Clayton Brittan - Deep Learning Tiny Image Classifier - 4/8/2024

DataSet: https://www.cs.toronto.edu/~kriz/cifar.html

Reference: Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.

Topic

My project is a Deep Learning Image Classification Model, that aims to classify 32x32 images into 10 different subclasses. The dataset I am using is called Cifar10 and is a subset of the 80 Million tiny images dataset. This project aims to be able to take in an arbitrary 32x32 image and classify it accordingly. This model attempt to solve a image classification problem using Keras, a tensor flow library adept at creating visual classification models.

This project is important because it allows anyone to identify the subject matter of very small photos accurately. A 32x32 is very small and because of it's limited information it may be hard to discern what the photo actually is, this model will be able to accurately predict what the photo is of. As well this model is able to take in new 32x32 photos and classify them accordingly. This project is important to me as I want to further my understanding of deep learning and this project served as a great introduction to image classification. With this model as a baseline you can incorporate even more classes and larger image sizes so that the model could classify any arbitrary image the user inputs.

My motivation behind this project was to be able to build a deep learning model for classification, these types of models are incredibly useful in many different aspects of society. One example is for security or criminal identification, if you can train the program on individuals faces it could identify criminals from cctv footage. That is just one example of the capabilities of Deep learning classification models, they have incredible potential which drove me to creating this project

Libraries

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import PIL
from PIL import Image
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.datasets import cifar10
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

Data Cleaning and EDA

Our primary concerns with Data Cleaning and EDA for our model is to make sure the data is in the correct form to be passed into the neural network. Luckily this dataset is part of the tensorflow/keras datasets library, so we can use this to automatically import and split our data along the predefined batches laid out by the dataset author. This saves us time and formatting for our image data as it already is in proper formatting. So our first step is to split the data into testing and training sets respectively

The image data is formatted as a 32x32 array where each entry in the array represents an RGB value in the form of a tuple (R, G, B). These values are by default between 0 and 255 so we need to normalize them to get them between 0-1. This is because we are using min-max activation functions, and these function operate optimally with normalized data. If we were to leave our data in the ranges of 0-255, the model may be less capable of understanding complex patterns in the data.

Now that we have loaded and normalized our data we can inspect our label values, and what you will see is that each image has a label in the form of a single element list, with its value being between 0-10 where each integer represent a different class. We need 10 neurons for our output layer, one representing each class, where our output is a probability distribution of each possible class. Since our output is a probability distribution of length 10, we cannot simply pass in and integer value as our labels. Therefor we need to One-Hot Encode our labels to be of the same form as our output layer. This means we will translate each label value into an array of size 10 where:

```
Array = { 1 if label;  0 if else }
```

Next we can check our data for missing values quickly with a boolean search, since there are no Null values we can continue.

Next we can print out the shape of the data, and what we expect it to look like

Then we can print out some examples of the data to get a better understanding of what the model is seeing.

Load and Split Data

In this step we load the data in using the keras.datasets library, this automatically formats our data into the training and testing set which saves alot of work.

We then One Hot Encode our label data so that it is in the correct form for our output layer of our neural network. If we leave it as an integer, our target and output variables would be mismatched. This way we can still retain the class information however now its reflecting the the index rather than the value.

As well we initialize our number of classes to use in our output layer

We also normalize the data values as they are intially in the form of RGB values which fall in the range (0,255) by diving them by their range we obtain values between 0-1. This allows our model to be more reliable.

In [ ]:
```
# Initialize data and split it using tensorflow datasets
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

num_classes = len(np.unique(y_train))
print(num_classes)


# One Hot encoding label data
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)


# Normalize the RGB values from 0-255 to 0-1
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255

print(y_train[0])
```

```
10
[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

Check for Null values

In [ ]:
```
# Boolean conditional check for Null values
Do_We_Have_Null_Values = None in X_train and None in X_test and None in y_train and None in y_test
if Do_We_Have_Null_Values == False:
    print("There are no Null Values in any dataset (X_train, X_test, y_train, y_test)")
else:
    print("Yall got so many Null values, lol couldn't be me.")
```

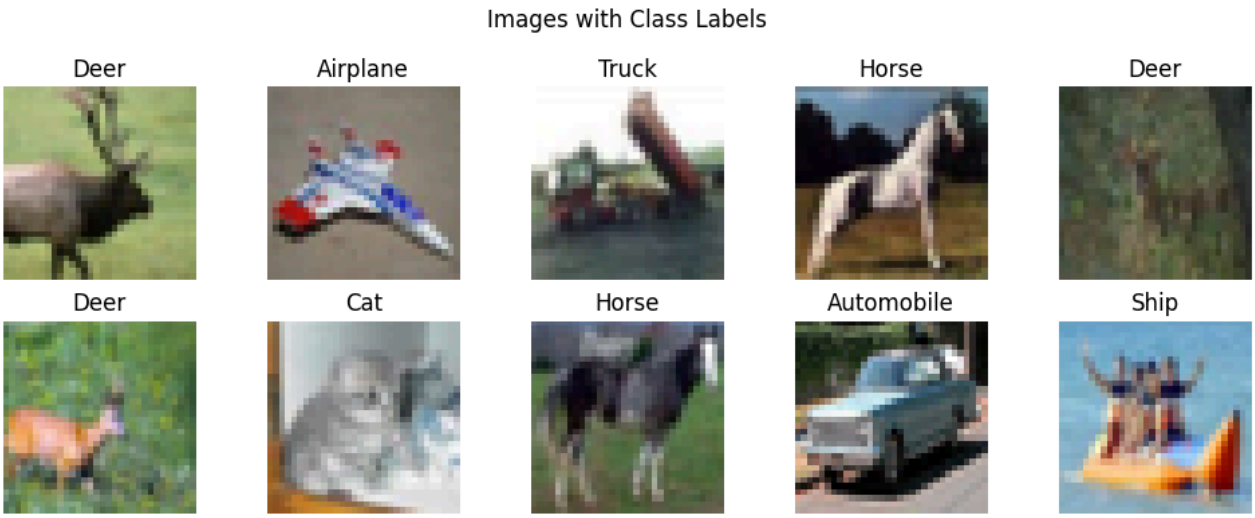
```
There are no Null Values in any dataset (X_train, X_test, y_train, y_test)
```

Inspect Shape of Data and Plot Examples

In [ ]:
```
import random
# Print the shape and format of data
data_shape = X_test[0].shape

print(f"Each entry of the data is of size {data_shape[0]} by {data_shape[1]} where each entry contains {data_shape[

print(f"Each entry represents an image of {data_shape[0]**2} pixels that can be classified into {num_classes} diffe

print(f"\n Here are some examples of the images in the dataset")


# Initialize class names
class_names = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck']

# Initialize figure to 2 rows of 5 axes
```

```
fig, axes = plt.subplots(2, 5, figsize=(10, 4))
# Initialize random indexes for phot selection
randList = random.sample(range(100), 10)

# Initialize loop to plot image and label
for i in range(10):
    image = X_train[randList[i]]

    row = i // 5
    col = i % 5

    axes[row, col].imshow(image)

    class_index = np.argmax(y_train[randList[i]])

    axes[row, col].set_title(class_names[class_index])
    axes[row, col].axis('off')
# Give figure title
fig.suptitle('Images with Class Labels', fontsize=12)
plt.tight_layout()
plt.show()
```

```
Each entry of the data is of size 32 by 32 where each entry contains 3 values

Each entry represents an image of 1024 pixels that can be classified into 10 different classes

 Here are some examples of the images in the dataset
```



Model Building / Model Choice

For our image classification we are using a Convolutional Nueral Network, this model was chosen as it allows for multiple convolution layers and complete hands on training and optimizing experience. There are other models used for image classification such as VGG16 however these models are pre-trained so you simply optimize it for your specific data needs, but it does not allow for as much creativity and freedom as creating your own CNN. I compare different CNN models based on the amount and type of layers as well as optimization techniques.

Model Building

Our first model is initialized with the input size, then we introduce a convolution layer to learn new features, we intialize it with a small number as it is our first layer, we choose 16. Then we reduce the dimensionality with a Pooling layer, Then we flatten the layers into a one dimensional vector which we then hit it with a Dense layer, initialized with a low value to reduce overfilling, and then we reach our output layer which uses softmax to produce our probability distribution for the label.

In [ ]:
```
# Initialize model
model = Sequential([
  layers.Input((32, 32, 3)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
```

```
  layers.Dense(16, activation='relu'),
  layers.Dense(10, activation='softmax')
])
```

In [ ]:
```
# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 16)	448
max_pooling2d (MaxPooling2D)	(None, 16, 16, 16)	0
flatten (Flatten)	(None, 4096)	0
dense (Dense)	(None, 16)	65,552
dense_1 (Dense)	(None, 10)	170

Total params: 66,170 (258.48 KB)

Trainable params: 66,170 (258.48 KB)

Non-trainable params: 0 (0.00 B)

Model Training

Next we fit the model on the training data, and itilialize it with 30 epochs, after the 30th epoch the model is fitted and we have an accuracy of accuracy: 0.7175 - loss: 0.7899

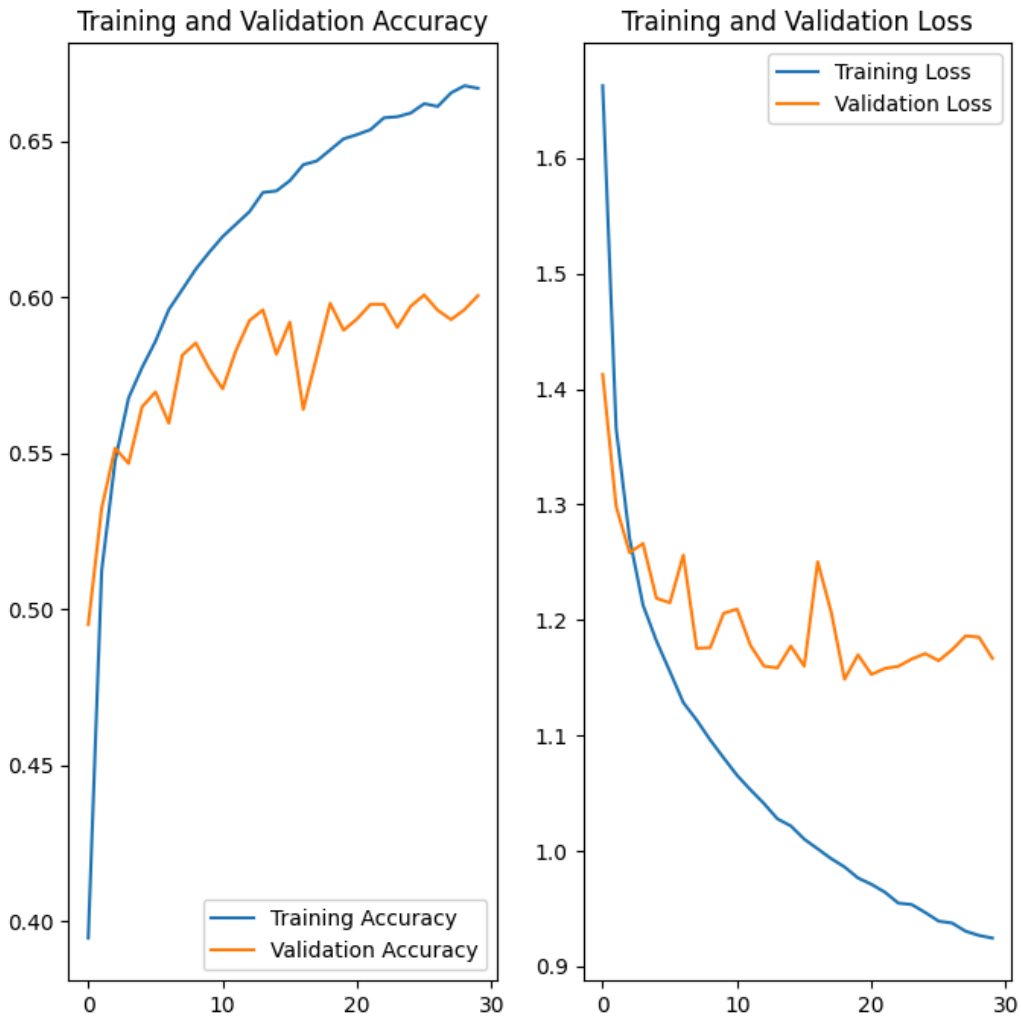
In [ ]:
```
# Fit the model
epochs=30
history = model.fit(
  X_train,
  y_train,
  validation_data= (X_test, y_test),
  epochs=epochs
)
```

```
Epoch 1/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.3092 - loss: 1.8695 - val_accuracy: 0.4951 - val_loss: 1.41
26
Epoch 2/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.5053 - loss: 1.3893 - val_accuracy: 0.5324 - val_loss: 1.29
83
Epoch 3/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.5456 - loss: 1.2801 - val_accuracy: 0.5516 - val_loss: 1.25
83
Epoch 4/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.5691 - loss: 1.2148 - val_accuracy: 0.5467 - val_loss: 1.26
61
Epoch 5/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.5762 - loss: 1.1806 - val_accuracy: 0.5648 - val_loss: 1.21
87
Epoch 6/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.5875 - loss: 1.1448 - val_accuracy: 0.5696 - val_loss: 1.21
48
Epoch 7/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.5971 - loss: 1.1227 - val_accuracy: 0.5596 - val_loss: 1.25
62
Epoch 8/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6015 - loss: 1.1074 - val_accuracy: 0.5814 - val_loss: 1.17
53
Epoch 9/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6149 - loss: 1.0800 - val_accuracy: 0.5853 - val_loss: 1.17
59
Epoch 10/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6169 - loss: 1.0785 - val_accuracy: 0.5771 - val_loss: 1.20
57
Epoch 11/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6225 - loss: 1.0525 - val_accuracy: 0.5706 - val_loss: 1.20
94
Epoch 12/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6270 - loss: 1.0453 - val_accuracy: 0.5830 - val_loss: 1.17
79
Epoch 13/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6318 - loss: 1.0297 - val_accuracy: 0.5925 - val_loss: 1.16
00
Epoch 14/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6333 - loss: 1.0254 - val_accuracy: 0.5959 - val_loss: 1.15
84
Epoch 15/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6399 - loss: 1.0117 - val_accuracy: 0.5817 - val_loss: 1.17
73
Epoch 16/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6433 - loss: 0.9949 - val_accuracy: 0.5920 - val_loss: 1.15
99
Epoch 17/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6454 - loss: 0.9939 - val_accuracy: 0.5640 - val_loss: 1.25
03
Epoch 18/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6485 - loss: 0.9754 - val_accuracy: 0.5809 - val_loss: 1.20
69
Epoch 19/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6475 - loss: 0.9799 - val_accuracy: 0.5980 - val_loss: 1.14
86
Epoch 20/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6551 - loss: 0.9686 - val_accuracy: 0.5894 - val_loss: 1.16
97
Epoch 21/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6577 - loss: 0.9588 - val_accuracy: 0.5930 - val_loss: 1.15
28
Epoch 22/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6544 - loss: 0.9559 - val_accuracy: 0.5977 - val_loss: 1.15
79
Epoch 23/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6612 - loss: 0.9479 - val_accuracy: 0.5977 - val_loss: 1.15
98
Epoch 24/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6603 - loss: 0.9484 - val_accuracy: 0.5903 - val_loss: 1.16
61
Epoch 25/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6634 - loss: 0.9382 - val_accuracy: 0.5971 - val_loss: 1.17
```

```
08
Epoch 26/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6682 - loss: 0.9256 - val_accuracy: 0.6007 - val_loss: 1.16
46
Epoch 27/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6643 - loss: 0.9295 - val_accuracy: 0.5958 - val_loss: 1.17
42
Epoch 28/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6726 - loss: 0.9138 - val_accuracy: 0.5928 - val_loss: 1.18
61
Epoch 29/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6714 - loss: 0.9144 - val_accuracy: 0.5960 - val_loss: 1.18
51
Epoch 30/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.6728 - loss: 0.9083 - val_accuracy: 0.6005 - val_loss: 1.16
67
```

In [ ]:

```python
# Get accuracy and val_accuracy from history
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

# Get loss and val_loss from history
loss = history.history['loss']
val_loss = history.history['val_loss']

# Set range to number of epochs
epochs_range = range(epochs)

# Plot data and label axes and title
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()
```



In [ ]:
```
# Print loss and accuracy
loss, accuracy = model.evaluate(X_test, y_test)
print('Test accuracy:', accuracy)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6012 - loss: 1.1542
Test accuracy: 0.6004999876022339
```

Model Building

For our improved model we implment strategies to mitigate the lose and improve the accuracy. As you can tell from the graphs above our model would have benifited from more epochs as it hasn't plateaud. So for this model we will run it for more epochs.

As well we have implemented more layers including 4 augmentation layers being two flips: Horizontal, and Vertical, A 10% rotation and a 10% zoom. These augmentation layers allow the model to have better general performance.

We also increase the number of convolution layers from 1 to 3, with pooling layers between them, and finally a flattening layer which leads to our Dense layer followed by our output layer which remain unchanged.

In [ ]:
```
# Initialize model
model_improved = keras.Sequential(
  [
    layers.Input((32,32,3)),
    layers.RandomFlip("horizontal"),
    layers.RandomFlip("vertical"),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1),
    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(64, activation='relu'),
    layers.Dense(10, activation='softmax')
  ]
)
```

In [ ]:
```
# Compile the model
model_improved.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Optimizers

Here we experimented with different optimization functions:

SGD Produced a final 10/10 Epoch result of:

```
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 12s 8ms/step - accuracy: 0.5552 - loss: 1.2473 - val_accuracy:
0.5917 - val_loss: 1.1627
```

Where as Adam produced a final 10/10 Epoch result of:

```
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 12s 8ms/step - accuracy: 0.6944 - loss: 0.8643 - val_accuracy:
0.7070 - val_loss: 0.8386
```

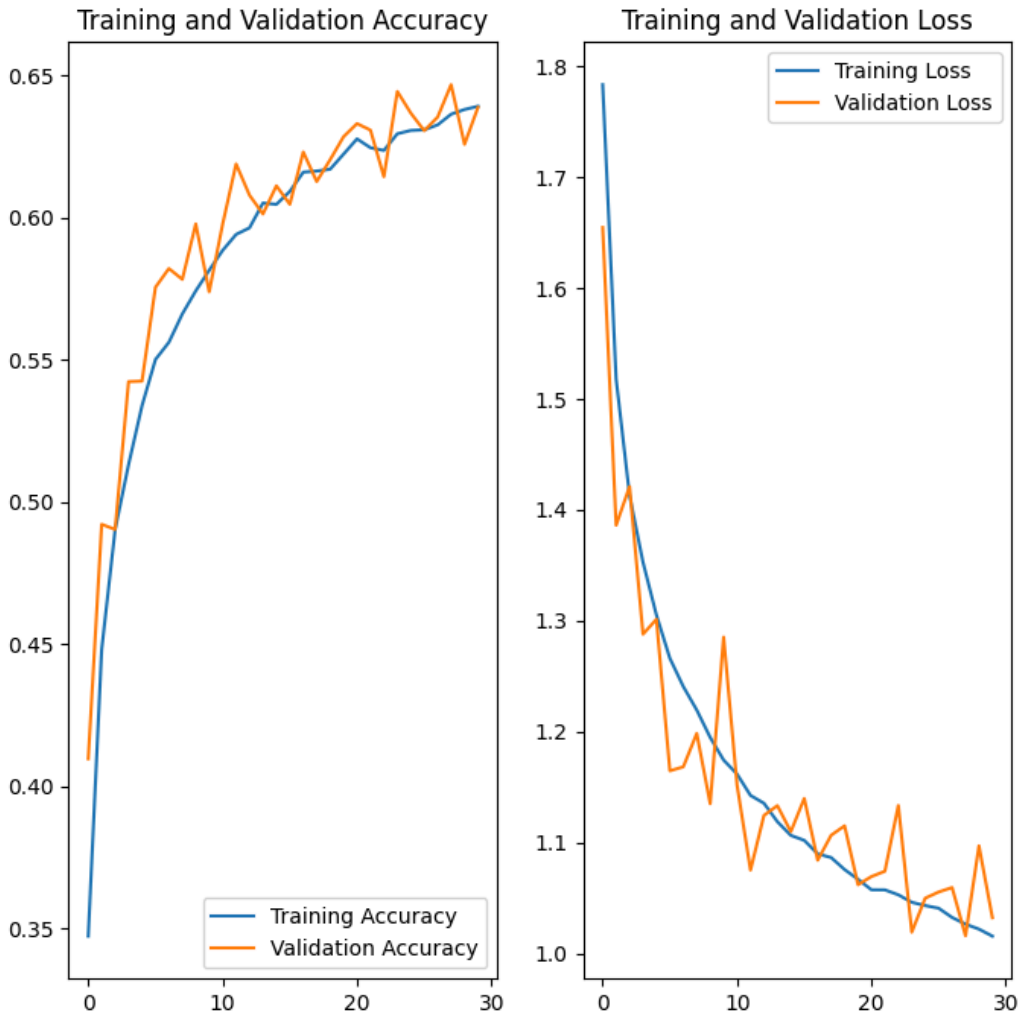
In [ ]:
```
# Fit the model
epochs=30
history2 = model_improved.fit(
  X_train,
  y_train,
  validation_data= (X_test, y_test),
  epochs=epochs
)
```

```
Epoch 1/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 15s 8ms/step - accuracy: 0.2725 - loss: 1.9665 - val_accuracy: 0.4096 - val_loss: 1.6
546
Epoch 2/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.4313 - loss: 1.5558 - val_accuracy: 0.4921 - val_loss: 1.3
860
Epoch 3/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.4850 - loss: 1.4274 - val_accuracy: 0.4903 - val_loss: 1.4
210
Epoch 4/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.5117 - loss: 1.3627 - val_accuracy: 0.5423 - val_loss: 1.2
878
Epoch 5/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.5305 - loss: 1.3087 - val_accuracy: 0.5425 - val_loss: 1.3
013
Epoch 6/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.5455 - loss: 1.2751 - val_accuracy: 0.5755 - val_loss: 1.1
646
Epoch 7/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.5547 - loss: 1.2469 - val_accuracy: 0.5821 - val_loss: 1.1
682
Epoch 8/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.5700 - loss: 1.2122 - val_accuracy: 0.5783 - val_loss: 1.1
983
Epoch 9/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.5718 - loss: 1.2006 - val_accuracy: 0.5978 - val_loss: 1.1
349
Epoch 10/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.5829 - loss: 1.1717 - val_accuracy: 0.5739 - val_loss: 1.2
852
Epoch 11/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.5860 - loss: 1.1667 - val_accuracy: 0.5977 - val_loss: 1.1
516
Epoch 12/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.5914 - loss: 1.1492 - val_accuracy: 0.6189 - val_loss: 1.0
750
Epoch 13/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.5973 - loss: 1.1351 - val_accuracy: 0.6079 - val_loss: 1.1
244
Epoch 14/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6059 - loss: 1.1189 - val_accuracy: 0.6013 - val_loss: 1.1
332
Epoch 15/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6026 - loss: 1.1150 - val_accuracy: 0.6112 - val_loss: 1.1
097
Epoch 16/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6134 - loss: 1.1003 - val_accuracy: 0.6047 - val_loss: 1.1
397
Epoch 17/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6159 - loss: 1.0881 - val_accuracy: 0.6231 - val_loss: 1.0
840
Epoch 18/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6198 - loss: 1.0803 - val_accuracy: 0.6127 - val_loss: 1.1
065
Epoch 19/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6177 - loss: 1.0711 - val_accuracy: 0.6204 - val_loss: 1.1
150
Epoch 20/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6227 - loss: 1.0656 - val_accuracy: 0.6285 - val_loss: 1.0
619
Epoch 21/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.6277 - loss: 1.0568 - val_accuracy: 0.6331 - val_loss: 1.0
692
Epoch 22/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6262 - loss: 1.0510 - val_accuracy: 0.6308 - val_loss: 1.0
741
Epoch 23/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6202 - loss: 1.0610 - val_accuracy: 0.6144 - val_loss: 1.1
334
Epoch 24/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6260 - loss: 1.0525 - val_accuracy: 0.6444 - val_loss: 1.0
191
Epoch 25/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6272 - loss: 1.0514 - val_accuracy: 0.6368 - val_loss: 1.0
```

```
498
Epoch 26/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6327 - loss: 1.0380 - val_accuracy: 0.6306 - val_loss: 1.0
553
Epoch 27/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6339 - loss: 1.0299 - val_accuracy: 0.6354 - val_loss: 1.0
593
Epoch 28/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6384 - loss: 1.0271 - val_accuracy: 0.6468 - val_loss: 1.0
158
Epoch 29/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6379 - loss: 1.0255 - val_accuracy: 0.6258 - val_loss: 1.0
969
Epoch 30/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.6366 - loss: 1.0203 - val_accuracy: 0.6388 - val_loss: 1.0
321
```

In [ ]:

```python
# Get accuracy and val_accuracy from history
acc = history2.history['accuracy']
val_acc = history2.history['val_accuracy']

# Get loss and val_loss from history
loss = history2.history['loss']
val_loss = history2.history['val_loss']

# Set range to number of epochs
epochs_range = range(epochs)

# Plot data and label axes and title
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()
```



In [ ]:
```
# Get accuracy and loss
loss, accuracy = model_improved.evaluate(X_test, y_test)
print('Test accuracy:', accuracy)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.6415 - loss: 1.0246
Test accuracy: 0.6388000249862671
```

Analysis

As you can see our accuracy is only 0.572 and our loss is around 1.2 if we compare this to our previous model with only 1 convolution layer we can see that is out preforms this current model.

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6048 - loss: 1.1413
Test accuracy: 0.5996000170707703
```

So we need to refine our model even more, and perhaps we over-fit it with too many layers.

In [ ]:
```
# Initialize Model
modelV3 = keras.Sequential(
  [
    layers.Input((32,32,3)),
    layers.Conv2D(16, kernel_size=3, activation='relu', padding='same'),
    layers.MaxPooling2D(),
    layers.Conv2D(32, kernel_size=3, activation='relu', padding='same'),
    layers.MaxPooling2D(),
    layers.Conv2D(64, kernel_size=3, activation='relu', padding='same'),
    layers.MaxPooling2D(),
    layers.Conv2D(128, kernel_size=3, activation='relu', padding='same'),
    layers.MaxPooling2D(),
    layers.Conv2D(256, kernel_size=3, activation='relu', padding='same'),
    layers.MaxPooling2D(),
```

```
    layers.Flatten(),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')

  ]
)
```

In [ ]:
```
# Compile Model
modelV3.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In [ ]:
```
# Fit Model
epochs=30
history3 = modelV3.fit(
  X_train,
  y_train,
  validation_data= (X_test, y_test),
  epochs=epochs
)
```

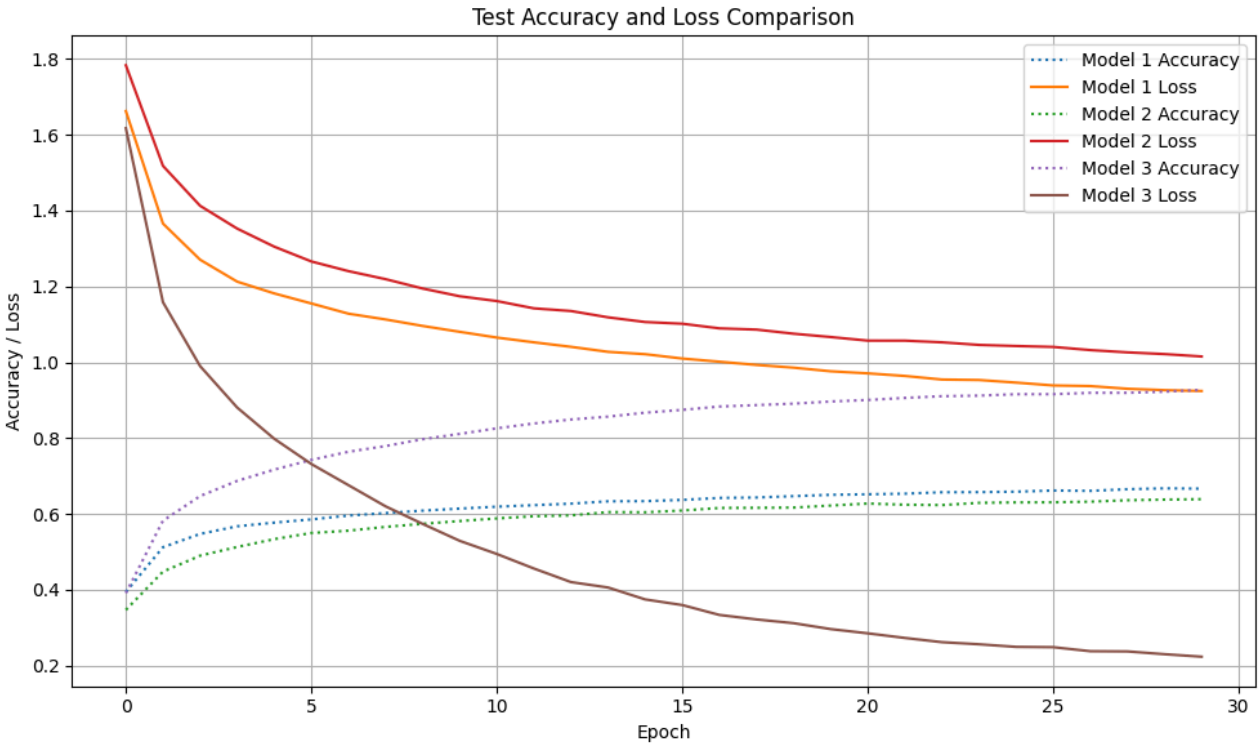
```
Epoch 1/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 19s 11ms/step - accuracy: 0.2735 - loss: 1.8939 - val_accuracy: 0.5325 - val_loss: 1.
2835
Epoch 2/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.5661 - loss: 1.1959 - val_accuracy: 0.5846 - val_loss: 1.
1489
Epoch 3/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.6400 - loss: 1.0046 - val_accuracy: 0.6371 - val_loss: 1.
0247
Epoch 4/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.6880 - loss: 0.8813 - val_accuracy: 0.6869 - val_loss: 0.
9060
Epoch 5/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7187 - loss: 0.7923 - val_accuracy: 0.6393 - val_loss: 1.
0574
Epoch 6/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7459 - loss: 0.7231 - val_accuracy: 0.6902 - val_loss: 0.
9303
Epoch 7/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7645 - loss: 0.6682 - val_accuracy: 0.7064 - val_loss: 0.
8699
Epoch 8/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7850 - loss: 0.6028 - val_accuracy: 0.7054 - val_loss: 0.
9087
Epoch 9/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.8034 - loss: 0.5505 - val_accuracy: 0.7077 - val_loss: 0.
8950
Epoch 10/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.8233 - loss: 0.4980 - val_accuracy: 0.7018 - val_loss: 0.
9521
Epoch 11/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.8360 - loss: 0.4672 - val_accuracy: 0.7125 - val_loss: 0.
9755
Epoch 12/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.8459 - loss: 0.4353 - val_accuracy: 0.7066 - val_loss: 1.
0701
Epoch 13/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.8559 - loss: 0.4035 - val_accuracy: 0.6986 - val_loss: 1.
0898
Epoch 14/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.8678 - loss: 0.3722 - val_accuracy: 0.7038 - val_loss: 1.
1295
Epoch 15/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.8754 - loss: 0.3512 - val_accuracy: 0.7113 - val_loss: 1.
1458
Epoch 16/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.8861 - loss: 0.3292 - val_accuracy: 0.6949 - val_loss: 1.
1917
Epoch 17/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.8941 - loss: 0.3035 - val_accuracy: 0.7014 - val_loss: 1.
2542
Epoch 18/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.8956 - loss: 0.2969 - val_accuracy: 0.7024 - val_loss: 1.
2481
Epoch 19/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9001 - loss: 0.2833 - val_accuracy: 0.6872 - val_loss: 1.
3535
Epoch 20/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9038 - loss: 0.2748 - val_accuracy: 0.6902 - val_loss: 1.
4771
Epoch 21/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9065 - loss: 0.2698 - val_accuracy: 0.6944 - val_loss: 1.
4567
Epoch 22/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9118 - loss: 0.2523 - val_accuracy: 0.7023 - val_loss: 1.
4851
Epoch 23/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9188 - loss: 0.2418 - val_accuracy: 0.7089 - val_loss: 1.
5144
Epoch 24/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9169 - loss: 0.2440 - val_accuracy: 0.6992 - val_loss: 1.
5016
Epoch 25/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9227 - loss: 0.2307 - val_accuracy: 0.6956 - val_loss: 1.
```

```
6351
Epoch 26/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9232 - loss: 0.2307 - val_accuracy: 0.7029 - val_loss: 1.
4971
Epoch 27/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9266 - loss: 0.2140 - val_accuracy: 0.6887 - val_loss: 1.
6754
Epoch 28/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9258 - loss: 0.2213 - val_accuracy: 0.6988 - val_loss: 1.
6545
Epoch 29/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9275 - loss: 0.2159 - val_accuracy: 0.7028 - val_loss: 1.
5635
Epoch 30/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9337 - loss: 0.2078 - val_accuracy: 0.6990 - val_loss: 1.
8069
```

Results and Analysis

In [ ]:

```python
# Function to plot each model's history, takes in Model.history value and the desired label
def plot_history(history, label):
    plt.plot(history.history['accuracy'], label=label+' Accuracy', linestyle = ':')
    plt.plot(history.history['loss'], label=label+' Loss')
    plt.legend()

# Get model data from history
model1_history = history
model2_history = history2
model3_history = history3

# Size figure
plt.figure(figsize=(10, 6))

# Plot the model data using plot_history
plot_history(model1_history, 'Model 1')
plot_history(model2_history, 'Model 2')
plot_history(model3_history, 'Model 3')

# Plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Accuracy / Loss')
plt.title('Test Accuracy and Loss Comparison')

# Show grid and plot
plt.grid(True)
plt.tight_layout()
plt.show()
```



Analysis

From the graph we can see that our third model produces the best accuracy and minimizes the loss, by choosing the right layers and the correct amount of layers we can improve our model's performance. The third model surpassed the other two as it had proper depth, as well as utilizing greater convolution and this is evident when we evaluate our models on the test data. We can see that with our third model the accuracy increased however our loss is very high. This is likely due to our model being very good at classifying the training data but lacking in general performance with unseen data. To improve this we could introduce augmentation layers.

The metric chosen for evaluation were accuracy and loss as since we are dealing with image classification our primary concern is if we are classifying images correctly. The best way to measure this is to use the accuracy and loss measurements on our models.

In [ ]:
```
# Get accuracy and loss for each model and print

lossV1, accuracyV1 = model.evaluate(X_test, y_test)
lossV2, accuracyV2 = model_improved.evaluate(X_test, y_test)
lossV3, accuracyV3 = modelV3.evaluate(X_test, y_test)
print(f"The accuracy for the model 1 is {accuracyV1} and the loss for the model is {lossV1} \n")
print(f"The accuracy for the model 2 is {accuracyV2} and the loss for the model is {lossV2} \n")
print(f"The accuracy for the model 3 is {accuracyV3} and the loss for the model is {lossV3} \n")
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 927us/step - accuracy: 0.6012 - loss: 1.1542
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6415 - loss: 1.0246
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7021 - loss: 1.7902
The accuracy for the model 1 is 0.6004999876022339 and the loss for the model is 1.1666947603225708

The accuracy for the model 2 is 0.6388000249862671 and the loss for the model is 1.0321449041366577

The accuracy for the model 3 is 0.6990000009536743 and the loss for the model is 1.8068647384643555
```

Discussion and Conclusion

This project has shown me that there are many variables at play when trying to build a Convolution Neural Network, and they types of layers and the width and depth of the layers plays a tremendous role on the models efficacy. The optimization of these layers is also another task, so trying to optimize CNN's is a very intensive process. This process would have been sped up a lot with GPU utilization however I could not install the proper Nvidia drivers to allow me to use my GPU which significantly slowed

down model evaluation and testing. With GPU utilization I would have been able to tweak the layers and hyperparameters more as I would have spent less time waiting for the model to be fitted.

Even still, I am happy with the results as the model is able to properly classify images a majority of the time and it's even able to classify this image I just got off of google

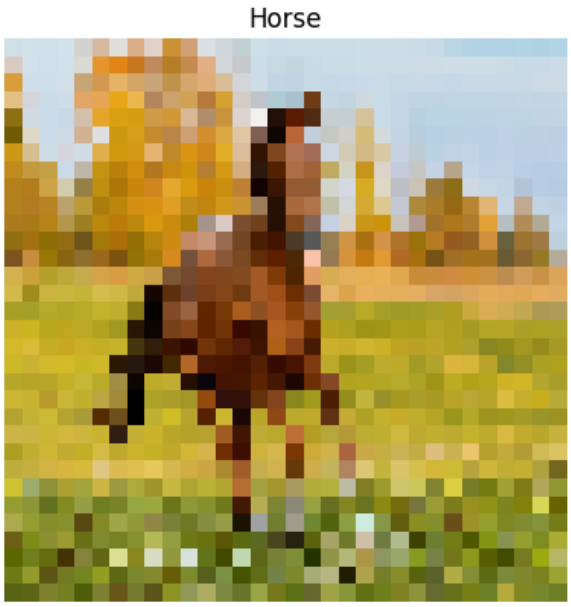
(https://www.tastingtable.com/img/gallery/ive-tried-horse-meat-here-are-my-thoughts/l-intro-1682539768.jpg).

 Horsey

In [ ]:
```
# Open custom Image for model testing
image = Image.open("Horsey.png")

# Format Image data for model
image = image.convert('RGB')
image_data = np.array(image)
image_data_batched= image_data.reshape((1,32, 32, 3))
image_data_batched = image_data_batched.astype('float32') / 255.0


# Get Prediction from Model and assign it's class value to the label
prediction = modelV3.predict(image_data_batched)
prediction = np.argmax(prediction)
print(prediction)

# Show image and give it a title
prediction = class_names[prediction]
plt.title(prediction)
plt.axis('off')
plt.imshow(image_data)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 88ms/step
7
```

Out[ ]: <matplotlib.image.AxesImage at 0x1e952a246b0>



Extra

Added augmentation Layer and it increased the accuracy while decreasing the loss as expected, this Model is significantly more versatile that the last.

In [ ]:
```
model_final = keras.Sequential(
  [
    layers.Input((32,32,3)),
    layers.RandomFlip("vertical"),
    layers.RandomFlip("horizontal"),
```

```
    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.Conv2D(128, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(256, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(10, activation='softmax')
  ]
)
```

In [ ]:
```
model_final.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In [ ]:
```
epochs=30
history_final = model_final.fit(
  X_train,
  y_train,
  validation_data= (X_test, y_test),
  epochs=epochs
)
```

```
Epoch 1/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 19s 11ms/step - accuracy: 0.2720 - loss: 1.9100 - val_accuracy: 0.4794 - val_loss: 1.
3852
Epoch 2/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.4888 - loss: 1.3839 - val_accuracy: 0.5402 - val_loss: 1.
2717
Epoch 3/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.5618 - loss: 1.2158 - val_accuracy: 0.5688 - val_loss: 1.
1668
Epoch 4/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 17s 11ms/step - accuracy: 0.5974 - loss: 1.1159 - val_accuracy: 0.6141 - val_loss: 1.
0818
Epoch 5/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.6389 - loss: 1.0135 - val_accuracy: 0.6192 - val_loss: 1.
0693
Epoch 6/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.6515 - loss: 0.9747 - val_accuracy: 0.6405 - val_loss: 1.
0118
Epoch 7/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.6651 - loss: 0.9348 - val_accuracy: 0.6586 - val_loss: 0.
9573
Epoch 8/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.6870 - loss: 0.8818 - val_accuracy: 0.6687 - val_loss: 0.
9282
Epoch 9/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.6958 - loss: 0.8551 - val_accuracy: 0.6613 - val_loss: 0.
9697
Epoch 10/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.6999 - loss: 0.8431 - val_accuracy: 0.6735 - val_loss: 0.
9289
Epoch 11/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7131 - loss: 0.8079 - val_accuracy: 0.6800 - val_loss: 0.
9151
Epoch 12/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7194 - loss: 0.7863 - val_accuracy: 0.6833 - val_loss: 0.
9095
Epoch 13/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7309 - loss: 0.7687 - val_accuracy: 0.7005 - val_loss: 0.
8688
Epoch 14/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7362 - loss: 0.7409 - val_accuracy: 0.6944 - val_loss: 0.
8893
Epoch 15/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7455 - loss: 0.7178 - val_accuracy: 0.6942 - val_loss: 0.
8876
Epoch 16/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7501 - loss: 0.7095 - val_accuracy: 0.7029 - val_loss: 0.
8795
Epoch 17/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7509 - loss: 0.7007 - val_accuracy: 0.6941 - val_loss: 0.
9013
Epoch 18/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7567 - loss: 0.6837 - val_accuracy: 0.7119 - val_loss: 0.
8743
Epoch 19/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7627 - loss: 0.6673 - val_accuracy: 0.7071 - val_loss: 0.
8947
Epoch 20/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7667 - loss: 0.6480 - val_accuracy: 0.7016 - val_loss: 0.
8818
Epoch 21/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7678 - loss: 0.6471 - val_accuracy: 0.6934 - val_loss: 0.
9331
Epoch 22/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7776 - loss: 0.6363 - val_accuracy: 0.7105 - val_loss: 0.
8588
Epoch 23/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7814 - loss: 0.6174 - val_accuracy: 0.7125 - val_loss: 0.
8815
Epoch 24/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7786 - loss: 0.6227 - val_accuracy: 0.7041 - val_loss: 0.
8918
Epoch 25/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 17s 11ms/step - accuracy: 0.7803 - loss: 0.6129 - val_accuracy: 0.7129 - val_loss: 0.
```

```
8589
Epoch 26/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7859 - loss: 0.5987 - val_accuracy: 0.7100 - val_loss: 0.
8714
Epoch 27/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7848 - loss: 0.6057 - val_accuracy: 0.7175 - val_loss: 0.
8449
Epoch 28/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 17s 11ms/step - accuracy: 0.7888 - loss: 0.5902 - val_accuracy: 0.7123 - val_loss: 0.
8598
Epoch 29/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7943 - loss: 0.5778 - val_accuracy: 0.6987 - val_loss: 0.
9158
Epoch 30/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.7995 - loss: 0.5670 - val_accuracy: 0.7143 - val_loss: 0.
8692
```

In [ ]:
```
loss_final, accuracy_final = model_final.evaluate(X_test, y_test)
print(f"The accuracy for the model 1 is {accuracy_final} and the loss for the model is {loss_final} \n")
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7134 - loss: 0.8750
The accuracy for the model 1 is 0.7142999768257141 and the loss for the model is 0.8691927790641785
```

In [ ]:
```
# Open custom Image for model testing
image = Image.open("Horsey.png")

# Format Image data for model
image = image.convert('RGB')
image_data = np.array(image)
image_data_batched= image_data.reshape((1,32, 32, 3))
image_data_batched = image_data_batched.astype('float32') / 255.0


# Get Prediction from Model and assign it's class value to the label
prediction = model_final.predict(image_data_batched)
prediction = np.argmax(prediction)
print(prediction)

# Show image and give it a title
prediction = class_names[prediction]
plt.title(prediction)
plt.axis('off')
plt.imshow(image_data)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 152ms/step
7
```

Out[ ]: <matplotlib.image.AxesImage at 0x1e93a4e2cf0>

In [ ]:
```
def plot_history(history, label):
    plt.plot(history.history['accuracy'], label=label+' Accuracy', linestyle=':')
    plt.plot(history.history['loss'], label=label+' Loss')
    plt.legend()

# Get model data from history
model1_history = history
model2_history = history2
model3_history = history3
model_final_history = history_final

# Size figure
plt.figure(figsize=(10, 6))

# Plot the model data using plot_history
plot_history(model1_history, 'Model 1')
plot_history(model2_history, 'Model 2')
plot_history(model3_history, 'Model 3')
plot_history(model_final_history, 'Model Final')

# Plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Accuracy / Loss')
plt.title('Test Accuracy and Loss Comparison')

# Show grid and plot
plt.grid(True)
plt.tight_layout()
plt.show()
```