.imshow(temp_image)
    plt.title(color)
```

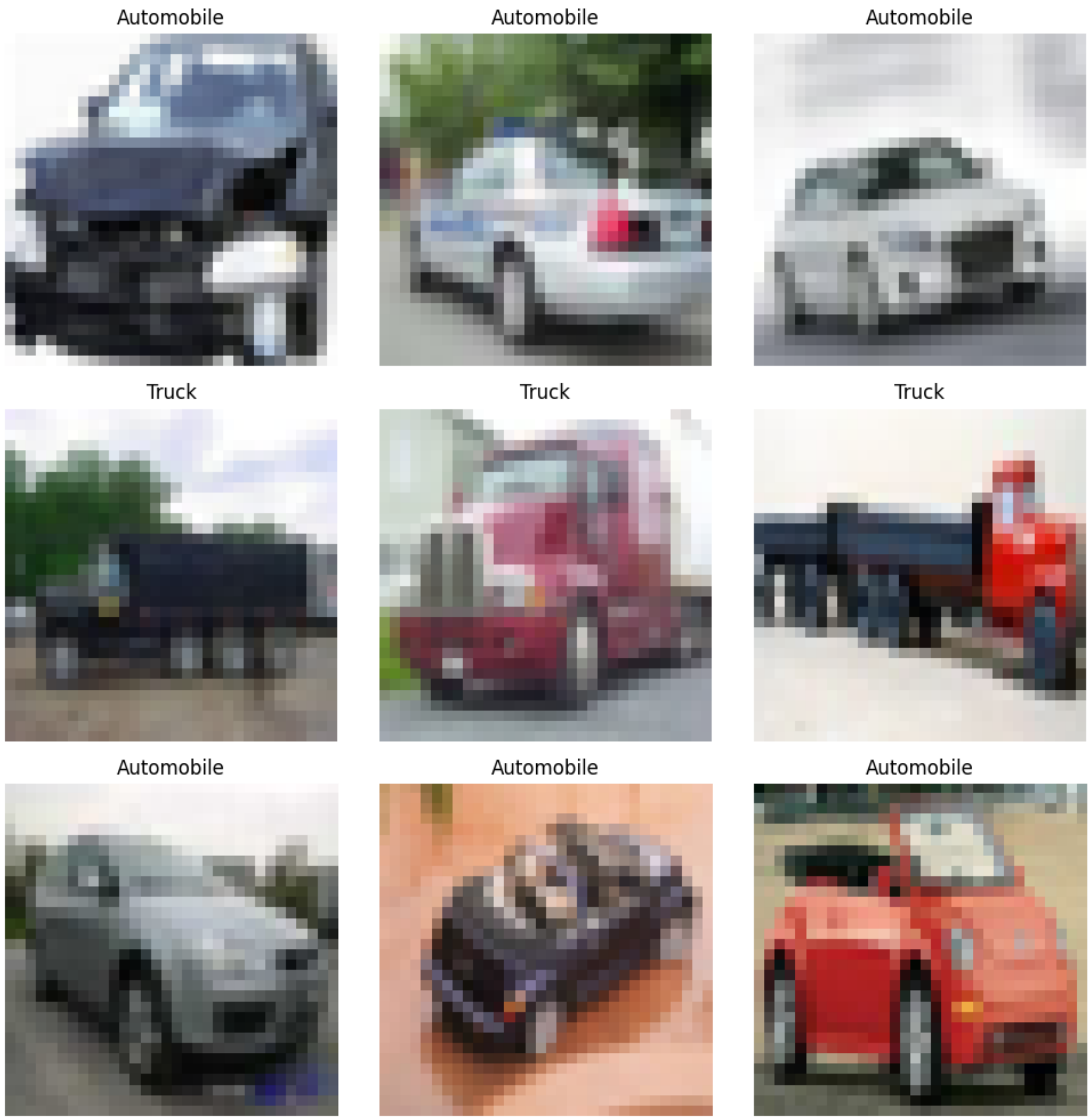
```
plt.show()
```

```
Image dimensions: (32, 32, 3)
```



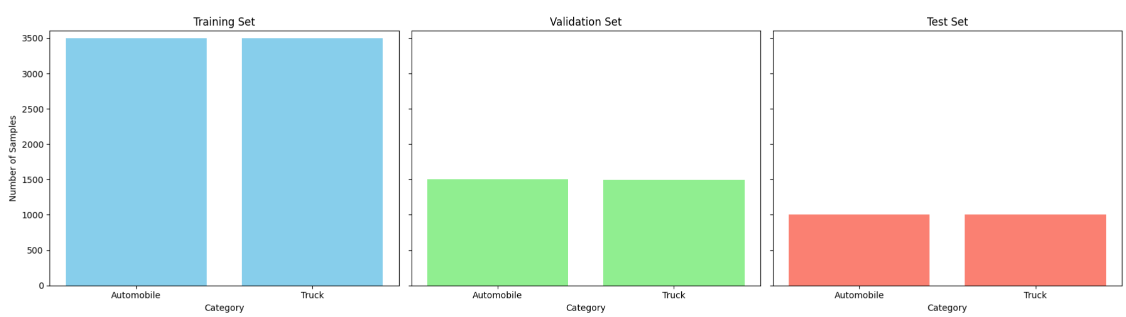
[4]:

```python
#Select 9 random images from your training set.
# Plot these images in a 3 X 3 grid, along with the corresponding category / label

np.random.seed(0)

num_images = 9
random_indices = np.random.choice(X_train.shape[0], num_images, replace=False)

# Label names for CIFAR-2 (binary classification from CIFAR-10)
label_names = {0: 'Automobile', 1: 'Truck'}

# Create a 3x3 grid for plotting
fig, axes = plt.subplots(3, 3, figsize=(10, 10))
axes = axes.flatten()

for i, idx in enumerate(random_indices):
    image = X_train[idx]
    label = y_train[idx]  # Adjust according to your label structure

    axes[i].imshow(image)
    axes[i].set_title(label_names[label])
    axes[i].axis('off')  # Hide the axes ticks

plt.tight_layout()
plt.show()
```



[5]:
```
# Function to count occurrences of each label
def count_labels(y):
    labels, counts = np.unique(y, return_counts=True)
    return labels, counts

# Prepare the data for plotting
labels_train, counts_train = count_labels(y_train)
labels_val, counts_val = count_labels(y_val)
labels_test, counts_test = count_labels(y_test)

# Determine the maximum count for consistent y-axis scale across plots
```

```
max_count = max(np.max(counts_train), np.max(counts_val), np.max(counts_test))
 ↪+100

# Create a 1-row, 3-column set of subplots
fig, axs = plt.subplots(1, 3, figsize=(18, 5), sharey=True)

# Plot for training set
axs[0].bar(labels_train, counts_train, color='skyblue')
axs[0].set_title('Training Set')
axs[0].set_xlabel('Category')
axs[0].set_ylabel('Number of Samples')
axs[0].set_xticks(labels_train)
axs[0].set_xticklabels(["Automobile", "Truck"])
axs[0].set_ylim(0, max_count)

# Plot for validation set
axs[1].bar(labels_val, counts_val, color='lightgreen')
axs[1].set_title('Validation Set')
axs[1].set_xlabel('Category')
axs[1].set_xticks(labels_val)
axs[1].set_xticklabels(["Automobile", "Truck"])
axs[1].set_ylim(0, max_count)

# Plot for test set
axs[2].bar(labels_test, counts_test, color='salmon')
axs[2].set_title('Test Set')
axs[2].set_xlabel('Category')
axs[2].set_xticks(labels_test)
axs[2].set_xticklabels(["Automobile", "Truck"])
axs[2].set_ylim(0, max_count)

plt.tight_layout()
plt.show()
```



3.2 2.2 Preprocessing

- Rescale the images data, so that the values lie between a range of 0 and 1.
- Hint: A simple way to do this is to divide by 255.0

[6]:
```
### YOUR CODE HERE
X_train_rescaled = X_train.astype('float32') / 255.0
X_val_rescaled = X_val.astype('float32') / 255.0
X_test_rescaled = X_test.astype('float32') / 255.0
```

3.3 2.3 Feedforward Neural Network

Reshape your data so that each image is flattened into a 1d array, and each of the train, test and validation sets are 2d arrays.

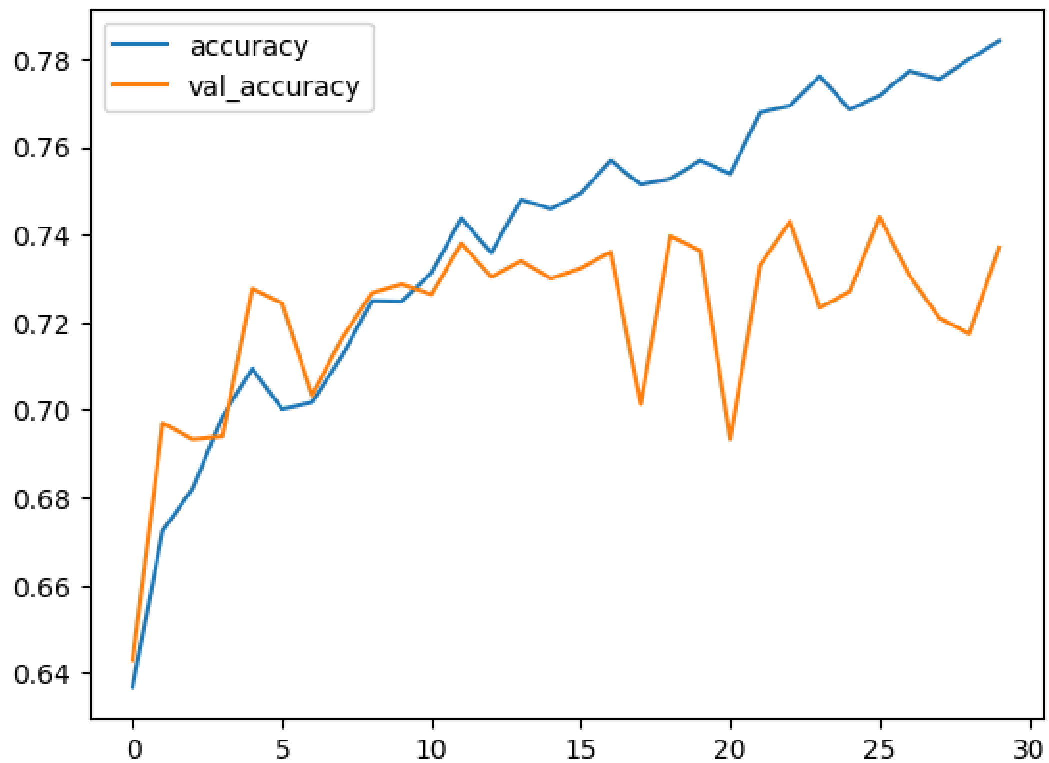
Essentially, your data should be an array of length N, where N is the number of observations (images) in the train / test / validation sets. Each element in the array is a flattened image, of length 3072 (32 X 32 X3)

[7]:
```
#### YOUR CODE HERE
X_train_flattened = X_train_rescaled.reshape(X_train_rescaled.shape[0], -1)
X_val_flattened = X_val_rescaled.reshape(X_val_rescaled.shape[0], -1)
X_test_flattened = X_test_rescaled.reshape(X_test_rescaled.shape[0], -1)
```

3.3.1 2.3.1 Build a neural network with the following parameters

- Architecture
  - Input dimensions: 3072
  - 1 hidden layer: 64 nodes, Relu activation
  - Output layer: 1 node, Sigmoid activation
- Compile the network:
  - Optimizer: Adam
  - Epochs: 30
  - Batch size: 32
  - Metrics: Accuracy
  - Remember to include the validation data in the compilation step.
- Outputs:
  - Plot the training and validation accuracy by epoch (See the example plot below). Do you see any evidence of overfitting in your plot?
  - Report the accuracy, Precision and Recall on the test set

Example plot



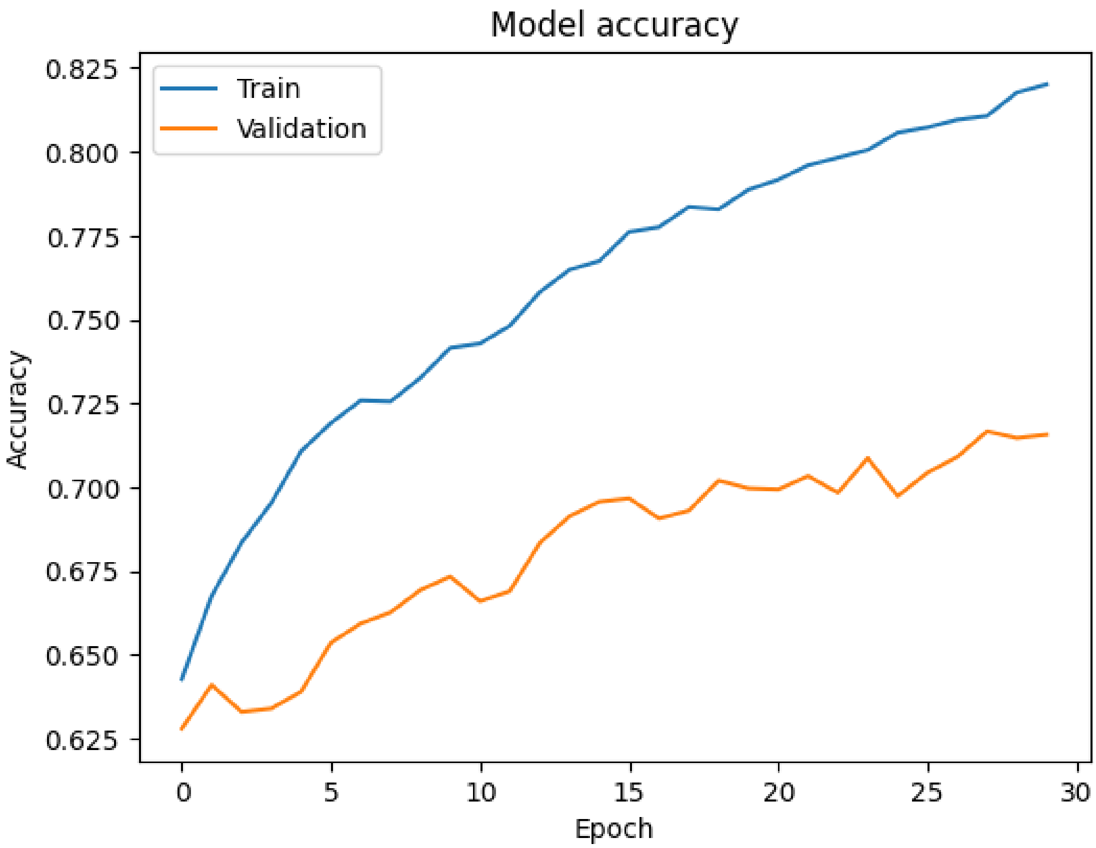
[8]:
```
### YOUR CODE HERE

from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
import tensorflow as tf
import keras.utils as ku
import random


np.random.seed(0)
tf.random.set_seed(0)
ku.set_random_seed(0)
random.seed(0)


#Architecture:
# Input dimensions: 3072
# 1 hidden layer: 64 nodes, Relu activation
# Output layer: 1 node, Sigmoid activation

# Build the neural network
model = Sequential()
model.add(Dense(64, input_dim=3072, activation='relu'))
```

```
model.add(Dense(1, activation='sigmoid'))

# Compile the network:
# Optimizer: Adam
# Epochs: 30
# Batch size: 32
# Metrics: Accuracy
model.compile(optimizer=Adam(),
              loss='binary_crossentropy',
              metrics=['accuracy'])
# Train the network
history = model.fit(X_train_flattened, y_train,
                    epochs=30,
                    batch_size=32,
                    validation_data=(X_val_flattened, y_val), verbose = 0)
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/keras/src/layers/core/dense.py:88: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[9]:

```
# Plot training & validation accuracy values
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



There is evidence of overfitting on this plot. As is apparent, the training data has consistently higher accuracy than the validation data for much of the plot.

[10]:
```
from sklearn.metrics import accuracy_score, precision_score
from sklearn.metrics import recall_score, classification_report

np.random.seed(0)
tf.random.set_seed(0)
ku.set_random_seed(0)
random.seed(0)

# Get predictions from the model
y_pred = model.predict(X_test_flattened)
y_pred = (y_pred > 0.5).astype('int32')

# Calculate accuracy, precision, and recall
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
```

```
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
```

```
63/63            0s 3ms/step
Accuracy: 0.701
Precision: 0.6497764530551415
Recall: 0.872
```

[11]:
```
# Evaluate the network on the test set
#test_loss, test_accuracy = model.evaluate(X_test_flattened, y_test)

# Classification report for Precision, Recall
print(classification_report(y_test, y_pred, target_names=['Automobile',␣
↪'Truck']))
```

```
              precision    recall  f1-score   support

  Automobile       0.81      0.53      0.64      1000
       Truck       0.65      0.87      0.74      1000

    accuracy                           0.70      2000
   macro avg       0.73      0.70      0.69      2000
weighted avg       0.73      0.70      0.69      2000
```

3.3.2 2.3.2. Tuning / Improving Performance

Now, go ahead and tune this network, or write up your own from scratch. The goal should be to exceed 75% overall classification accuracy on the test set. We don't expect you to implement cross-validation or any formal hyperparameter optimization techniques. Rather, the goal is to arrive at a model architecture that's acceptable to you via trial and error.

Remember that you have a number of hyperparameters to work with, including - the number / dimension of hidden layers - choice of activation functions, - type regularization, - optimization techniques

Note that you shouldn't need to train your model for more than 30 epochs.

The notebooks from Labs 9 and 10 are also a good starting point.

Outputs: - In 2-3 sentences, briefly explain the various choices/ decisions you made in building your model architecture. - Report the classification accuracy on the test set, along with the precision and recall for each class. - What do you notice about the precision and recall values, as well as the overall classification accuracy, in comparison to your outputs from 2.3.1?

[12]:
```
### YOUR CODE HERE
from keras.regularizers import l2
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```python
from tensorflow.keras.optimizers import Adam

np.random.seed(0)
tf.random.set_seed(0)
ku.set_random_seed(0)
random.seed(0)


# Model with increased complexity, lower learning rate, and regularization
model = Sequential()
model.add(Dense(128, input_dim=3072, activation='relu'))
model.add(Dense(64, activation='sigmoid', kernel_regularizer=l2(0.001)))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='binary_crossentropy',
              metrics=['accuracy'])


# Train the network
history = model.fit(X_train_flattened, y_train,
                    epochs=30,
                    batch_size=32,
                    validation_data=(X_val_flattened, y_val), verbose = 0)

# Get predictions from the model
y_pred = model.predict(X_test_flattened)
y_pred = (y_pred > 0.5).astype('int32')

# Calculate accuracy, precision, and recall
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/keras/src/layers/core/dense.py:88: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
63/63           0s 3ms/step
Accuracy: 0.754
Precision: 0.7382739212007504
Recall: 0.787
```

[13]:
```python
print(classification_report(y_test, y_pred, target_names=['Automobile',
    'Truck']))
```

```
              precision    recall  f1-score   support

  Automobile       0.77      0.72      0.75      1000
       Truck       0.74      0.79      0.76      1000

    accuracy                           0.75      2000
   macro avg       0.76      0.75      0.75      2000
weighted avg       0.76      0.75      0.75      2000
```

- In 2-3 sentences, briefly explain the various choices/ decisions you made in building your model architecture.
  - I increaseed the number of neurons and added an additional hidden layer to capture more complex patterns in the data. I also utilized the l2 regularization to try and account for the over fitting issue. Lastly, I decrease the learning rate to avoid overshooting.
- Report the classification accuracy on the test set, along with the precision and recall for each class.
  - The classification accuracy on the test set is 0.754. In terms of precision and recall, the precision for automobile is 0.77 and recall is 0.72. For Truck the precision is 0.74 and recall is 0.79.
- What do you notice about the precision and recall values, as well as the overall classification accuracy, in comparison to your outputs from 2.3.1?
  - Comparing to the outputs from 2.3.1, I observed that the overall accuracy and precision went up but the overall recall went down compared to 2.3.1. Also, at the class level, the precision for automobile when down but the recall went up by a lot. For truck, the precision went up by a lot while the recall went down.

3.4 2.4. Convolutional Neural Network

3.4.1 2.4.1. Build a CNN with the following parameters

- Architecture
  - Input dimensions: (32, 32, 3)
  - 1 Conv2D Layer:
    * Number of filters: 20.
    * Filter Dimension: 3 X 3.
    * Activation: Relu
  - Flatten
  - Output layer: 1 node, Sigmoid activation
- Compile the network:
  - Optimizer: Adam
  - Epochs: 20
  - Metrics: Accuracy
  - Remember to include the validation data in the compilation step.
- Outputs:
  - Plot the training and validation accuracy by epoch.

– Report the accuracy, Precision and Recall on the test set

[14]:
```python
### Your Code Here:
from keras.models import Sequential
from keras.layers import Conv2D, Flatten, Dense
from keras.optimizers import Adam

np.random.seed(0)
tf.random.set_seed(0)
ku.set_random_seed(0)
random.seed(0)

# Build the model
model_cnn = Sequential([
    Conv2D(20, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    Flatten(),
    Dense(1, activation='sigmoid')
])

# Compile the model
model_cnn.compile(optimizer=Adam(),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

# Train the model
history_cnn = model_cnn.fit(X_train_rescaled, y_train,
                            epochs=20,
                            batch_size=32,
                            validation_data=(X_val_rescaled, y_val),
                            verbose=1)
```

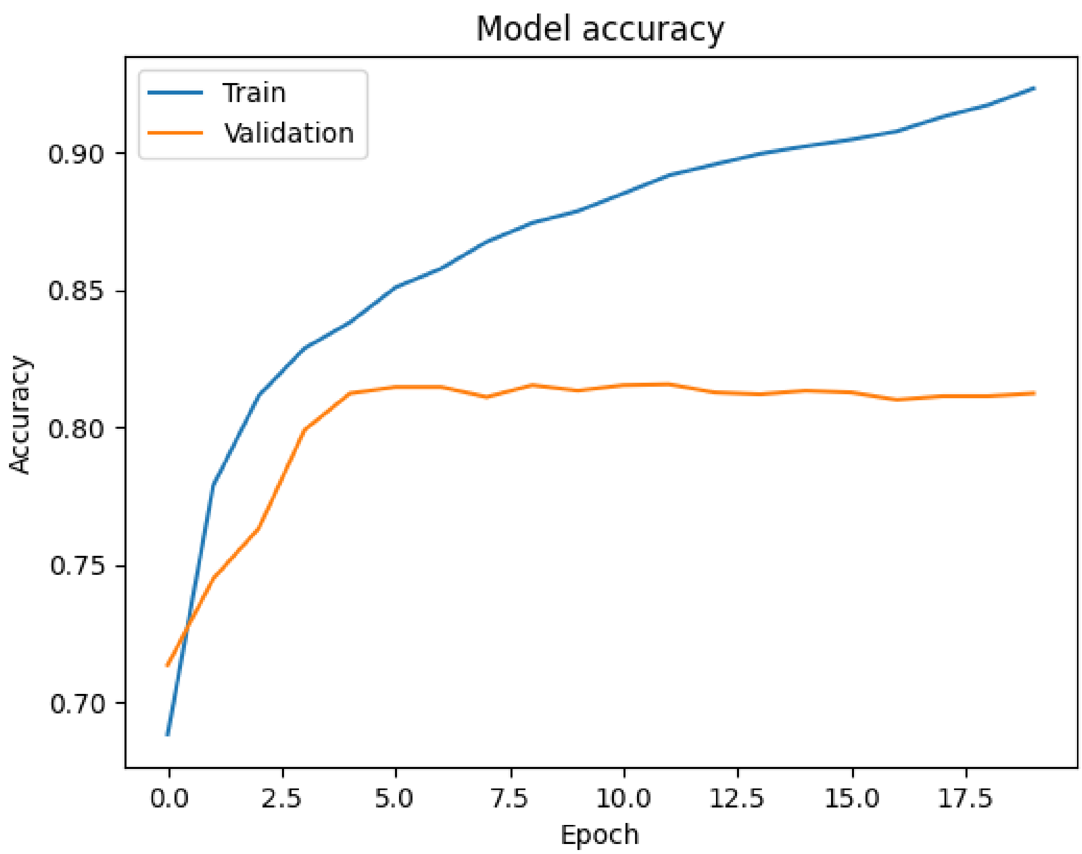
```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/keras/src/layers/convolutional/base_conv.py:99: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(

Epoch 1/20
219/219               3s 9ms/step -
accuracy: 0.6213 - loss: 0.6451 - val_accuracy: 0.7137 - val_loss: 0.5490
Epoch 2/20
219/219               2s 8ms/step -
accuracy: 0.7606 - loss: 0.5039 - val_accuracy: 0.7450 - val_loss: 0.5028
Epoch 3/20
219/219               2s 8ms/step -
accuracy: 0.8025 - loss: 0.4439 - val_accuracy: 0.7633 - val_loss: 0.4830
Epoch 4/20
219/219               2s 8ms/step -
```

```
accuracy: 0.8211 - loss: 0.4097 - val_accuracy: 0.7990 - val_loss: 0.4438
Epoch 5/20
219/219            2s 7ms/step -
accuracy: 0.8279 - loss: 0.3872 - val_accuracy: 0.8123 - val_loss: 0.4283
Epoch 6/20
219/219            2s 7ms/step -
accuracy: 0.8438 - loss: 0.3664 - val_accuracy: 0.8147 - val_loss: 0.4249
Epoch 7/20
219/219            2s 9ms/step -
accuracy: 0.8532 - loss: 0.3480 - val_accuracy: 0.8147 - val_loss: 0.4233
Epoch 8/20
219/219            2s 8ms/step -
accuracy: 0.8611 - loss: 0.3326 - val_accuracy: 0.8110 - val_loss: 0.4256
Epoch 9/20
219/219            2s 8ms/step -
accuracy: 0.8693 - loss: 0.3171 - val_accuracy: 0.8153 - val_loss: 0.4264
Epoch 10/20
219/219            2s 7ms/step -
accuracy: 0.8748 - loss: 0.3081 - val_accuracy: 0.8133 - val_loss: 0.4277
Epoch 11/20
219/219            2s 7ms/step -
accuracy: 0.8801 - loss: 0.2970 - val_accuracy: 0.8153 - val_loss: 0.4308
Epoch 12/20
219/219            2s 8ms/step -
accuracy: 0.8882 - loss: 0.2871 - val_accuracy: 0.8157 - val_loss: 0.4329
Epoch 13/20
219/219            2s 8ms/step -
accuracy: 0.8934 - loss: 0.2775 - val_accuracy: 0.8127 - val_loss: 0.4373
Epoch 14/20
219/219            2s 8ms/step -
accuracy: 0.8958 - loss: 0.2687 - val_accuracy: 0.8120 - val_loss: 0.4487
Epoch 15/20
219/219            2s 7ms/step -
accuracy: 0.8995 - loss: 0.2590 - val_accuracy: 0.8133 - val_loss: 0.4534
Epoch 16/20
219/219            2s 7ms/step -
accuracy: 0.9010 - loss: 0.2525 - val_accuracy: 0.8127 - val_loss: 0.4612
Epoch 17/20
219/219            2s 8ms/step -
accuracy: 0.9032 - loss: 0.2488 - val_accuracy: 0.8100 - val_loss: 0.4671
Epoch 18/20
219/219            2s 8ms/step -
accuracy: 0.9057 - loss: 0.2411 - val_accuracy: 0.8113 - val_loss: 0.4752
Epoch 19/20
219/219            2s 8ms/step -
accuracy: 0.9102 - loss: 0.2318 - val_accuracy: 0.8113 - val_loss: 0.4779
Epoch 20/20
219/219            2s 8ms/step -
```

```
accuracy: 0.9182 - loss: 0.2210 - val_accuracy: 0.8123 - val_loss: 0.4802
```

[15]:

```
### YOUR CODE HERE
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score
from sklearn.metrics import recall_score, classification_report

# Plot training & validation accuracy values
plt.plot(history_cnn.history['accuracy'])
plt.plot(history_cnn.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



[16]:

```
np.random.seed(0)
tf.random.set_seed(0)
ku.set_random_seed(0)
random.seed(0)
```

```python
# Predictions on test set
y_pred = model_cnn.predict(X_test_rescaled)
y_pred = (y_pred > 0.5).astype('int32')

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)

print(f"Test Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")

# Detailed classification report
print(classification_report(y_test, y_pred, target_names=['Automobile',␣
 ↪'Truck']))
```

```
63/63            0s 4ms/step
Test Accuracy: 0.7820
Precision: 0.8241
Recall: 0.7170
              precision    recall  f1-score   support

  Automobile       0.75      0.85      0.80      1000
       Truck       0.82      0.72      0.77      1000

    accuracy                           0.78      2000
   macro avg       0.79      0.78      0.78      2000
weighted avg       0.79      0.78      0.78      2000
```

3.4.2 2.4.2. Tuning / Improving Performance

Now, go ahead and tune this network, or write up your own from scratch. The goal should be to exceed 85% overall classification accuracy on the test set. We don't expect you to implement cross-validation or any formal hyperparameter optimization techniques. Rather, the goal is to arrive at a model architecture that's acceptable to you via trial and error.

Note that you shouldn't need to train your model for more than 30 epochs.

Remember that you have a number of hyperparameters to work with, including - the number / dimension of hidden layers - choice of activation functions, - type regularization, - optimization techniques - and other relevant aspects(adding data augmentation etc.)

The notebooks from Labs 9 and 10 are a good starting point in terms of putting together a more complex architecture.

Warning! If you intend to attempt **Extra Credit 1 and 2** (below), ensure that you carefully name / store the trained model you build in this step. It's fine to keep trained model in memory, or to save the weights to disk.

Outputs: - Report the classification accuracy on the test set, along with the precision and recall for each class. - Briefly explain your model architecture / choices you made in tuning your CNN (No more than 3 - 4 sentences) - What do you notice about the precision and recall values, as well as the overall classification accuracy, in comparison to the feed forward neural networks from part 2.3, and your baseline in 2.4.1?

[17]:
```
## YOUR CODE HERE
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import Input

model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])



model.compile(optimizer=Adam(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Data Augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
)
datagen.fit(X_train_rescaled)
```

```python
# Train
history_cnn = model.fit(datagen.flow(X_train_rescaled, y_train, batch_size=32),
                    epochs=30,
                    validation_data=(X_val_rescaled, y_val),
                    verbose=1)
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/keras/src/layers/convolutional/base_conv.py:99: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(
```

```
Epoch 1/30
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:120:
UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in
its constructor. `**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will be
ignored.
  self._warn_if_super_not_called()
```

```
219/219            16s 57ms/step -
accuracy: 0.6098 - loss: 0.8712 - val_accuracy: 0.6757 - val_loss: 0.6440
Epoch 2/30
219/219            13s 56ms/step -
accuracy: 0.6962 - loss: 0.5843 - val_accuracy: 0.7517 - val_loss: 0.5586
Epoch 3/30
219/219            13s 60ms/step -
accuracy: 0.7396 - loss: 0.5382 - val_accuracy: 0.7643 - val_loss: 0.4982
Epoch 4/30
219/219            13s 57ms/step -
accuracy: 0.7644 - loss: 0.5034 - val_accuracy: 0.5740 - val_loss: 0.9422
Epoch 5/30
219/219            13s 57ms/step -
accuracy: 0.7858 - loss: 0.4707 - val_accuracy: 0.8507 - val_loss: 0.3665
Epoch 6/30
219/219            12s 56ms/step -
accuracy: 0.7889 - loss: 0.4694 - val_accuracy: 0.7187 - val_loss: 0.5601
Epoch 7/30
219/219            12s 55ms/step -
accuracy: 0.8042 - loss: 0.4489 - val_accuracy: 0.7330 - val_loss: 0.5991
Epoch 8/30
219/219            13s 58ms/step -
accuracy: 0.8135 - loss: 0.4203 - val_accuracy: 0.8807 - val_loss: 0.2977
Epoch 9/30
219/219            13s 56ms/step -
accuracy: 0.8182 - loss: 0.4137 - val_accuracy: 0.8683 - val_loss: 0.3213
```

```
Epoch 10/30
219/219            13s 58ms/step -
accuracy: 0.8142 - loss: 0.4077 - val_accuracy: 0.7707 - val_loss: 0.5295
Epoch 11/30
219/219            12s 53ms/step -
accuracy: 0.8213 - loss: 0.4091 - val_accuracy: 0.7860 - val_loss: 0.4726
Epoch 12/30
219/219            13s 60ms/step -
accuracy: 0.8246 - loss: 0.3903 - val_accuracy: 0.7850 - val_loss: 0.4831
Epoch 13/30
219/219            12s 54ms/step -
accuracy: 0.8520 - loss: 0.3525 - val_accuracy: 0.7617 - val_loss: 0.5631
Epoch 14/30
219/219            12s 54ms/step -
accuracy: 0.8471 - loss: 0.3611 - val_accuracy: 0.8337 - val_loss: 0.3944
Epoch 15/30
219/219            12s 56ms/step -
accuracy: 0.8554 - loss: 0.3464 - val_accuracy: 0.7657 - val_loss: 0.5646
Epoch 16/30
219/219            12s 56ms/step -
accuracy: 0.8499 - loss: 0.3503 - val_accuracy: 0.8880 - val_loss: 0.2670
Epoch 17/30
219/219            12s 52ms/step -
accuracy: 0.8567 - loss: 0.3405 - val_accuracy: 0.9207 - val_loss: 0.2187
Epoch 18/30
219/219            12s 53ms/step -
accuracy: 0.8626 - loss: 0.3226 - val_accuracy: 0.8917 - val_loss: 0.2537
Epoch 19/30
219/219            15s 68ms/step -
accuracy: 0.8693 - loss: 0.3116 - val_accuracy: 0.9183 - val_loss: 0.2245
Epoch 20/30
219/219            13s 58ms/step -
accuracy: 0.8660 - loss: 0.3128 - val_accuracy: 0.8123 - val_loss: 0.4776
Epoch 21/30
219/219            13s 60ms/step -
accuracy: 0.8681 - loss: 0.3236 - val_accuracy: 0.8263 - val_loss: 0.4113
Epoch 22/30
219/219            14s 61ms/step -
accuracy: 0.8738 - loss: 0.3091 - val_accuracy: 0.9247 - val_loss: 0.2087
Epoch 23/30
219/219            13s 60ms/step -
accuracy: 0.8823 - loss: 0.2873 - val_accuracy: 0.9200 - val_loss: 0.2153
Epoch 24/30
219/219            14s 63ms/step -
accuracy: 0.8887 - loss: 0.2833 - val_accuracy: 0.8910 - val_loss: 0.2839
Epoch 25/30
219/219            19s 85ms/step -
accuracy: 0.8841 - loss: 0.2806 - val_accuracy: 0.9270 - val_loss: 0.2043
```

```
Epoch 26/30
219/219          19s 84ms/step -
accuracy: 0.8911 - loss: 0.2777 - val_accuracy: 0.8140 - val_loss: 0.4637
Epoch 27/30
219/219          14s 63ms/step -
accuracy: 0.8808 - loss: 0.2904 - val_accuracy: 0.9210 - val_loss: 0.2071
Epoch 28/30
219/219          14s 63ms/step -
accuracy: 0.8801 - loss: 0.2868 - val_accuracy: 0.9133 - val_loss: 0.2221
Epoch 29/30
219/219          13s 57ms/step -
accuracy: 0.8923 - loss: 0.2705 - val_accuracy: 0.9127 - val_loss: 0.2258
Epoch 30/30
219/219          13s 59ms/step -
accuracy: 0.8896 - loss: 0.2747 - val_accuracy: 0.8967 - val_loss: 0.2636
```

[18]:

```
np.random.seed(0)
tf.random.set_seed(0)
ku.set_random_seed(0)
random.seed(0)



# Predictions on test set
y_pred = model.predict(X_test_rescaled)
y_pred = (y_pred > 0.5).astype('int32')

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)

print(f"Test Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")

# Detailed classification report
print(classification_report(y_test, y_pred, target_names=['Automobile',␣
↪'Truck']))
```

```
63/63          1s 10ms/step
Test Accuracy: 0.8935
Precision: 0.9377
Recall: 0.8430
              precision    recall  f1-score   support

  Automobile       0.86      0.94      0.90      1000
       Truck       0.94      0.84      0.89      1000
```

```
    accuracy                           0.89      2000
   macro avg       0.90      0.89      0.89      2000
weighted avg       0.90      0.89      0.89      2000
```

[19]:
```
model.save('my_model.keras')
```

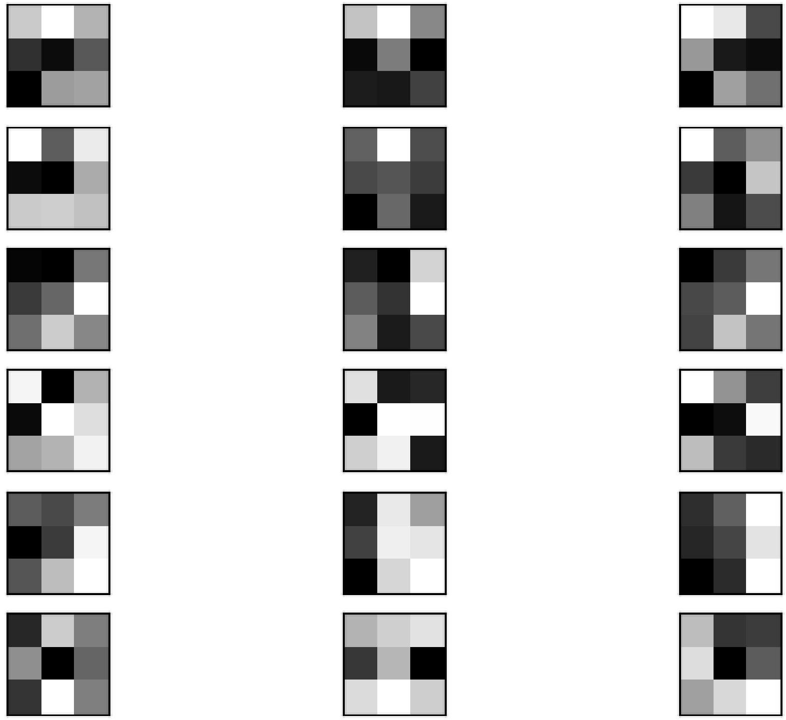
Outputs:

- Report the classification accuracy on the test set, along with the precision and recall for each class.
  - The accuracy of the test set is about 89.35%. The Automobile has a precision of 86% and a recall of 94%. The Truck class has a precision of 94% and a recall of 84%
- Briefly explain your model architecture / choices you made in tuning your CNN (No more than 3 - 4 sentences)
  - I introduced additional convolutional layers to capture more complex features and applied MaxPooling to reduce dimensionality and computational cost. Batch normalization was utilized to stabilize and speed up the training process, while dropout layers were used to address overfitting. I added data augmentation through ImageDataGenerator to enrich the dataset to enhance the model's out of sample fit by simulating variations in the training data.
- What do you notice about the precision and recall values, as well as the overall classification accuracy, in comparison to the feed forward neural networks from part 2.3, and your baseline in 2.4.1?
  - The overall accuracy, precision, and recall values are higher in 2.4.2 than in any model from 2.3. Interestingly this can also be said for 2.4.1 as well. Unlike the difference between 2.3.1 and 2.3.2, 2.4.2 simply has overall higher class precision and recall values as well as higher overall test model accuracy, precision, and recall. In 2.3.2, we saw that higher accuracy came when class precision and recall became more balanced. In 2.4.2, values simply increase across the board without this stark change in "closeness" between class precision and recall.

3.4.3 2.5: Convolutional Filters

Now, let's attempt to better understand what our CNN is doing under the hood. We'll start by visually examining our convolutional filters.

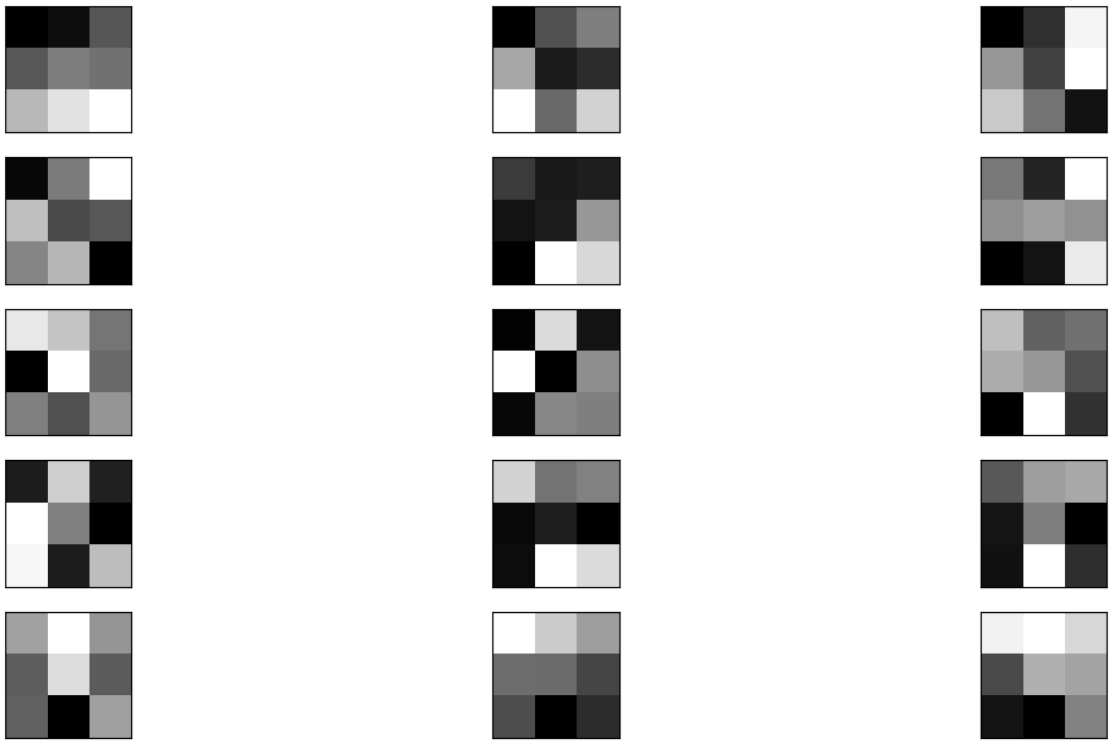
- We'll focus on the first convolutional layer in your CNN.
  - Use the get_weights() method to obtain the filters.
  - Plot the first 5 filters, for each channel (your plot will be a grid of 5 X 3).
  - Your plot will resemble the one below (the exact nature of the visual patterns will depend on your model architecture etc.)
  - What do you observe about the filters you visualize?



Example output

[20]:
```
## YOUR CODE HERE

import matplotlib.pyplot as plt

# Assuming 'model' is your trained model and the first layer is the Conv2D layer
filters, biases = model.layers[0].get_weights()

# Normalize filter values between 0 and 1 for visualization
f_min, f_max = filters.min(), filters.max()
filters = (filters - f_min) / (f_max - f_min)

# Plot first 5 filters
n_filters, ix = 5, 1
plt.figure(figsize=(15, 8))
for i in range(n_filters):
    # Get the filter
    f = filters[:, :, :, i]
    # Plot each channel separately
    for j in range(3):
        # Specify subplot and turn of axis
        ax = plt.subplot(n_filters, 3, ix)
        ax.set_xticks([])
        ax.set_yticks([])
        # Plot filter channel in grayscale
        plt.imshow(f[:, :, j], cmap='gray')
```

```
        ix += 1
# Show the figure
plt.show()
```



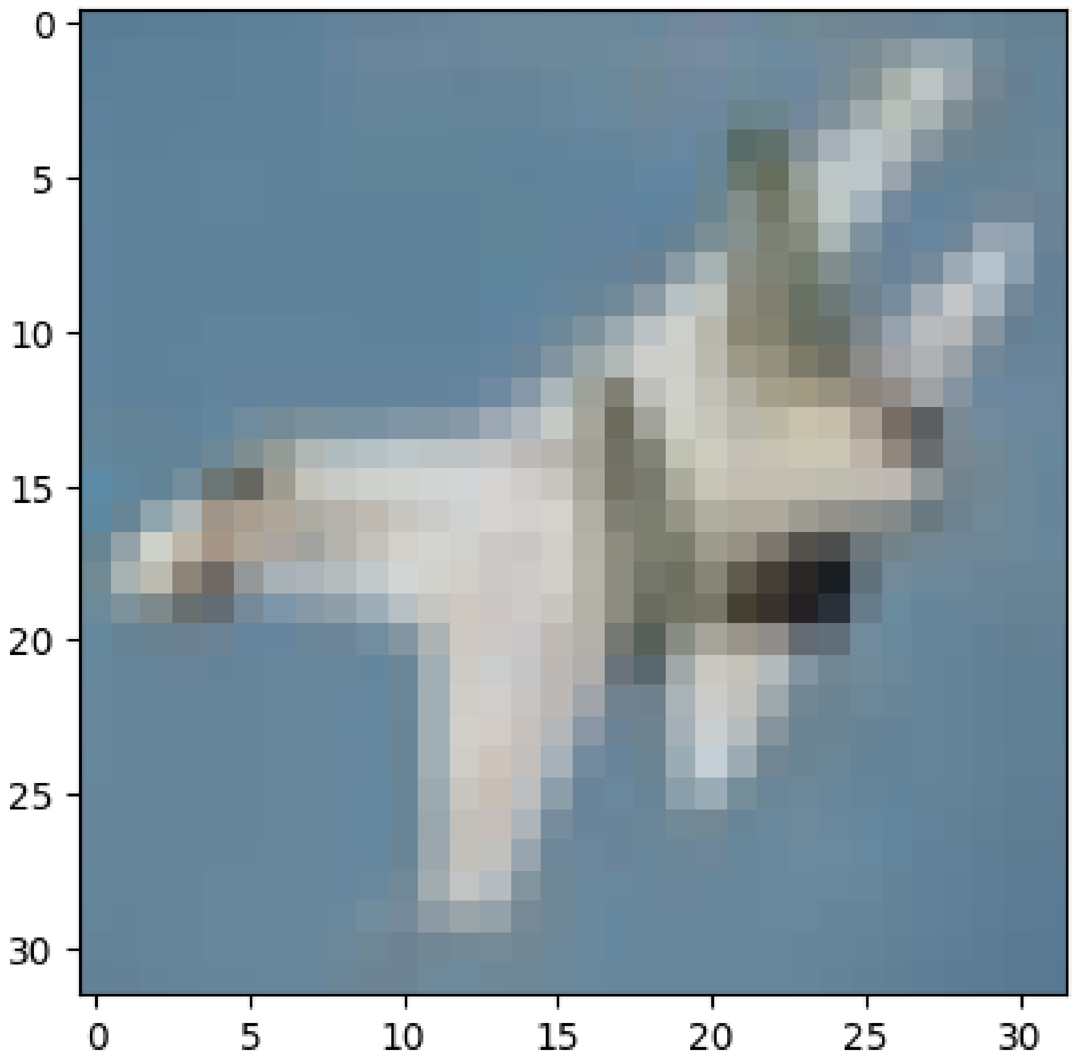
- What do you observe about the filters you visualize?
    - each filter has a unique pattern, which implies that each one is looking for something different in the input images. This diversity allows the network to capture a wide variety of features from the input space
    - Some filters show gradients from light to dark or vice versa. These types of filters can be sensitive to particular orientations of light gradients, which could be indicative of certain textures or shapes

3.4.4 Extra Credit 1: Feature Maps

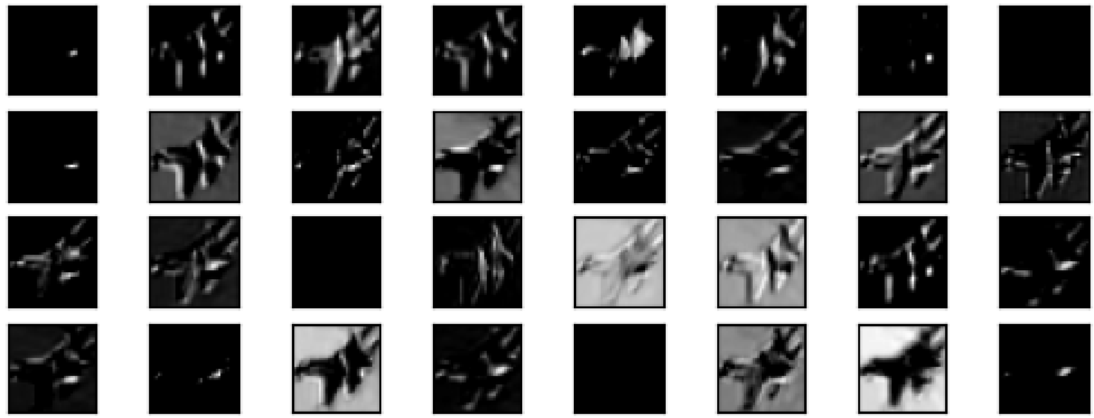
A feature map, or an activation map allows us to examine the result of applying the filters to a given input. The broad intuition is that feature maps closer to the input image detect fine-grained detail, whereas feature maps closer to the output of the model capture more generic aspects.

Your task is to create and visualize a feature map (i.e the outputs) from the first convolutional layer in your trained CNN.

In order to do this, proceed as follows: - Identify a nice image from your training data – ideally, something that has some distinguishing properties to the naked eye. - Pass this image through your trained CNN from **2.4.2**, and store the output from the first convolutional layer – this is your feature map! Note that there are multiple ways to do this; the simplest is to create a copy of your trained CNN, and remove the later layers. The Models function can help you do this. - Note that the size of the feature map depends on how many filters you have in the layer. - Outputs: - plot 1) The raw image from the training data, and 2) the feature map. An example is shown below: - what do you observe about the feature maps?



Raw Image



Feature Map

[21]:
```
### YOUR CODE HERE

# First, store the inputs / outputs from the first convolutional /
# hidden layer in your network.
# Hint: The keras documentation will be helpful
# (https://keras.io/guides/functional_api/)


# Note that you can create a model using another model/ layer's inputs /␣
↪outputs:
# model = keras.Model(inputs=inputs, outputs=outputs, name="mnist_model")

from tensorflow.keras.models import load_model, Model
from keras import layers


# Load the entire model
model = load_model('my_model.keras')
activation_model = Model(inputs=model.layers[0].input, outputs=model.layers[0].
↪output)

# Then, pass your chosen image through(i.e predict)
# Image preprocessing as before
img_index = 0  # Replace with your chosen index
chosen_image = X_train_rescaled[img_index]
img_tensor = np.expand_dims(chosen_image, axis=0)
# Get the feature map for the image
feature_maps = activation_model.predict(img_tensor)
```

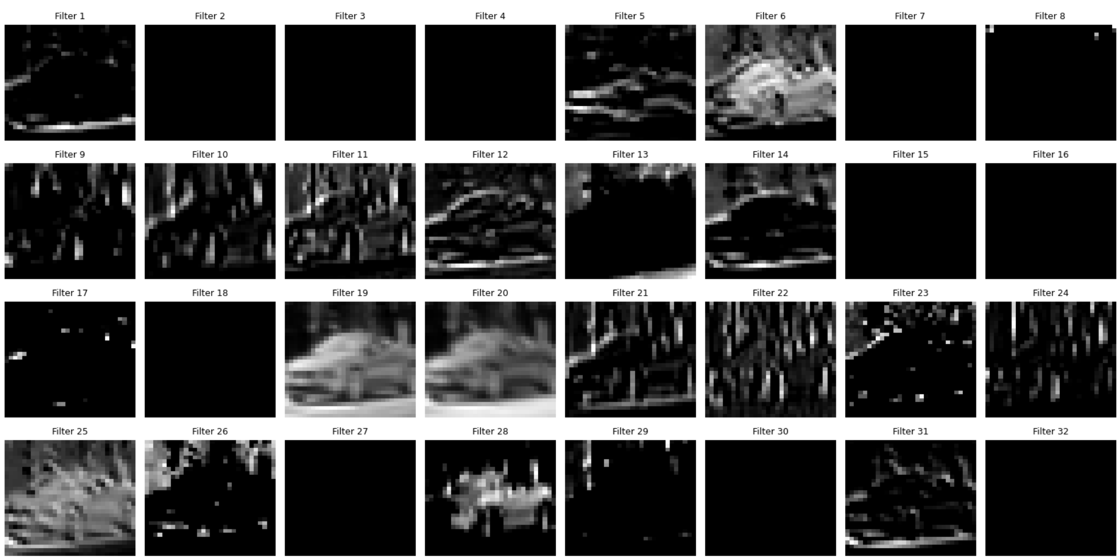
```
1/1           0s 52ms/step
```

- what do you observe about the feature maps?
  - Some of the feature maps are monotone. You can't see anything but a single color. Alternatively, some have a much clearly car image, while others show an image but it's very unclear

- What might this imply. Maybe there are some inactive or less active filters. Some filters may not activate strongly in response to the input image, resulting in feature maps that look almost blank or monotone.
- Some filters might also be redundant. If multiple filters are learning to detect similar features, their feature maps might look very similar or even monotone if those features aren't present in the specific image.

[22]:

```python
## Plot the original image, and the feature maps
# Plot the raw image
plt.figure(figsize=(4, 4))
plt.imshow(chosen_image)
plt.title("Raw Image")
plt.axis('off')
plt.show()

# Number of columns for our grid
n_cols = 8
# Number of rows, each filter gets its own row
n_rows = feature_maps.shape[-1] // n_cols

# Set up the matplotlib figure and axis grid
fig, axes = plt.subplots(n_rows, n_cols, figsize=(n_cols * 2, n_rows * 2))
axes = axes.ravel()  # Flatten the axes array for easy iteration

for i in range(feature_maps.shape[-1]):  # Iterate over the number of filters
    # Get the feature map for the ith filter
    feature_map = feature_maps[0, :, :, i]
    # Display the feature map in the ith subplot
    axes[i].imshow(feature_map, cmap='gray', aspect='auto')
    axes[i].axis('off')  # Turn off axis ticks and labels
    axes[i].set_title(f'Filter {i+1}', fontsize=9)

# Adjust layout so that titles don't overlap
fig.tight_layout(h_pad=1, w_pad=1)
plt.show()
```



3.4.5 Extra Credit 2: Transfer Learning / Fine tuning

Suppose you have a slightly different classification task at hand: to correctly separate trucks from airplanes.

We'll examine how we can use an already trained model to do this, instead of coding up a new neural network from scratch.

You are required to implement two approaches:

- First, we'll use the CNN from D2 above – and simply update the weights.
- Second, we'll load a pre-trained model from keras/ tensorflow (e.g. ResNet50, or VGG19). While these models haven't seen the exact images in this dataset, they have been trained on a large general corpus. Since these models have millions of weights, so we'll implement the following approach:
  - Load a pre-trained model
  - Freeze the weights by setting trainable = False.
  - Build a new model by adding additional layers to the base model.
  - Train the new model and evaluate performance.
- Compare the performance of both approaches, and briefly summarize your observations

We have provided some helper code and hints in the cells below.

Warning! Note that the second approach could be slow / time-consuming. If you are attempting it, please ensure that you budget ~20 mins to 1hour (worst case) for the code to complete running for this part.

This is a handy reference: https://keras.io/guides/transfer_learning/#transfer-learning-amp-finetuning

[23]:
```
def cifar_2moreclasses(pos_class, neg_class):
    """
    Helper code to clean the CIFAR 10 dataset, and remove the unnecessary␣
 ↪classes.
    """
    ## Load data
    label_names = ["airplane",
              "automobile",
              "bird",
              "cat",
              "deer",
              "dog",
              "frog",
              "horse",
              "ship",
              "truck"]


    label_map = {i:99 for i in range(len(label_names))}
    label_map[pos_class] = 1
    label_map[neg_class] = 0

    (X_train_val, y_train_val), (X_test, y_test) = cifar10.load_data()

    (X_train_val, y_train_val), (X_test, y_test) = cifar10.load_data()
    y_train_val1 = np.array([[label_map[y[0]]] for y in y_train_val])
    y_test1 = np.array([[label_map[y[0]]] for y in y_test])
```

```
    X_train_val_clean = X_train_val[np.where(y_train_val1 != 99)[0]]
    y_train_val_clean =  y_train_val1[np.where(y_train_val1 != 99)]

    X_test_clean = X_test[np.where(y_test1 != 99 )[0]]
    y_test_clean = y_test1[np.where(y_test1 != 99)]

    return X_train_val_clean, y_train_val_clean, X_test_clean, y_test_clean
```

[24]:

```
## Load data
X_train_val1, y_train_val1, X_test1, y_test1 = cifar_2moreclasses(9, 0)

## Split into train, validation and test.
N_train, N_validation, N_test = 7000, 3000, 2000

X_train1 = X_train_val1[:N_train,:,:]
y_train1 = y_train_val1[:N_train]

X_val1 = X_train_val1[N_train: N_train + N_validation,:,:]
y_val1 = y_train_val1[N_train: N_train + N_validation]

X_test1 = X_test1[:N_test]
y_test1 = y_test1[:N_test]

print(X_train1.shape, X_val1.shape, X_test1.shape)
```

```
(7000, 32, 32, 3) (3000, 32, 32, 3) (2000, 32, 32, 3)
```

[25]:

```
#### APPROACH 1
```

[26]:

```
from keras.models import clone_model

np.random.seed(0)
tf.random.set_seed(0)
ku.set_random_seed(0)
random.seed(0)

# Helper Code: Cloning the model (Edit the line below
# with the name of your model from D2)
model_from_D2 = load_model('my_model.keras')
model_cnn3 = clone_model( model_from_D2 )

# To do: Compile the model
model_cnn3.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

# To do: Copy weights from model_from_D2.
```

```
#Reference: https://keras.io/2.15/api/models/model_saving_apis/
 ↪weights_saving_and_loading/
model_cnn3.set_weights(model_from_D2.get_weights())


# To do: Preprocess the data

# Preprocess the new dataset (which only has trucks and airplanes)
X_train1 = X_train1.astype('float32') / 255.0
X_val1 = X_val1.astype('float32') / 255.0
X_test1 = X_test1.astype('float32') / 255.0

# To do: Train this model (10 epochs)
# Train the cloned model on the new dataset
history_cnn3 = model_cnn3.fit(X_train1,
                              y_train1,
                              epochs=10,
                              validation_data=(X_val1, y_val1))
```

```
Epoch 1/10
219/219              15s 54ms/step -
accuracy: 0.8540 - loss: 0.3373 - val_accuracy: 0.9137 - val_loss: 0.2230
Epoch 2/10
219/219              13s 59ms/step -
accuracy: 0.9114 - loss: 0.2211 - val_accuracy: 0.9247 - val_loss: 0.1898
Epoch 3/10
219/219              18s 81ms/step -
accuracy: 0.9250 - loss: 0.1789 - val_accuracy: 0.8620 - val_loss: 0.3400
Epoch 4/10
219/219              17s 79ms/step -
accuracy: 0.9372 - loss: 0.1639 - val_accuracy: 0.9213 - val_loss: 0.2223
Epoch 5/10
219/219              12s 53ms/step -
accuracy: 0.9409 - loss: 0.1454 - val_accuracy: 0.9387 - val_loss: 0.1594
Epoch 6/10
219/219              11s 52ms/step -
accuracy: 0.9538 - loss: 0.1231 - val_accuracy: 0.9380 - val_loss: 0.1744
Epoch 7/10
219/219              15s 67ms/step -
accuracy: 0.9558 - loss: 0.1124 - val_accuracy: 0.9177 - val_loss: 0.2385
Epoch 8/10
219/219              11s 51ms/step -
accuracy: 0.9500 - loss: 0.1214 - val_accuracy: 0.9333 - val_loss: 0.1849
Epoch 9/10
219/219              10s 47ms/step -
accuracy: 0.9640 - loss: 0.0962 - val_accuracy: 0.8983 - val_loss: 0.3538
Epoch 10/10
219/219              11s 49ms/step -
```

```
accuracy: 0.9674 - loss: 0.0830 - val_accuracy: 0.9217 - val_loss: 0.2592
```

[27]:
```python
#To do: Evaluate performance
np.random.seed(0)
tf.random.set_seed(0)
ku.set_random_seed(0)
random.seed(0)

# Evaluate the updated model's performance
performance_cnn3 = model_cnn3.evaluate(X_test1, y_test1)
```

```
63/63            1s 7ms/step -
accuracy: 0.9195 - loss: 0.2629
```

[28]:
```python
from sklearn.metrics import accuracy_score, precision_score
from sklearn.metrics import recall_score, classification_report

np.random.seed(0)
tf.random.set_seed(0)
ku.set_random_seed(0)
random.seed(0)

# Generate predictions for the updated CNN model
y_pred_cnn3 = model_cnn3.predict(X_test1)
y_pred_cnn3 = (y_pred_cnn3 > 0.5).astype('int32')

# Calculate accuracy, precision, and recall
accuracy_cnn3 = accuracy_score(y_test1, y_pred_cnn3)
precision_cnn3 = precision_score(y_test1, y_pred_cnn3)
recall_cnn3 = recall_score(y_test1, y_pred_cnn3)

# Print the metrics
print(f"Updated CNN Model Test Accuracy: {accuracy_cnn3:.4f}")
print(f"Precision: {precision_cnn3:.4f}")
print(f"Recall: {recall_cnn3:.4f}")

# Print a detailed classification report
print(classification_report(y_test1, y_pred_cnn3, target_names=['Airplane',␣
↪'Truck']))
```

```
63/63            1s 10ms/step
Updated CNN Model Test Accuracy: 0.9225
Precision: 0.8880
Recall: 0.9670
              precision    recall  f1-score   support

    Airplane       0.96      0.88      0.92      1000
       Truck       0.89      0.97      0.93      1000
```

```
    accuracy                           0.92      2000
   macro avg       0.93      0.92      0.92      2000
weighted avg       0.93      0.92      0.92      2000
```

[29]:

```
### APPROACH 2
```

[30]:

```
## Helper code: load pre-trained model.
# Feel free to load something else.
## Available options can be found here:
# https://keras.io/api/applications/#keras-applications

from keras.layers import GlobalAveragePooling2D, Dense, Dropout
from keras.applications import ResNet50
from keras.applications import ResNet50
base_model_1 = ResNet50(include_top = False,␣
↪weights='imagenet',input_shape=(32,32,3))

np.random.seed(0)
tf.random.set_seed(0)
ku.set_random_seed(0)
random.seed(0)


## To Do: Freeze the weights
for layer in base_model_1.layers:
    layer.trainable = False

## Now initialize a new model --
# add the pre-trained weights, along with some additional layers.
# Hint / helper code --
# here's one way to do this, but feel free to use your own.
# model_1= Sequential()
# model_1.add(base_model_1)
# model_1.add(Flatten())

# To Do: Add new dense layers along with Dropout etc. Add at least 2 dense␣
↪layers -- you are free to pick the number of nodes.
# Remember to finish with the classification head (i.e Dense layer with 1 node␣
↪and sigmoid activation. )

model_1 = Sequential([
    base_model_1,
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    Dropout(0.25),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
```

```
    BatchNormalization(),
    Dropout(0.25),
    GlobalAveragePooling2D(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(36, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

## To Do: Compile the Model
model_1.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

[31]:

```
# To do: print the model summary.
# Ensure that weights for the pre-trained model are frozen.

model_1.summary()
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
resnet50 (Functional)	?	23,587,712
conv2d_3 (Conv2D)	?	0 (unbuilt)
batch_normalization_2 (BatchNormalization)	?	0 (unbuilt)
dropout_3 (Dropout)	?	0
conv2d_4 (Conv2D)	?	0 (unbuilt)
batch_normalization_3 (BatchNormalization)	?	0 (unbuilt)
dropout_4 (Dropout)	?	0
global_average_pooling2d (GlobalAveragePooling2D)	?	0 (unbuilt)
dense_8 (Dense)	?	0 (unbuilt)
dropout_5 (Dropout)	?	0

```
 dense_9 (Dense)                  ?                            0 (unbuilt)

 dropout_6 (Dropout)              ?                                      0

 dense_10 (Dense)                 ?                            0 (unbuilt)
```

Total params: 23,587,712 (89.98 MB)

Trainable params: 0 (0.00 B)

Non-trainable params: 23,587,712 (89.98 MB)

[32]:
```python
## To do: Fit the model for 10 epochs.
from tensorflow.keras.applications.resnet50 import preprocess_input

#I used the resnet50 preprocessing function because it returned
#slightly higher accuracy, but overall the results are still close to random␣
 ↪guessing

X_train1_resized = preprocess_input(X_train1) #X_train1 / 255.0
X_val1_resized =  preprocess_input(X_val1) #X_val1 / 255.0
X_test1_resized = preprocess_input(X_test1) #X_test1 / 255.0

history_11 = model_1.fit(X_train1_resized,
                         y_train1, epochs=10,
                         validation_data=(X_val1_resized, y_val1))
```

```
Epoch 1/10
219/219              46s 152ms/step -
accuracy: 0.6234 - loss: 0.6863 - val_accuracy: 0.5050 - val_loss: 0.7171
Epoch 2/10
219/219              34s 157ms/step -
accuracy: 0.7868 - loss: 0.4752 - val_accuracy: 0.5680 - val_loss: 0.7573
Epoch 3/10
219/219              32s 146ms/step -
accuracy: 0.8023 - loss: 0.4511 - val_accuracy: 0.5047 - val_loss: 2.3068
Epoch 4/10
219/219              30s 137ms/step -
accuracy: 0.8154 - loss: 0.4287 - val_accuracy: 0.4997 - val_loss: 3.6310
Epoch 5/10
219/219              32s 144ms/step -
accuracy: 0.8238 - loss: 0.4252 - val_accuracy: 0.6053 - val_loss: 0.7821
Epoch 6/10
219/219              31s 143ms/step -
```

```
accuracy: 0.8291 - loss: 0.4090 - val_accuracy: 0.5987 - val_loss: 0.8104
Epoch 7/10
219/219            30s 139ms/step -
accuracy: 0.8385 - loss: 0.4001 - val_accuracy: 0.5073 - val_loss: 1.9906
Epoch 8/10
219/219            29s 130ms/step -
accuracy: 0.8367 - loss: 0.4038 - val_accuracy: 0.8103 - val_loss: 0.4690
Epoch 9/10
219/219            31s 142ms/step -
accuracy: 0.8331 - loss: 0.4113 - val_accuracy: 0.4963 - val_loss: 4.2676
Epoch 10/10
219/219            31s 140ms/step -
accuracy: 0.8402 - loss: 0.3967 - val_accuracy: 0.5153 - val_loss: 1.5551
```

[33]:

```
#To do: Evaluate model performance

performance_1 = model_1.evaluate(X_test1_resized, y_test1)

# Generate predictions for the transfer learning model
y_pred_transfer = model_1.predict(X_test1_resized)
y_pred_transfer = (y_pred_transfer > 0.5).astype('int32')

# Calculate accuracy, precision, and recall
accuracy_transfer = accuracy_score(y_test1, y_pred_transfer)
precision_transfer = precision_score(y_test1, y_pred_transfer)
recall_transfer = recall_score(y_test1, y_pred_transfer)

# Print the metrics
print(f"Transfer Learning Model Test Accuracy: {accuracy_transfer:.4f}")
print(f"Precision: {precision_transfer:.4f}")
print(f"Recall: {recall_transfer:.4f}")

# Print a detailed classification report
print(classification_report(y_test1, y_pred_transfer, target_names=['Airplane',␣
 ↪'Truck']))
```

```
63/63            7s 104ms/step -
accuracy: 0.5306 - loss: 1.4928
63/63            12s 146ms/step
Transfer Learning Model Test Accuracy: 0.5225
Precision: 0.5115
Recall: 1.0000
              precision    recall  f1-score   support

    Airplane       1.00      0.04      0.09      1000
       Truck       0.51      1.00      0.68      1000

    accuracy                           0.52      2000
```

```
   macro avg       0.76      0.52      0.38      2000
weighted avg       0.76      0.52      0.38      2000
```

The first approach is much more time efficient and results in much higher accuracy. While the second approach can be tuned to get higher accuracy, it will take longer for additional tuning, and is thus less efficient than going with the first approach.