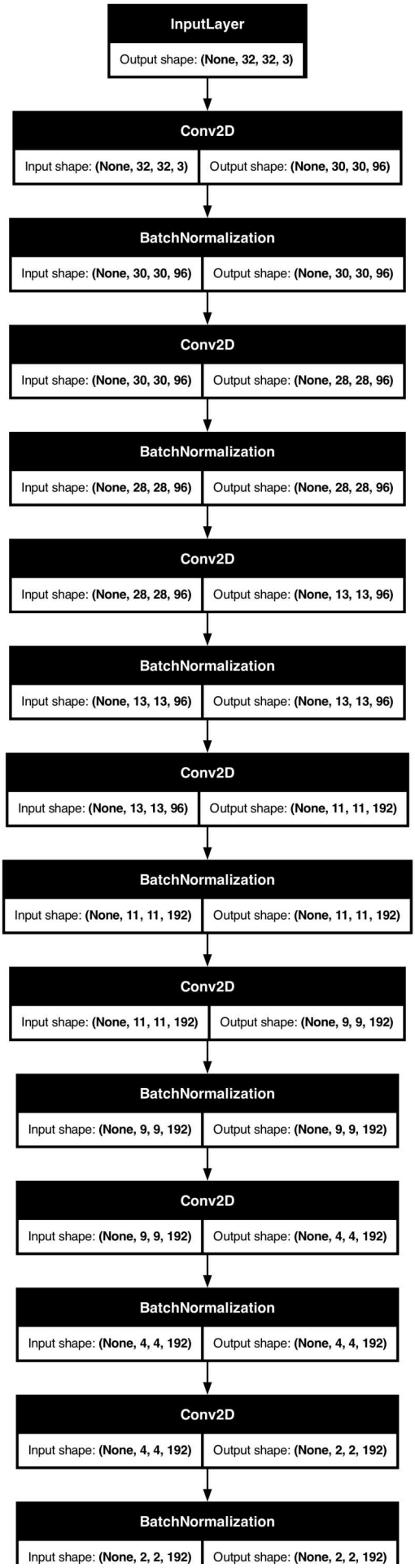
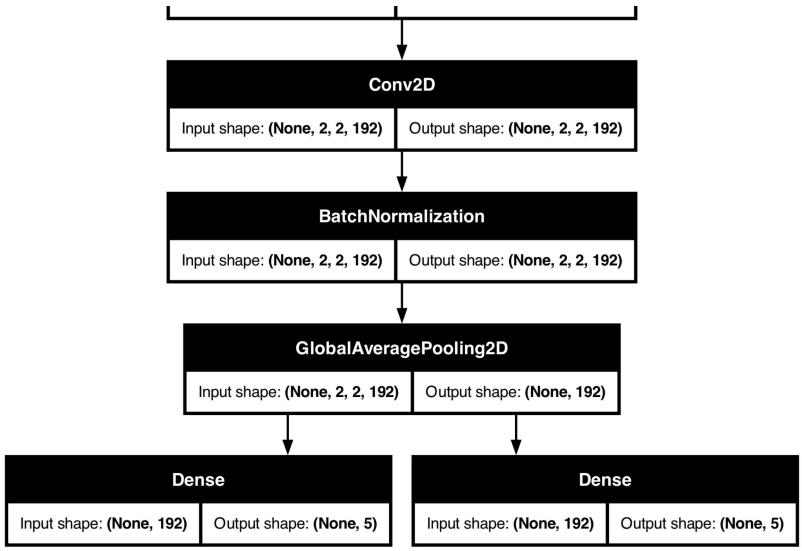
Learning to CIFAR Back!

The CIFAR-10 dataset is a benchmark for computer vision tasks dating to 2009. We construct a model capable of disentangling an input image created by averaging two random samples from CIFAR-10 and predicting the categories of the two components. We achieve 77.2% accuracy with a near identical F1 Score, which demonstrates the balanced nature of our dataset and model predictions.

This novel task requires training on small datasets (50,000 examples) with small spatial resolution (32x32x3). We overcome this difficulty by eliminating max pooling operations in favor of an All Convolutional Model, inspired by the second place CIFAR-10 holders. The writeup takes the form of a Jupyter notebook, first displaying our most capable model, then chronologically detailing how we arrived to this result.

77% Accuracy All CNN Model


InputLayer
Output shape: (None, 32, 32, 3)
Conv2D
Input shape: (None, 32, 32, 3)
Output shape: (None, 30, 30, 96)
BatchNormalization
Input shape: (None, 30, 30, 96)
Output shape: (None, 30, 30, 96)
Conv2D
Input shape: (None, 30, 30, 96)
Output shape: (None, 28, 28, 96)
BatchNormalization
Input shape: (None, 28, 28, 96)
Output shape: (None, 28, 28, 96)
Conv2D
Input shape: (None, 28, 28, 96)
Output shape: (None, 13, 13, 96)
BatchNormalization
Input shape: (None, 13, 13, 96)
Output shape: (None, 13, 13, 96)
Conv2D
Input shape: (None, 13, 13, 96)
Output shape: (None, 11, 11, 192)
BatchNormalization
Input shape: (None, 11, 11, 192)
Output shape: (None, 11, 11, 192)
Conv2D
Input shape: (None, 11, 11, 192)
Output shape: (None, 9, 9, 192)
BatchNormalization
Input shape: (None, 9, 9, 192)
Output shape: (None, 9, 9, 192)
Conv2D
Input shape: (None, 9, 9, 192)
Output shape: (None, 4, 4, 192)
BatchNormalization
Input shape: (None, 4, 4, 192)
Output shape: (None, 4, 4, 192)
Conv2D
Input shape: (None, 4, 4, 192)
Output shape: (None, 2, 2, 192)
BatchNormalization
Input shape: (None, 2, 2, 192)
Output shape: (None, 2, 2, 192)



Description

The model has a feature extractor composed purely of two dimensional convolutions. Instead of maxpooling, down sampling the spatial resolution is accomplished by a stride of 2. The extractor reduces the resolution to a stack of 192 2x2 filters. This is flattened and passed to two independent classifiers. These are single layer fully connected layers, with a softmax to produce the 5 possible class probabilities. The network takes advantage of Batch Norm layers to prevent overfitting and no data augmentation, as it did not lead to better results. We optimize with RMSProp and by reducing learning rate on plateau. Each classifier has its own cross entropy loss to back propagate through the network, and we observe they do not improve equally, an area for further research on the task.

Experimental Setup

The reader is invited to train locally by uncommenting the **fit()** calls where indicated.

Libraries

In [16]:
```
import tensorflow as tf
import keras
from tensorflow.keras import layers, regularizers
from tensorflow.keras.layers import LeakyReLU
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import cifar10
from tensorflow.nn import fractional_max_pool
from tensorflow.keras import backend as K
import numpy as np
from matplotlib import pyplot as plt
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import Callback
from keras.saving import load_model
from sklearn.metrics import f1_score, accuracy_score
```

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

Data Preparation

In [17]:
```
(cifar10_x_train, cifar10_y_train), (cifar10_x_test, cifar10_y_test) = cifar10.load
assert cifar10_x_train.shape == (50000, 32, 32, 3)
assert cifar10_x_test.shape == (10000, 32, 32, 3)
assert cifar10_y_train.shape == (50000, 1)
assert cifar10_y_test.shape == (10000, 1)

# Split 10 percent of the training set into validation set
cifar10_x_train, cifar10_x_val, cifar10_y_train, cifar10_y_val = train_test_split(
    cifar10_x_train, cifar10_y_train, test_size=0.1
)

# First classifier: "airplane", "automobile", "bird", "cat", "deer"
# Second classifier: "dog", "frog", "horse", "ship", "truck"
classes = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse",

# Normalizing to range (0,1)
cifar10_x_train = (cifar10_x_train/255.).astype(np.float32)
cifar10_x_val = (cifar10_x_val / 255.).astype(np.float32)
cifar10_x_test = (cifar10_x_test/255.).astype(np.float32)
```

Split the images in three groups, according to their label.

In [3]:
```
cond_1 = cifar10_y_train[:,0] < 5
cifar10_x_train_1 = cifar10_x_train[cond_1]
cifar10_y_train_1 = cifar10_y_train[cond_1]

cond_2 = cifar10_y_train[:,0] >= 5
cifar10_x_train_2 = cifar10_x_train[cond_2]
cifar10_y_train_2 = cifar10_y_train[cond_2]

cond_1_val = cifar10_y_val[:,0] < 5
cifar10_x_val_1 = cifar10_x_val[cond_1_val]
cifar10_y_val_1 = cifar10_y_val[cond_1_val]

cond_2_val = cifar10_y_val[:,0] >= 5
cifar10_x_val_2 = cifar10_x_val[cond_2_val]
cifar10_y_val_2 = cifar10_y_val[cond_2_val]

cond_1_test = cifar10_y_test[:,0] < 5
cifar10_x_test_1 = cifar10_x_test[cond_1_test]
cifar10_y_test_1 = cifar10_y_test[cond_1_test]

cond_2_test = cifar10_y_test[:,0] >= 5
cifar10_x_test_2 = cifar10_x_test[cond_2_test]
cifar10_y_test_2 = cifar10_y_test[cond_2_test]
```

Define the generators. The input consists of two datasets (X1,X2), their corresponding labels (Y1,Y2), and a batch size.

The generator returns (x_data,y_data), where:

- x_data is a batch of images obtained by averaging random samples from X1 and X2.
- y_data is a dictionary of batches of labels corresponding to the component images, expressed in categorical format, in other words, one hot encoded.

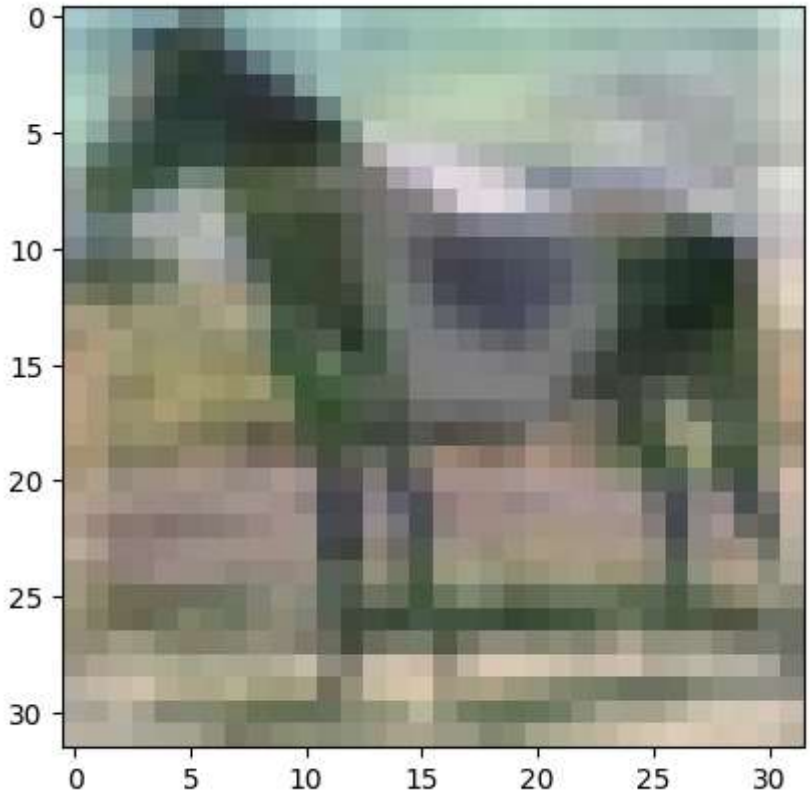
In [18]:
```
batchsize = 64

def datagenerator(X1,X2,Y1,Y2,batchsize):
  size1 = X1.shape[0]
  size2 = X2.shape[0]
  Y1_cat = tf.keras.utils.to_categorical(Y1, num_classes=5)
  Y2_cat = tf.keras.utils.to_categorical(Y2-5, num_classes=5)

  while True:
    num1 = np.random.randint(0, size1, batchsize)
    num2 = np.random.randint(0, size2, batchsize)
    x_data = (X1[num1] + X2[num2]) / 2.0
    y_data = {'output1': Y1_cat[num1], 'output2': Y2_cat[num2]}
    yield tf.convert_to_tensor(x_data), y_data

traingen = datagenerator(cifar10_x_train_1,cifar10_x_train_2,cifar10_y_train_1,cifa
valgen = datagenerator(cifar10_x_val_1,cifar10_x_val_2,cifar10_y_val_1,cifar10_y_va
testgen = datagenerator(cifar10_x_test_1,cifar10_x_test_2,cifar10_y_test_1,cifar10_
```

We generate an example, display the image that the model will take as input, and print the categories of the two overlapping components.

The reader is invited to re-run the cell to display new examples.

In [29]:
```
datagen = datagenerator(cifar10_x_train_1,cifar10_x_train_2,cifar10_y_train_1,cifar
x, y = next(datagen)
print("first: {}, second: {}".format(classes[np.argmax(y['output1'][0])],classes[np
plt.imshow(x[0]);
```

```
first: bird, second: horse
```



Model Evaluation

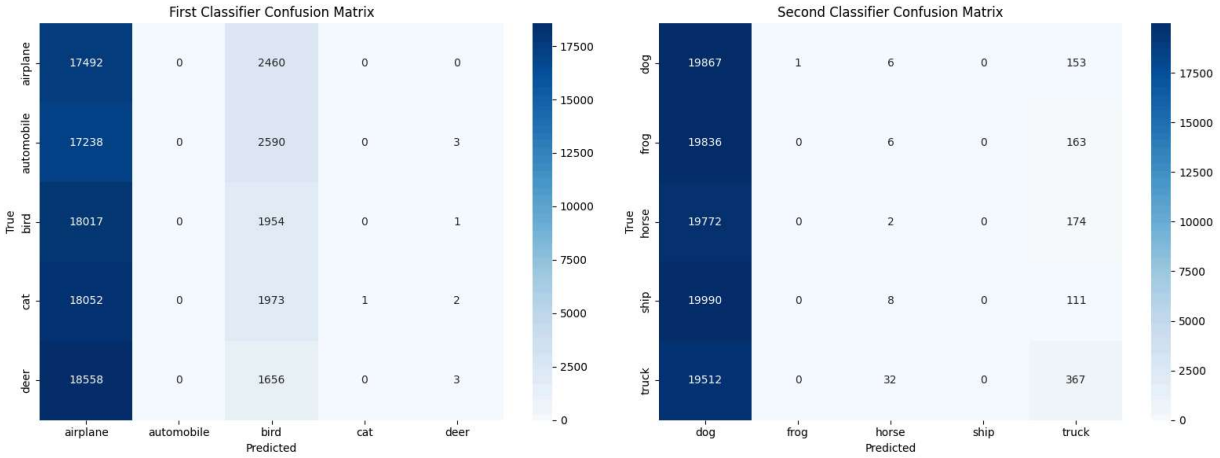
We create our evaluation method and test it with a model which generates random classes.

In [30]:
```
def random_model():
    inputs = keras.Input(shape=(32,32,3))
    x = layers.Flatten()(inputs)
    output1 = layers.Dense(5, name='output1')(x)
    output2 = layers.Dense(5, name='output2')(x)
    model = keras.Model(
    inputs,
    outputs={'output1':output1, 'output2':output2}, name='Toy',
    )
    model.compile(
    optimizer='Adam',
    loss={'output1': 'categorical_crossentropy', 'output2': 'categorical_crossentro
    metrics={'output1':'accuracy', 'output2':'accuracy'})
    return model

def evaluate_model(model, gen, steps=10, repeat=10, classes=classes):
    y1 = []
    pred1 = []
    y2 = []
    pred2 = []

    for _ in tqdm(range(steps * repeat)):
        # Generator returns 1000 image batches
        x, y = next(gen)
```

```
        y1.extend(np.argmax(y['output1'], axis=1))
        y2.extend(np.argmax(y['output2'], axis=1))

        pred = model.predict(x, verbose=0)
        pred1.extend(np.argmax(pred['output1'], axis=1))
        pred2.extend(np.argmax(pred['output2'], axis=1))

    acc1 = accuracy_score(y1, pred1)
    acc2 = accuracy_score(y2, pred2)
    mean_accuracy = np.mean([acc1, acc2])
    std_accuracy = np.std([acc1, acc2])
    print(f"First Classifier Accuracy: {acc1:.4f}")
    print(f"Second Classifier Accuracy: {acc2:.4f}")
    print(f"Mean Accuracy: {mean_accuracy:.4f} ± {std_accuracy:.4f}")

    f1_1 = f1_score(y1, pred1, average='weighted')
    f1_2 = f1_score(y2, pred2, average='weighted')
    mean_f1 = np.mean([f1_1, f1_2])
    std_f1 = np.std([f1_1, f1_2])
    print(f"First Classifier F1 Score: {f1_1:.4f}")
    print(f"Second Classifier F1 Score: {f1_2:.4f}")
    print(f"Mean F1 Score: {mean_f1:.4f} ± {std_f1:.4f}")

    cm1 = confusion_matrix(y1, pred1)
    cm2 = confusion_matrix(y2, pred2)
    fig, axes = plt.subplots(1, 2, figsize=(16, 6))

    sns.heatmap(cm1, annot=True, fmt='d', cmap='Blues',
                xticklabels=classes[:5] if classes else None,
                yticklabels=classes[:5] if classes else None, ax=axes[0])
    axes[0].set_xlabel('Predicted')
    axes[0].set_ylabel('True')
    axes[0].set_title('First Classifier Confusion Matrix')

    sns.heatmap(cm2, annot=True, fmt='d', cmap='Blues',
                xticklabels=classes[5:10] if classes else None,
                yticklabels=classes[5:10] if classes else None, ax=axes[1])
    axes[1].set_xlabel('Predicted')
    axes[1].set_ylabel('True')
    axes[1].set_title('Second Classifier Confusion Matrix')

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

In [34]:
```
model = random_model()
evaluate_model(model, testgen)
```

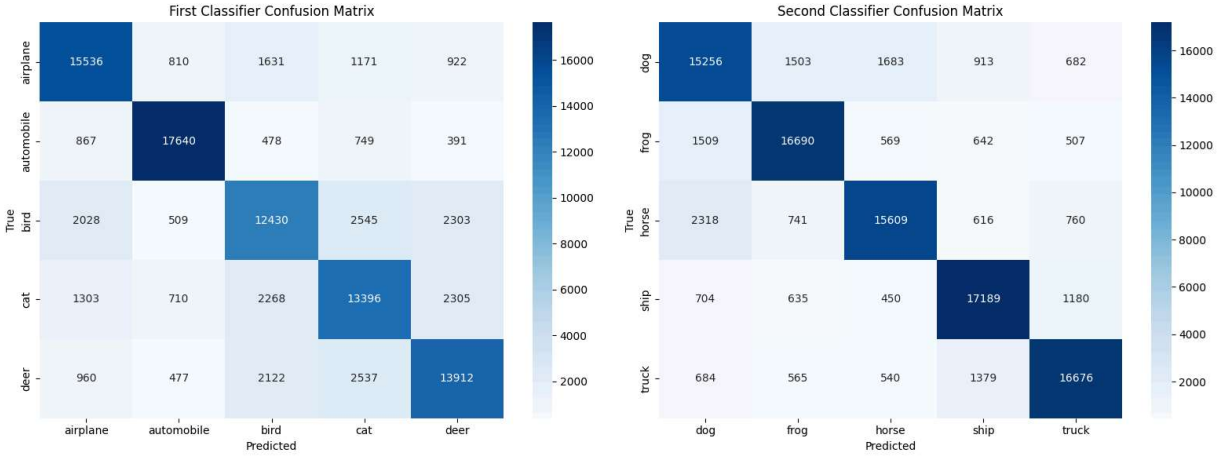
```
100%|██████████| 100/100 [00:21<00:00,  4.56it/s]
First Classifier Accuracy: 0.1945
Second Classifier Accuracy: 0.2024
Mean Accuracy: 0.1984 ± 0.0039
First Classifier F1 Score: 0.0894
Second Classifier F1 Score: 0.0739
Mean F1 Score: 0.0817 ± 0.0078
```



As expected, the accuracy is around 1/5 = 0.2! Our function repeats the evaluation over the entire test set ten times, and computes the standard deviation. This is how we arrive at our official project result. No more waiting, we load our top model weights to report the results!

In [10]:
```
model = load_model('77.2acc.keras')
evaluate_model(model, testgen)
```

```
100%|██████████| 100/100 [02:17<00:00,  1.38s/it]
First Classifier Accuracy: 0.7291
Second Classifier Accuracy: 0.8142
Mean Accuracy: 0.7717 ± 0.0425
First Classifier F1 Score: 0.7287
Second Classifier F1 Score: 0.8142
Mean F1 Score: 0.7715 ± 0.0427
```



As stated in the beginning, our model performs admirably with an accuracy of 77.2%! From the confusion matrices, we can see it performs about equally well on all classes. However, the first classifier has greater difficulty than the second. We know this from the lower accuracy score and confusion between animals: notice the shaded areas for deer, cat, and bird!

Model Checkpoints

We implement a custom model callback to save the model weights when the average accuracy of both classifiers improves during training. Without this, we would be forced to use loss as a proxy for the metric we care about the most. We care about the validation accuracy since it is representative of the unseen data the model will see during testing. This being said, we did not use the save checkpoint often since training was fast and the session reliable!

In [35]:
```
class MeanAccModelCheckpoint(Callback):
    def __init__(self, filepath, monitor1='val_output1_accuracy', monitor2='val_out
        super(MeanAccModelCheckpoint, self).__init__()
        self.filepath = filepath
        self.monitor1 = monitor1
        self.monitor2 = monitor2
        self.mode = mode
        self.verbose = verbose
        self.best_score = -float('inf') if mode == 'max' else float('inf')

    # Called at the end of an epoch during training
    def on_epoch_end(self, epoch, logs=None):
        acc1 = logs.get(self.monitor1)
        acc2 = logs.get(self.monitor2)

        # Compute the average accuracy
        avg_accuracy = (acc1 + acc2) / 2 if acc1 is not None and acc2 is not None e

        if avg_accuracy is not None:
            if (self.mode == 'max' and avg_accuracy > self.best_score) or \
               (self.mode == 'min' and avg_accuracy < self.best_score):
                # Update the best score
                self.best_score = avg_accuracy

                # Format the filepath
                save_path = self.filepath.format(
                    output1_accuracy=acc1,
                    output2_accuracy=acc2,
                    epoch=epoch + 1,  # Epoch is zero-indexed
                    loss=logs.get('loss')
                )

                # Save the model
                if self.verbose:
                    print(f"\nEpoch {epoch + 1}: Average accuracy improved to {avg_
                self.model.save(save_path)
            elif self.verbose:
                print(f"\nEpoch {epoch + 1}: Average accuracy did not improve (curr
```

We use the built in callback for early stopping so we do not have to worry about training for too few or too many epochs. When the validation loss stops decreasing, the model will try ten more times to improve or revert to the best weights. The validation loss metric is an average of the validation losses of each one of the two classifiers. We also use a built in callback to lower the learning rate when training stalls, to converge to better results.

In [36]:
```
early_stopping = keras.callbacks.EarlyStopping(monitor='val_loss',
                                   patience=10,
                                   mode='min',
                                   restore_best_weights=True,
                                   verbose=1)
reduce_lr = keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
                                   factor=0.2,
                                   patience=5,
                                   min_lr=1e-9,
                                   verbose=1)
```

Model Creation

We use the Keras Functional API to create our model. This is the model whose architecture we plotted in the beginning, and which performed the best.

In [37]:
```
def ACCNRR():
    # CIFAR-10
    inputs = keras.Input(shape=(32,32,3))
    inputs = layers.RandomFlip("horizontal")(inputs)
    # Feature Extractor
    x = layers.Conv2D(96, (3,3), activation='relu')(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(96, (3,3), activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(96, (3,3), activation='relu', strides=2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation='relu', strides=2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (1,1), activation='relu')(x)
    x = layers.BatchNormalization()(x)
    # Classifers
    x = layers.GlobalAveragePooling2D()(x)
    output1 = layers.Dense(5, activation='softmax', name='output1', kernel_regulari
    output2 = layers.Dense(5, activation='softmax', name='output2', kernel_regulari
    # Model creation
    model = keras.Model(
        inputs,
        outputs={'output1':output1, 'output2':output2}, name='ACCNRR',
        )
    return model
```

We use the garbage collector to free all memory, then load and compile the model with its hyperparamters. The optimizer is RMS Prop, which combined with reducing learning rate over training leads to stable training. We apply label smoothing to the classifier losses, since

the signal is quite noisy this will help with learning. Finally, we display the model summary and ascertain its size as a medium model, at over 1 Million trainable parameters.

In [46]:
```
K.clear_session()
model = ACCNRR()
model.compile(optimizer='rmsprop',
              loss={'output1': keras.losses.CategoricalCrossentropy(label_smoothing
              metrics={'output1': 'accuracy', 'output2': 'accuracy'})


model.summary()
```

Model: "ACCNRR"

Layer (type)	Output Shape	Param #	Connected to
keras_tensor_1CLONE (InputLayer)	(None, 32, 32, 3)	0	-
conv2d (Conv2D)	(None, 30, 30, 96)	2,688	keras_tensor_1CL…
batch_normalization (BatchNormalizatio…	(None, 30, 30, 96)	384	conv2d[1][0]
conv2d_1 (Conv2D)	(None, 28, 28, 96)	83,040	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 28, 28, 96)	384	conv2d_1[1][0]
conv2d_2 (Conv2D)	(None, 13, 13, 96)	83,040	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 13, 13, 96)	384	conv2d_2[1][0]
conv2d_3 (Conv2D)	(None, 11, 11, 192)	166,080	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 11, 11, 192)	768	conv2d_3[1][0]
conv2d_4 (Conv2D)	(None, 9, 9, 192)	331,968	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 9, 9, 192)	768	conv2d_4[1][0]
conv2d_5 (Conv2D)	(None, 4, 4, 192)	331,968	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 4, 4, 192)	768	conv2d_5[1][0]
conv2d_6 (Conv2D)	(None, 2, 2, 192)	331,968	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 2, 2, 192)	768	conv2d_6[1][0]
conv2d_7 (Conv2D)	(None, 2, 2, 192)	37,056	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 2, 2, 192)	768	conv2d_7[1][0]
global_average_poo… (GlobalAveragePool…	(None, 192)	0	batch_normalizat…
output1 (Dense)	(None, 5)	965	global_average_p…

output2 (Dense)	(None, 5)	965	global_average_p…

Total params: 1,374,730 (5.24 MB)

Trainable params: 1,372,234 (5.23 MB)

Non-trainable params: 2,496 (9.75 KB)

Run the code cell below to train it for yourself!

In [50]:
```
# history = model.fit(
#     traingen,
#     epochs=250,
#     steps_per_epoch=len(cifar10_x_train)//batchsize, # necessary since generator
#     validation_data=valgen,
#     validation_steps=len(cifar10_x_val)//batchsize,  # necessary since generator
#     callbacks=[early_stopping, reduce_lr],
#     verbose=1
# )
model = load_model('77.2acc.keras')
```

Evaluate the model for yourself! This will match the results we shared earlier!

In [52]:
```
# evaluate_model(model, testgen)
```

Experimental Setup

As the reader might imagine, one does not simply invent a top performing model. The first step was idea generation and research. A few key points:

- CIFAR-10 is a small dataset
- Images in CIFAR-10 only measure 32x32 pixels
- Classifying CIFAR-10 is a very popular project
- Everything was to be implemented in Keras and TensorFlow

Since CIFAR-10 is a small dataset, the model should be slim, it should have strong regularization to avoid overfitting and the dataset should be augmented to help it generalize. It was also a shame to divide training data to create the validation set, this could be mitigated with a smarter data generator and k fold validation. Since we are required to use Keras Tensorflow, we should take full advantage of its tools, namely the Functional API, so we spend more time experimenting and less troubleshooting. Most importantly, since CIFAR-10 is popular, we would be able to adapt an existing top performer to this task by adding a second classification head.

Research

We googled "CIFAR-10 best accuracy" and fell upon a reddit post about a person trying to train it as fast as possible. However, we are not interested in light speed CIFAR-10, so we

tried again, googling only "CIFAR-10" and finding the main website for this dataset. From there, we discovered the world rankings. Now we will read these papers in order and apply the knowledge within to train our model. The first is Fractional Max Pooling, with an accuracy of 96.53%. As we read this, a doubt strikes us, is the task similar enough? For example, as a human, it is harder distinguish the two classes than just one... Assuredly, the model will struggle more as well.

Fractional Max Pooling

The innovation of this paper is to allow max pooling to reduce the spatial size of the features by less than an integer factor. This will allow slower data compression and make training easier. For example, the paper finds good results with a factor of

$$\sqrt{2} \approx 1.41$$

rather than 2, the lowest integer factor. The authors of the paper use compressed notation to express their CIFAR-10 model.

$$(160nC2 - FMP\sqrt[3]{2})_{12} - C2 - C1 - output$$

It took several iterations to translate in code. Along the way, we decided to subclass keras.Model() rather than use the Functional API since fractional max pooling is a custom layer. The general model architecture is a feature extractor whose filters increases linearly by 160 each layer, followed by the fractional max pool which downsamples by 1.26, repeated 10 times. The C2 represents a 2x2 kernel. At the end, there is a 1x1 kernel to lower the feature maps then a one layer fully connected classifier. The architecture with best performance we obtained is below.

In [53]:
```
@keras.saving.register_keras_serializable()
class FMP3(Model):
    def __init__(self):
        super(FMP3, self).__init__()
        # Input layer size 32x32

        # Six convolutional layers (C2 layers). Each convolutional layer should hav
        self.conv1 = layers.Conv2D(32, (2, 2), activation=LeakyReLU())
        self.conv2 = layers.Conv2D(64, (2, 2), activation=LeakyReLU())
        self.conv3 = layers.Conv2D(96, (2, 2), activation=LeakyReLU())
        self.conv4 = layers.Conv2D(128, (2, 2), activation=LeakyReLU())
        self.conv5 = layers.Conv2D(160, (2, 2), activation=LeakyReLU())
        self.conv6 = layers.Conv2D(192, (2, 2), activation=LeakyReLU())

        # Six fractional max pooling layers
        self.pooling_ratio1 = [1.0, 31/22, 31/22, 1.0]
        self.pooling_ratio2 = [1.0, 21/15, 21/15, 1.0]
        self.pooling_ratio3 = [1.0, 14/10, 14/10, 1.0]
        self.pooling_ratio4 = [1.0, 9/6, 9/6, 1.0]
        self.pooling_ratio5 = [1.0, 5/4, 5/4, 1.0]
        self.pooling_ratio6 = [1.0, 3/2, 3/2, 1.0]
```

```
        # Two final convolutional layers
        self.conv7 = layers.Conv2D(192, (2, 2), activation=LeakyReLU()) # 4x4
        self.conv8 = layers.Conv2D(192, (1, 1), activation=LeakyReLU())

        # Linearly increasing dropout in the hidden layers
        self.dropout = layers.Dropout(0.5)  # 50% dropout in the final hidden layer

        # Two Classifiers
        self.fc1 = layers.Dense(5, activation='softmax', name='output1')  # First h
        self.fc2 = layers.Dense(5, activation='softmax', name='output2')  # Second

    def call(self, inputs):
        # Feature Extractor
        x = self.conv1(inputs) # 31x31
        x, _, _ = fractional_max_pool(x, self.pooling_ratio1, pseudo_random=True, o
        x = self.conv2(x) # 21x21
        x, _, _ = fractional_max_pool(x, self.pooling_ratio2, pseudo_random=True, o
        x = self.conv3(x) # 14x14
        x, _, _ = fractional_max_pool(x, self.pooling_ratio3, pseudo_random=True, o
        x = self.conv4(x) # 9x9
        x, _, _ = fractional_max_pool(x, self.pooling_ratio4, pseudo_random=True, o
        x = self.conv5(x) #5x5
        x, _, _ = fractional_max_pool(x, self.pooling_ratio5, pseudo_random=True, o
        x = self.conv6(x) # 3x3
        x, _, _ = fractional_max_pool(x, self.pooling_ratio6, pseudo_random=True, o
        x = self.conv7(x) # 1x1
        # IDEA: dropout here at 25%
        x = self.conv8(x) #1x1

        # Classifier
        x = layers.Flatten()(x) # future improvement might be to use global average
        x = self.dropout(x)
        output1 = self.fc1(x)  # First half
        output2 = self.fc2(x)  # Second half

        return {'output1': output1, 'output2': output2}

    def get_config(self):
        # Serialize the configuration of the model
        config = super(FMP3, self).get_config()
        config.update({
            'conv1': self.conv1.get_config(),
            'conv2': self.conv2.get_config(),
            'conv3': self.conv3.get_config(),
            'conv4': self.conv4.get_config(),
            'conv5': self.conv5.get_config(),
            'conv6': self.conv6.get_config(),
            'conv7': self.conv7.get_config(),
            'conv8': self.conv8.get_config(),
            'dropout': self.dropout.get_config(),
            'fc1': self.fc1.get_config(),
            'fc2': self.fc2.get_config(),
            'pooling_ratios': [
                self.pooling_ratio1,
                self.pooling_ratio2,
                self.pooling_ratio3,
```

```
                self.pooling_ratio4,
                self.pooling_ratio5,
                self.pooling_ratio6
            ]
        })
        return config

    @classmethod
    def from_config(cls, config):
        return cls()
```

It is substantially smaller than the model from the paper. Since we used a custom max pooling layer, this model was not optimized for CUDA and was CPU bottlenecked during training. Below, we verify the model size is under 500,000 parameters!

In [54]:
```
K.clear_session()
model = FMP3()
model.build(input_shape=(None, 32, 32, 3))
model.compile(optimizer='adam',
              loss={'output1': 'categorical_crossentropy', 'output2': 'categorical_
              metrics={'output1': 'accuracy', 'output2': 'accuracy'})
model(tf.random.normal((1, 32, 32, 3)))
model.summary()
```

```
/home/nicolas/Github/image-seperation/venv/lib/python3.10/site-packages/keras/src/la
yers/layer.py:393: UserWarning: `build()` was called on layer 'fmp3', however the la
yer does not have a `build()` method implemented and it looks like it has unbuilt st
ate. This will cause the layer to be marked as built, despite not being actually bui
lt, which may cause failures down the line. Make sure to implement a proper `build()
` method.
  warnings.warn(
```

Model: "fmp3"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(1, 31, 31, 32)	416
conv2d_1 (Conv2D)	(1, 21, 21, 64)	8,256
conv2d_2 (Conv2D)	(1, 14, 14, 96)	24,672
conv2d_3 (Conv2D)	(1, 9, 9, 128)	49,280
conv2d_4 (Conv2D)	(1, 5, 5, 160)	82,080
conv2d_5 (Conv2D)	(1, 3, 3, 192)	123,072
conv2d_6 (Conv2D)	(1, 1, 1, 192)	147,648
conv2d_7 (Conv2D)	(1, 1, 1, 192)	37,056
dropout (Dropout)	?	0
output1 (Dense)	(1, 5)	965
output2 (Dense)	(1, 5)	965

Total params: 474,410 (1.81 MB)

Trainable params: 474,410 (1.81 MB)

Non-trainable params: 0 (0.00 B)

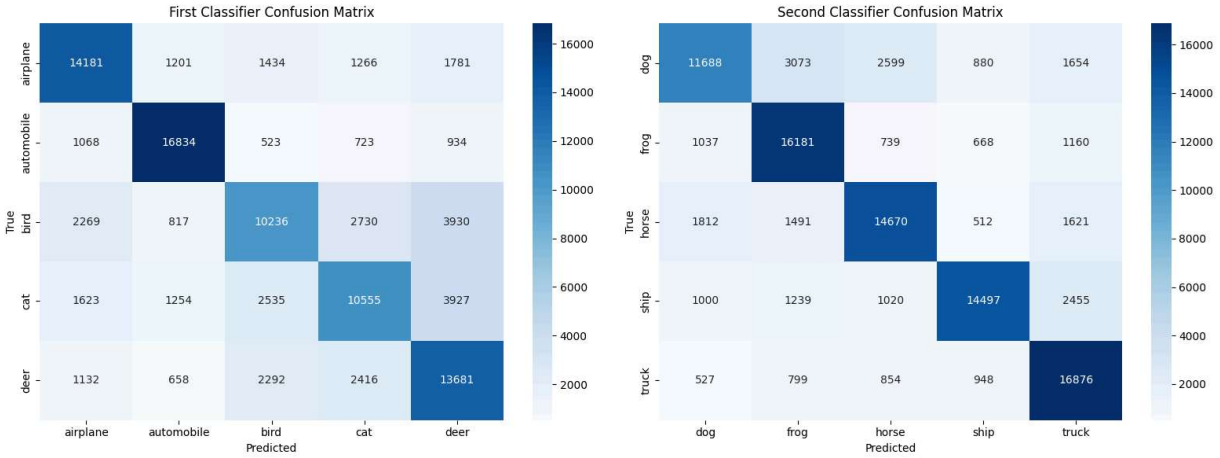
Train or load the best model weights to evaluate it.

In [59]:
```
# history = model.fit(
#     traingen,            # Training data generator
#     epochs=100,          # Number of epochs to train
#     steps_per_epoch=len(cifar10_x_train)//batchsize,
#     validation_data=valgen,  # Validation data generator
#     validation_steps=len(cifar10_x_val)//batchsize,  # Steps per validation epoch
#     callbacks=[early_stopping],  # Early stopping callback
#     verbose=2
# )
model = load_model('69.55acc.keras')
```

Nearly 70% accuracy on a small model like this one is a noteworthy result!

In [60]:
```
evaluate_model(model, testgen)
```

```
100%|██████████| 100/100 [00:59<00:00,  1.68it/s]
First Classifier Accuracy: 0.6549
Second Classifier Accuracy: 0.7391
Mean Accuracy: 0.6970 ± 0.0421
First Classifier F1 Score: 0.6524
Second Classifier F1 Score: 0.7369
Mean F1 Score: 0.6947 ± 0.0422
```



Along the way, we developed many new skills. Some of the problems we faced:

- overfitting from no checkpointing or early stopping
- understanding the paper's notation
- data augmentation in the form of horizontal flipping and pixel translation, which reduced performance
- optimizing two losses from seperate heads
- dropout making the gradient too noisy, stalling training
- model.summary() not working on custom models without running dummy tensors first, thank you 39-upvote-answer
- fit_generator() deprecated in favor of fit() but plenty of documentation on the internet still exists 1), 2
- creating our own data generator class which inherits from Keras Sequences
- why was the generator was not working with fit()? The output of the generator must have a dictionary whose entries match those of the model when compiled, and the last layers must be named as well as the objects, or else if does not work in the Functional API
- with **tqdm** we first imported it directly, then remembered we have the import the module with the same name as the library
- custom model saving and loading requires a decorator

The Fractional Max Pooling claimed an accuracy of 96.53%, which we do not come close to reaching. We also implemented a behemeoth 62 Million parameter network version. We decided to perform more research. We read the following.

In summary, I personally think the good performance in the Fractional Max-Pooling paper is achieved by a combination of using spatially-sparse CNN with fractional max-pooling and small filters (and network in network) which enable building a deep network even when the input image spatial size is small. Hence in regular CNN network, simply replace regular max pooling with fractional max pooling does not necessarily give you a better performance.
Zhongyu Kuang

This makes sense and we move on. Heading back to the best performing model page, we read the second top performing model STRIVING FOR SIMPLICITY: THE ALL CONVOLUTIONAL NET with a reported accuracy of 95.59%. This model uses standard layers, and is therefore easier to implement since we can use the Functional API. This saves time, allowing for more experimentation! The paper does a nice job describing its architectural choices, summarizing their results compared to other models including our first try in this table.

CIFAR-10 classification error

Method	Error (%)
with large data augmentation	
Spatially Sparse CNN [6]	4.47%
Large ALL-CNN (Ours)	4.41%
Fractional Pooling (1 test) [6]	4.50%
Fractional Pooling (100 tests) [6]	**3.47%**

Their model performs very well. We implement our own versions.

In [61]:
```
def ACCN():
    # CIFAR-10
    inputs = keras.Input(shape=(32,32,3))
    # Feature Extractor
    x = layers.Conv2D(96, (3,3), activation='relu')(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(96, (3,3), activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(96, (3,3), activation='relu', strides=2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation='relu', strides=2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (1,1), activation='relu')(x)
    x = layers.BatchNormalization()(x)
    # Classifers
    x = layers.GlobalAveragePooling2D()(x)
    output1 = layers.Dense(5, activation='softmax', name='output1')(x)
    output2 = layers.Dense(5, activation='softmax', name='output2')(x)
    # Model creation
    model = keras.Model(
        inputs,
        outputs={'output1':output1, 'output2':output2}, name='ACCN',
        )
    return model
```

With a lot of experimentation, we reach the state of the art we shared earlier. We also tried to make a final groundbreaking change, where will take in a much larger input image and convolve through a much deeper network!

In [85]:
```
def LACNN():
    # CIFAR-10 Large
    inputs = keras.Input(shape=(126,126,3))
    #inputs = layers.RandomFlip("horizontal")(inputs) Optional Data Augmentation
    #inputs = layers.RandomTranslation(height_factor=5/32, width_factor=5/32, fill_
    #inputs = layers.RandomRotation(factor=0.2)(inputs)

    # Feature Extractor
    x = layers.Conv2D(320, (2,2), activation=LeakyReLU())(inputs)
    x = layers.Conv2D(320, (2,2), activation=LeakyReLU())(x)
    x = layers.Conv2D(320, (2,2), activation=LeakyReLU(), strides=2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(640, (2,2), activation=LeakyReLU())(x)
    x = layers.Conv2D(640, (2,2), activation=LeakyReLU())(x)
    x = layers.Conv2D(640, (2,2), activation=LeakyReLU(), strides=2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(960, (2,2), activation=LeakyReLU())(x)
    x = layers.Conv2D(960, (2,2), activation=LeakyReLU())(x)
    x = layers.Conv2D(960, (2,2), activation=LeakyReLU(), strides=2)(x)
```

```
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(1280, (2,2), activation=LeakyReLU())(x)
    x = layers.Conv2D(1280, (2,2), activation=LeakyReLU())(x)
    x = layers.Conv2D(960, (2,2), activation=LeakyReLU(), strides=2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(1600, (2,2), activation=LeakyReLU())(x)
    x = layers.Conv2D(1600, (2,2), activation=LeakyReLU())(x)
    x = layers.Conv2D(1600, (2,2), activation=LeakyReLU(), strides=2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(1920, (2,2), activation=LeakyReLU())(x)

    # Classifers
    x = layers.Flatten()(x)
    output1 = layers.Dense(5, activation='softmax', name='output1', kernel_regulari
    output2 = layers.Dense(5, activation='softmax', name='output2', kernel_regulari
    # Model creation
    model = keras.Model(
        inputs,
        outputs={'output1':output1, 'output2':output2}, name='LACNN',
        )
    return model
```

In [86]:
```
K.clear_session()
model = LACNN()
model.compile(optimizer='rmsprop',
              loss={'output1': keras.losses.CategoricalCrossentropy(), 'output2': k
              metrics={'output1': 'accuracy', 'output2': 'accuracy'})
model.summary()
```

Model: "LACNN"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 126, 126, 3)	0	-
conv2d (Conv2D)	(None, 125, 125, 320)	4,160	input_layer[0][0]
conv2d_1 (Conv2D)	(None, 124, 124, 320)	409,920	conv2d[0][0]
conv2d_2 (Conv2D)	(None, 62, 62, 320)	409,920	conv2d_1[0][0]
batch_normalization (BatchNormalizatio…	(None, 62, 62, 320)	1,280	conv2d_2[0][0]
conv2d_3 (Conv2D)	(None, 61, 61, 640)	819,840	batch_normalizat…
conv2d_4 (Conv2D)	(None, 60, 60, 640)	1,639,040	conv2d_3[0][0]
conv2d_5 (Conv2D)	(None, 30, 30, 640)	1,639,040	conv2d_4[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 30, 30, 640)	2,560	conv2d_5[0][0]
conv2d_6 (Conv2D)	(None, 29, 29, 960)	2,458,560	batch_normalizat…
conv2d_7 (Conv2D)	(None, 28, 28, 960)	3,687,360	conv2d_6[0][0]
conv2d_8 (Conv2D)	(None, 14, 14, 960)	3,687,360	conv2d_7[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 14, 14, 960)	3,840	conv2d_8[0][0]
conv2d_9 (Conv2D)	(None, 13, 13, 1280)	4,916,480	batch_normalizat…
conv2d_10 (Conv2D)	(None, 12, 12, 1280)	6,554,880	conv2d_9[0][0]
conv2d_11 (Conv2D)	(None, 6, 6, 960)	4,916,160	conv2d_10[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 6, 6, 960)	3,840	conv2d_11[0][0]
conv2d_12 (Conv2D)	(None, 5, 5, 1600)	6,145,600	batch_normalizat…

conv2d_13 (Conv2D)	(None, 4, 4, 1600)	10,241,600	conv2d_12[0][0]
conv2d_14 (Conv2D)	(None, 2, 2, 1600)	10,241,600	conv2d_13[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 2, 2, 1600)	6,400	conv2d_14[0][0]
conv2d_15 (Conv2D)	(None, 1, 1, 1920)	12,289,920	batch_normalizat…
flatten (Flatten)	(None, 1920)	0	conv2d_15[0][0]
output1 (Dense)	(None, 5)	9,605	flatten[0][0]
output2 (Dense)	(None, 5)	9,605	flatten[0][0]

Total params: 70,098,570 (267.40 MB)

Trainable params: 70,089,610 (267.37 MB)

Non-trainable params: 8,960 (35.00 KB)

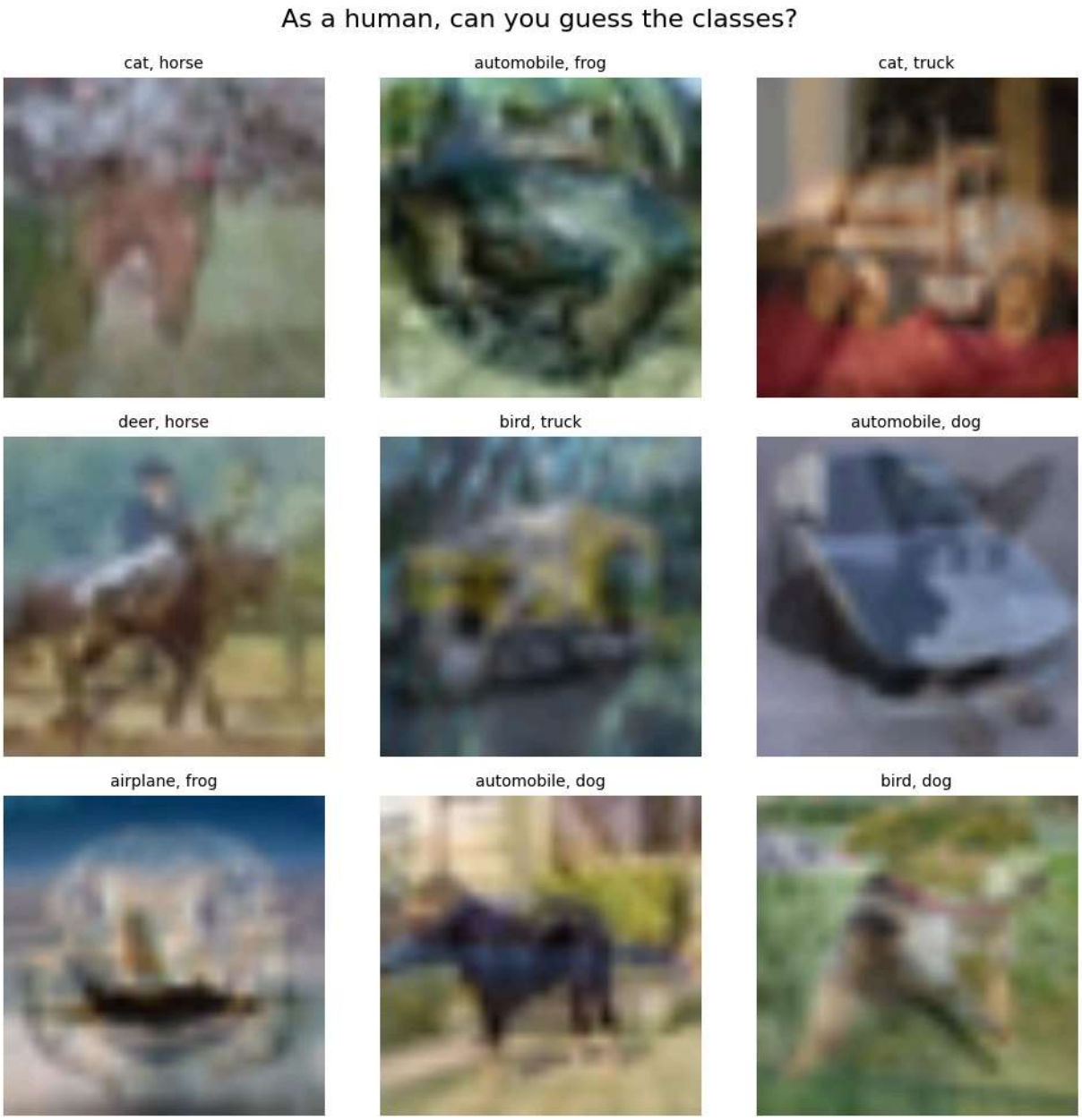
This large model requires images with a much larger resolution of 126x126, so the 32x32 images are centered and padded or scaled. To do so, we implement another generator. Then scaling, rotation, and flipping transformations will take advantage of the extra space! The paper wanted to the images center padded with blank space, but we found scaling to work much better, because many more neurons could be active in the classification task.

In [92]:
```
batchsize = 64

def resize_images(image, target_size=(126, 126)):
    # Resize all images in the batch to the target size with padding
    paddings = [[0, 0], [47, 47], [47, 47], [0, 0]]
    padded_image = tf.pad(image, paddings, mode='CONSTANT', constant_values=0)
    return padded_image

def scale_images(images, target_size=(126, 126)):
    # Scale all images in the batch to the target size
    return tf.image.resize(images, target_size)

def datageneratorlarge(X1, X2, Y1, Y2, batchsize, target_size=(126, 126)):
    size1 = X1.shape[0]
    size2 = X2.shape[0]
    Y1_cat = tf.keras.utils.to_categorical(Y1, num_classes=5)
    Y2_cat = tf.keras.utils.to_categorical(Y2 - 5, num_classes=5)

    while True:
        num1 = np.random.randint(0, size1, batchsize)
        num2 = np.random.randint(0, size2, batchsize)

        # Generate images
        x1_batch = X1[num1]
        x2_batch = X2[num2]
```

```
        # Average image production
        x_data = (X1[num1] + X2[num2]) / 2.0

        # Resize or scale image
        # x_data = resize_images(x_data, target_size)
        x_data = scale_images(x_data, target_size)

        # Create labels
        y_data = {'output1': Y1_cat[num1], 'output2': Y2_cat[num2]}

        # Dictionary for y_data with keys matching the model
        y_data = {'output1': Y1_cat[num1], 'output2': Y2_cat[num2]}

        # Convert numpy to tf tensor
        yield tf.convert_to_tensor(x_data), y_data

# Update data generators
traingenlarge = datageneratorlarge(cifar10_x_train_1, cifar10_x_train_2, cifar10_y_
valgenlarge = datageneratorlarge(cifar10_x_val_1, cifar10_x_val_2, cifar10_y_val_1,
testgenlarge = datageneratorlarge(cifar10_x_test_1, cifar10_x_test_2, cifar10_y_tes
```

In [96]:

```
datagenlarge = datageneratorlarge(cifar10_x_train_1, cifar10_x_train_2, cifar10_y_t
x, y = next(datagenlarge)

fig, axes = plt.subplots(3, 3, figsize=(10, 10))
fig.suptitle("As a human, can you guess the classes?", fontsize=16)

for i, ax in enumerate(axes.flat):
    ax.imshow(x[i])
    ax.axis('off')

    label1 = classes[np.argmax(y['output1'][i])]
    label2 = classes[np.argmax(y['output2'][i]) + 5]
    ax.set_title(f"{label1}, {label2}", fontsize=10)

plt.tight_layout()
plt.subplots_adjust(top=0.92)
plt.show()
```



While this is very exciting, the model at first did not converge during training. Since we implemented the paper's architecture to the letter, this was a disappointing discovery. We believe its depth alongside the blank padding in the input makes the gradient too noisy. Therefore, we removed all the data augmentations and scaled the images to their new size, instead, as mentioned. This still was not enough. To make the model learn better than random baseline, we had to remove all the dropout layers and add some batch normalization layers. A great learning experience!

In [ ]:
```
early_stopping = keras.callbacks.EarlyStopping(monitor='val_loss',  # Track the val
                                patience=20,         # Number of epochs to wait afte
                                mode='min',         # Stop when the value stops decr
                                restore_best_weights=True,  # Restore the best weigh
                                verbose=1)
reduce_lr = keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                               patience=10, min_lr=1e-9, verbose=1)
history = model.fit(
```

```
    traingenlarge,
    epochs=250,
    steps_per_epoch=len(cifar10_x_train)//batchsize, # necessary since generator lo
    validation_data=valgenlarge,
    validation_steps=len(cifar10_x_val)//batchsize,  # necessary since generator lo
    callbacks=[early_stopping, reduce_lr],
    verbose=1
)
```

```
Epoch 1/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 289s 411ms/step - loss: 2.7667 - output1_accuracy: 0.41
42 - output1_loss: 1.3938 - output2_accuracy: 0.4287 - output2_loss: 1.3691 - val_lo
ss: 2.8648 - val_output1_accuracy: 0.3676 - val_output1_loss: 1.4868 - val_output2_a
ccuracy: 0.4189 - val_output2_loss: 1.3729 - learning_rate: 0.0010
Epoch 2/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 289s 410ms/step - loss: 2.6904 - output1_accuracy: 0.42
85 - output1_loss: 1.3587 - output2_accuracy: 0.4468 - output2_loss: 1.3261 - val_lo
ss: 3.2670 - val_output1_accuracy: 0.2967 - val_output1_loss: 1.7768 - val_output2_a
ccuracy: 0.4004 - val_output2_loss: 1.4829 - learning_rate: 0.0010
Epoch 3/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 289s 412ms/step - loss: 2.6246 - output1_accuracy: 0.44
70 - output1_loss: 1.3199 - output2_accuracy: 0.4655 - output2_loss: 1.2968 - val_lo
ss: 2.5721 - val_output1_accuracy: 0.4413 - val_output1_loss: 1.3089 - val_output2_a
ccuracy: 0.4854 - val_output2_loss: 1.2539 - learning_rate: 0.0010
Epoch 4/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 412ms/step - loss: 2.5777 - output1_accuracy: 0.45
52 - output1_loss: 1.2957 - output2_accuracy: 0.4748 - output2_loss: 1.2721 - val_lo
ss: 2.5503 - val_output1_accuracy: 0.4561 - val_output1_loss: 1.2990 - val_output2_a
ccuracy: 0.4982 - val_output2_loss: 1.2402 - learning_rate: 0.0010
Epoch 5/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 412ms/step - loss: 2.5428 - output1_accuracy: 0.46
63 - output1_loss: 1.2818 - output2_accuracy: 0.4847 - output2_loss: 1.2492 - val_lo
ss: 2.5674 - val_output1_accuracy: 0.4483 - val_output1_loss: 1.3081 - val_output2_a
ccuracy: 0.4850 - val_output2_loss: 1.2461 - learning_rate: 0.0010
Epoch 6/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 289s 411ms/step - loss: 2.5119 - output1_accuracy: 0.46
95 - output1_loss: 1.2668 - output2_accuracy: 0.4928 - output2_loss: 1.2314 - val_lo
ss: 2.9274 - val_output1_accuracy: 0.3750 - val_output1_loss: 1.5062 - val_output2_a
ccuracy: 0.4343 - val_output2_loss: 1.4062 - learning_rate: 0.0010
Epoch 7/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 413ms/step - loss: 2.4676 - output1_accuracy: 0.47
35 - output1_loss: 1.2565 - output2_accuracy: 0.5170 - output2_loss: 1.1957 - val_lo
ss: 2.4431 - val_output1_accuracy: 0.4643 - val_output1_loss: 1.2658 - val_output2_a
ccuracy: 0.5329 - val_output2_loss: 1.1610 - learning_rate: 0.0010
Epoch 8/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 294s 418ms/step - loss: 2.4201 - output1_accuracy: 0.48
89 - output1_loss: 1.2352 - output2_accuracy: 0.5302 - output2_loss: 1.1682 - val_lo
ss: 2.4280 - val_output1_accuracy: 0.4796 - val_output1_loss: 1.2309 - val_output2_a
ccuracy: 0.5216 - val_output2_loss: 1.1795 - learning_rate: 0.0010
Epoch 9/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 412ms/step - loss: 2.3996 - output1_accuracy: 0.49
29 - output1_loss: 1.2276 - output2_accuracy: 0.5377 - output2_loss: 1.1544 - val_lo
ss: 2.4747 - val_output1_accuracy: 0.4627 - val_output1_loss: 1.2883 - val_output2_a
ccuracy: 0.5325 - val_output2_loss: 1.1676 - learning_rate: 0.0010
Epoch 10/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 289s 411ms/step - loss: 2.3734 - output1_accuracy: 0.49
93 - output1_loss: 1.2123 - output2_accuracy: 0.5431 - output2_loss: 1.1424 - val_lo
ss: 2.4003 - val_output1_accuracy: 0.4758 - val_output1_loss: 1.2494 - val_output2_a
ccuracy: 0.5483 - val_output2_loss: 1.1313 - learning_rate: 0.0010
Epoch 11/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 413ms/step - loss: 2.3292 - output1_accuracy: 0.50
22 - output1_loss: 1.1960 - output2_accuracy: 0.5554 - output2_loss: 1.1138 - val_lo
ss: 2.3850 - val_output1_accuracy: 0.4724 - val_output1_loss: 1.2407 - val_output2_a
ccuracy: 0.5543 - val_output2_loss: 1.1248 - learning_rate: 0.0010
Epoch 12/250
```

```
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 413ms/step - loss: 2.3010 - output1_accuracy: 0.50
99 - output1_loss: 1.1931 - output2_accuracy: 0.5716 - output2_loss: 1.0883 - val_lo
ss: 2.3571 - val_output1_accuracy: 0.4934 - val_output1_loss: 1.2137 - val_output2_a
ccuracy: 0.5459 - val_output2_loss: 1.1236 - learning_rate: 0.0010
Epoch 13/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 292s 415ms/step - loss: 2.2610 - output1_accuracy: 0.51
81 - output1_loss: 1.1726 - output2_accuracy: 0.5772 - output2_loss: 1.0680 - val_lo
ss: 2.3133 - val_output1_accuracy: 0.4856 - val_output1_loss: 1.2328 - val_output2_a
ccuracy: 0.5905 - val_output2_loss: 1.0601 - learning_rate: 0.0010
Epoch 14/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 413ms/step - loss: 2.2247 - output1_accuracy: 0.52
08 - output1_loss: 1.1689 - output2_accuracy: 0.5923 - output2_loss: 1.0350 - val_lo
ss: 2.1809 - val_output1_accuracy: 0.5162 - val_output1_loss: 1.1822 - val_output2_a
ccuracy: 0.6226 - val_output2_loss: 0.9777 - learning_rate: 0.0010
Epoch 15/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 293s 417ms/step - loss: 2.1616 - output1_accuracy: 0.53
29 - output1_loss: 1.1417 - output2_accuracy: 0.6103 - output2_loss: 0.9983 - val_lo
ss: 2.1937 - val_output1_accuracy: 0.5180 - val_output1_loss: 1.1654 - val_output2_a
ccuracy: 0.5992 - val_output2_loss: 1.0066 - learning_rate: 0.0010
Epoch 16/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 289s 411ms/step - loss: 2.1193 - output1_accuracy: 0.54
23 - output1_loss: 1.1208 - output2_accuracy: 0.6208 - output2_loss: 0.9766 - val_lo
ss: 2.2747 - val_output1_accuracy: 0.5086 - val_output1_loss: 1.1952 - val_output2_a
ccuracy: 0.5950 - val_output2_loss: 1.0575 - learning_rate: 0.0010
Epoch 17/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 413ms/step - loss: 2.0778 - output1_accuracy: 0.55
12 - output1_loss: 1.1059 - output2_accuracy: 0.6347 - output2_loss: 0.9501 - val_lo
ss: 2.1203 - val_output1_accuracy: 0.5230 - val_output1_loss: 1.1691 - val_output2_a
ccuracy: 0.6516 - val_output2_loss: 0.9293 - learning_rate: 0.0010
Epoch 18/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 292s 415ms/step - loss: 2.0171 - output1_accuracy: 0.56
00 - output1_loss: 1.0904 - output2_accuracy: 0.6547 - output2_loss: 0.9046 - val_lo
ss: 2.1011 - val_output1_accuracy: 0.5353 - val_output1_loss: 1.1268 - val_output2_a
ccuracy: 0.6328 - val_output2_loss: 0.9523 - learning_rate: 0.0010
Epoch 19/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 292s 415ms/step - loss: 1.9627 - output1_accuracy: 0.56
98 - output1_loss: 1.0665 - output2_accuracy: 0.6652 - output2_loss: 0.8738 - val_lo
ss: 2.0468 - val_output1_accuracy: 0.5457 - val_output1_loss: 1.1163 - val_output2_a
ccuracy: 0.6556 - val_output2_loss: 0.9081 - learning_rate: 0.0010
Epoch 20/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 292s 415ms/step - loss: 1.9263 - output1_accuracy: 0.57
33 - output1_loss: 1.0512 - output2_accuracy: 0.6753 - output2_loss: 0.8525 - val_lo
ss: 2.0436 - val_output1_accuracy: 0.5337 - val_output1_loss: 1.1409 - val_output2_a
ccuracy: 0.6673 - val_output2_loss: 0.8796 - learning_rate: 0.0010
Epoch 21/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 413ms/step - loss: 1.8810 - output1_accuracy: 0.59
37 - output1_loss: 1.0223 - output2_accuracy: 0.6829 - output2_loss: 0.8355 - val_lo
ss: 1.9177 - val_output1_accuracy: 0.5647 - val_output1_loss: 1.0767 - val_output2_a
ccuracy: 0.6903 - val_output2_loss: 0.8171 - learning_rate: 0.0010
Epoch 22/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 414ms/step - loss: 1.8537 - output1_accuracy: 0.58
91 - output1_loss: 1.0272 - output2_accuracy: 0.6954 - output2_loss: 0.8031 - val_lo
ss: 1.9614 - val_output1_accuracy: 0.5475 - val_output1_loss: 1.1009 - val_output2_a
ccuracy: 0.6797 - val_output2_loss: 0.8371 - learning_rate: 0.0010
Epoch 23/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 412ms/step - loss: 1.8310 - output1_accuracy: 0.59
```

```
49 - output1_loss: 1.0195 - output2_accuracy: 0.7031 - output2_loss: 0.7880 - val_lo
ss: 1.8931 - val_output1_accuracy: 0.5717 - val_output1_loss: 1.0552 - val_output2_a
ccuracy: 0.6897 - val_output2_loss: 0.8143 - learning_rate: 0.0010
Epoch 24/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 414ms/step - loss: 1.7744 - output1_accuracy: 0.61
21 - output1_loss: 0.9798 - output2_accuracy: 0.7056 - output2_loss: 0.7707 - val_lo
ss: 1.8789 - val_output1_accuracy: 0.5687 - val_output1_loss: 1.0711 - val_output2_a
ccuracy: 0.7045 - val_output2_loss: 0.7842 - learning_rate: 0.0010
Epoch 25/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 414ms/step - loss: 1.7581 - output1_accuracy: 0.60
98 - output1_loss: 0.9768 - output2_accuracy: 0.7151 - output2_loss: 0.7574 - val_lo
ss: 1.9366 - val_output1_accuracy: 0.5613 - val_output1_loss: 1.0861 - val_output2_a
ccuracy: 0.6931 - val_output2_loss: 0.8267 - learning_rate: 0.0010
Epoch 26/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 289s 412ms/step - loss: 1.7297 - output1_accuracy: 0.61
46 - output1_loss: 0.9612 - output2_accuracy: 0.7232 - output2_loss: 0.7444 - val_lo
ss: 1.8400 - val_output1_accuracy: 0.5837 - val_output1_loss: 1.0449 - val_output2_a
ccuracy: 0.7035 - val_output2_loss: 0.7709 - learning_rate: 0.0010
Epoch 27/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 289s 411ms/step - loss: 1.7042 - output1_accuracy: 0.62
42 - output1_loss: 0.9513 - output2_accuracy: 0.7255 - output2_loss: 0.7286 - val_lo
ss: 1.8722 - val_output1_accuracy: 0.5845 - val_output1_loss: 1.0499 - val_output2_a
ccuracy: 0.7011 - val_output2_loss: 0.7976 - learning_rate: 0.0010
Epoch 28/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 413ms/step - loss: 1.6715 - output1_accuracy: 0.63
33 - output1_loss: 0.9321 - output2_accuracy: 0.7311 - output2_loss: 0.7148 - val_lo
ss: 1.9135 - val_output1_accuracy: 0.5881 - val_output1_loss: 1.0395 - val_output2_a
ccuracy: 0.6833 - val_output2_loss: 0.8498 - learning_rate: 0.0010
Epoch 29/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 412ms/step - loss: 1.6584 - output1_accuracy: 0.63
42 - output1_loss: 0.9272 - output2_accuracy: 0.7354 - output2_loss: 0.7066 - val_lo
ss: 1.7723 - val_output1_accuracy: 0.6174 - val_output1_loss: 0.9845 - val_output2_a
ccuracy: 0.7071 - val_output2_loss: 0.7629 - learning_rate: 0.0010
Epoch 30/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 412ms/step - loss: 1.6403 - output1_accuracy: 0.63
76 - output1_loss: 0.9202 - output2_accuracy: 0.7378 - output2_loss: 0.6954 - val_lo
ss: 1.7458 - val_output1_accuracy: 0.6094 - val_output1_loss: 1.0003 - val_output2_a
ccuracy: 0.7330 - val_output2_loss: 0.7210 - learning_rate: 0.0010
Epoch 31/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 293s 417ms/step - loss: 1.6251 - output1_accuracy: 0.64
14 - output1_loss: 0.9092 - output2_accuracy: 0.7393 - output2_loss: 0.6912 - val_lo
ss: 1.8836 - val_output1_accuracy: 0.5982 - val_output1_loss: 1.0425 - val_output2_a
ccuracy: 0.6995 - val_output2_loss: 0.8159 - learning_rate: 0.0010
Epoch 32/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 414ms/step - loss: 1.6142 - output1_accuracy: 0.64
64 - output1_loss: 0.8996 - output2_accuracy: 0.7429 - output2_loss: 0.6898 - val_lo
ss: 1.7370 - val_output1_accuracy: 0.6158 - val_output1_loss: 0.9674 - val_output2_a
ccuracy: 0.7135 - val_output2_loss: 0.7448 - learning_rate: 0.0010
Epoch 33/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 413ms/step - loss: 1.5778 - output1_accuracy: 0.65
52 - output1_loss: 0.8817 - output2_accuracy: 0.7486 - output2_loss: 0.6714 - val_lo
ss: 1.7664 - val_output1_accuracy: 0.6022 - val_output1_loss: 1.0018 - val_output2_a
ccuracy: 0.7194 - val_output2_loss: 0.7398 - learning_rate: 0.0010
Epoch 34/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 289s 411ms/step - loss: 1.5647 - output1_accuracy: 0.65
68 - output1_loss: 0.8749 - output2_accuracy: 0.7533 - output2_loss: 0.6648 - val_lo
```

```
ss: 1.7173 - val_output1_accuracy: 0.6074 - val_output1_loss: 0.9864 - val_output2_a
ccuracy: 0.7370 - val_output2_loss: 0.7057 - learning_rate: 0.0010
Epoch 35/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 414ms/step - loss: 1.5501 - output1_accuracy: 0.66
11 - output1_loss: 0.8638 - output2_accuracy: 0.7511 - output2_loss: 0.6613 - val_lo
ss: 1.7512 - val_output1_accuracy: 0.6064 - val_output1_loss: 0.9887 - val_output2_a
ccuracy: 0.7270 - val_output2_loss: 0.7376 - learning_rate: 0.0010
Epoch 36/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 413ms/step - loss: 1.5367 - output1_accuracy: 0.66
22 - output1_loss: 0.8690 - output2_accuracy: 0.7565 - output2_loss: 0.6428 - val_lo
ss: 1.7320 - val_output1_accuracy: 0.6136 - val_output1_loss: 0.9875 - val_output2_a
ccuracy: 0.7352 - val_output2_loss: 0.7190 - learning_rate: 0.0010
Epoch 37/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 412ms/step - loss: 1.5176 - output1_accuracy: 0.66
84 - output1_loss: 0.8546 - output2_accuracy: 0.7626 - output2_loss: 0.6378 - val_lo
ss: 1.6982 - val_output1_accuracy: 0.6330 - val_output1_loss: 0.9460 - val_output2_a
ccuracy: 0.7246 - val_output2_loss: 0.7262 - learning_rate: 0.0010
Epoch 38/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 414ms/step - loss: 1.5065 - output1_accuracy: 0.67
21 - output1_loss: 0.8452 - output2_accuracy: 0.7607 - output2_loss: 0.6358 - val_lo
ss: 1.7159 - val_output1_accuracy: 0.6070 - val_output1_loss: 0.9862 - val_output2_a
ccuracy: 0.7346 - val_output2_loss: 0.7041 - learning_rate: 0.0010
Epoch 39/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 413ms/step - loss: 1.4788 - output1_accuracy: 0.67
62 - output1_loss: 0.8316 - output2_accuracy: 0.7675 - output2_loss: 0.6217 - val_lo
ss: 1.7471 - val_output1_accuracy: 0.6068 - val_output1_loss: 1.0164 - val_output2_a
ccuracy: 0.7344 - val_output2_loss: 0.7048 - learning_rate: 0.0010
Epoch 40/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 294s 418ms/step - loss: 1.4795 - output1_accuracy: 0.67
56 - output1_loss: 0.8263 - output2_accuracy: 0.7675 - output2_loss: 0.6278 - val_lo
ss: 1.6945 - val_output1_accuracy: 0.6176 - val_output1_loss: 0.9757 - val_output2_a
ccuracy: 0.7450 - val_output2_loss: 0.6935 - learning_rate: 0.0010
Epoch 41/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 292s 415ms/step - loss: 1.4526 - output1_accuracy: 0.68
16 - output1_loss: 0.8174 - output2_accuracy: 0.7727 - output2_loss: 0.6100 - val_lo
ss: 1.9339 - val_output1_accuracy: 0.6112 - val_output1_loss: 1.0041 - val_output2_a
ccuracy: 0.6831 - val_output2_loss: 0.9047 - learning_rate: 0.0010
Epoch 42/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 413ms/step - loss: 1.4269 - output1_accuracy: 0.68
33 - output1_loss: 0.8068 - output2_accuracy: 0.7773 - output2_loss: 0.5950 - val_lo
ss: 1.7256 - val_output1_accuracy: 0.6178 - val_output1_loss: 0.9843 - val_output2_a
ccuracy: 0.7364 - val_output2_loss: 0.7152 - learning_rate: 0.0010
Epoch 43/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 294s 418ms/step - loss: 1.4198 - output1_accuracy: 0.68
46 - output1_loss: 0.8064 - output2_accuracy: 0.7814 - output2_loss: 0.5879 - val_lo
ss: 1.6134 - val_output1_accuracy: 0.6408 - val_output1_loss: 0.9256 - val_output2_a
ccuracy: 0.7508 - val_output2_loss: 0.6625 - learning_rate: 0.0010
Epoch 44/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 291s 413ms/step - loss: 1.4244 - output1_accuracy: 0.68
62 - output1_loss: 0.8082 - output2_accuracy: 0.7765 - output2_loss: 0.5909 - val_lo
ss: 1.6452 - val_output1_accuracy: 0.6374 - val_output1_loss: 0.9501 - val_output2_a
ccuracy: 0.7446 - val_output2_loss: 0.6698 - learning_rate: 0.0010
Epoch 45/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 297s 422ms/step - loss: 1.3827 - output1_accuracy: 0.69
08 - output1_loss: 0.7903 - output2_accuracy: 0.7903 - output2_loss: 0.5670 - val_lo
ss: 1.6101 - val_output1_accuracy: 0.6468 - val_output1_loss: 0.9103 - val_output2_a
```

```
ccuracy: 0.7532 - val_output2_loss: 0.6748 - learning_rate: 0.0010
Epoch 46/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 290s 412ms/step - loss: 1.3935 - output1_accuracy: 0.69
19 - output1_loss: 0.7866 - output2_accuracy: 0.7853 - output2_loss: 0.5816 - val_lo
ss: 1.6672 - val_output1_accuracy: 0.6174 - val_output1_loss: 0.9929 - val_output2_a
ccuracy: 0.7550 - val_output2_loss: 0.6486 - learning_rate: 0.0010
Epoch 47/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 293s 417ms/step - loss: 1.3667 - output1_accuracy: 0.69
91 - output1_loss: 0.7733 - output2_accuracy: 0.7905 - output2_loss: 0.5678 - val_lo
ss: 1.6190 - val_output1_accuracy: 0.6312 - val_output1_loss: 0.9451 - val_output2_a
ccuracy: 0.7714 - val_output2_loss: 0.6484 - learning_rate: 0.0010
Epoch 48/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 292s 416ms/step - loss: 1.3664 - output1_accuracy: 0.69
68 - output1_loss: 0.7814 - output2_accuracy: 0.7914 - output2_loss: 0.5597 - val_lo
ss: 1.7229 - val_output1_accuracy: 0.6308 - val_output1_loss: 0.9416 - val_output2_a
ccuracy: 0.7240 - val_output2_loss: 0.7556 - learning_rate: 0.0010
Epoch 49/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 298s 423ms/step - loss: 1.3325 - output1_accuracy: 0.70
85 - output1_loss: 0.7606 - output2_accuracy: 0.7991 - output2_loss: 0.5463 - val_lo
ss: 1.5573 - val_output1_accuracy: 0.6370 - val_output1_loss: 0.9237 - val_output2_a
ccuracy: 0.7690 - val_output2_loss: 0.6079 - learning_rate: 0.0010
Epoch 50/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 292s 415ms/step - loss: 1.3197 - output1_accuracy: 0.71
11 - output1_loss: 0.7567 - output2_accuracy: 0.8008 - output2_loss: 0.5373 - val_lo
ss: 1.6105 - val_output1_accuracy: 0.6434 - val_output1_loss: 0.9087 - val_output2_a
ccuracy: 0.7548 - val_output2_loss: 0.6763 - learning_rate: 0.0010
Epoch 51/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 293s 416ms/step - loss: 1.3207 - output1_accuracy: 0.71
00 - output1_loss: 0.7522 - output2_accuracy: 0.8015 - output2_loss: 0.5429 - val_lo
ss: 1.6243 - val_output1_accuracy: 0.6462 - val_output1_loss: 0.9132 - val_output2_a
ccuracy: 0.7588 - val_output2_loss: 0.6851 - learning_rate: 0.0010
Epoch 52/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 292s 416ms/step - loss: 1.3047 - output1_accuracy: 0.71
15 - output1_loss: 0.7452 - output2_accuracy: 0.8020 - output2_loss: 0.5335 - val_lo
ss: 1.5607 - val_output1_accuracy: 0.6444 - val_output1_loss: 0.9076 - val_output2_a
ccuracy: 0.7728 - val_output2_loss: 0.6276 - learning_rate: 0.0010
Epoch 53/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 295s 419ms/step - loss: 1.2890 - output1_accuracy: 0.71
22 - output1_loss: 0.7338 - output2_accuracy: 0.8045 - output2_loss: 0.5298 - val_lo
ss: 1.6934 - val_output1_accuracy: 0.6352 - val_output1_loss: 0.9751 - val_output2_a
ccuracy: 0.7480 - val_output2_loss: 0.6930 - learning_rate: 0.0010
Epoch 54/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 297s 423ms/step - loss: 1.2966 - output1_accuracy: 0.71
04 - output1_loss: 0.7467 - output2_accuracy: 0.8055 - output2_loss: 0.5244 - val_lo
ss: 1.5699 - val_output1_accuracy: 0.6599 - val_output1_loss: 0.8825 - val_output2_a
ccuracy: 0.7610 - val_output2_loss: 0.6614 - learning_rate: 0.0010
Epoch 55/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 296s 421ms/step - loss: 1.2652 - output1_accuracy: 0.72
28 - output1_loss: 0.7180 - output2_accuracy: 0.8102 - output2_loss: 0.5212 - val_lo
ss: 1.6780 - val_output1_accuracy: 0.6310 - val_output1_loss: 0.9509 - val_output2_a
ccuracy: 0.7404 - val_output2_loss: 0.7014 - learning_rate: 0.0010
Epoch 56/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 295s 420ms/step - loss: 1.2587 - output1_accuracy: 0.72
36 - output1_loss: 0.7202 - output2_accuracy: 0.8139 - output2_loss: 0.5128 - val_lo
ss: 1.6386 - val_output1_accuracy: 0.6252 - val_output1_loss: 0.9568 - val_output2_a
ccuracy: 0.7642 - val_output2_loss: 0.6563 - learning_rate: 0.0010
```

```
Epoch 57/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 300s 426ms/step - loss: 1.2331 - output1_accuracy: 0.73
06 - output1_loss: 0.7032 - output2_accuracy: 0.8123 - output2_loss: 0.5040 - val_lo
ss: 1.5872 - val_output1_accuracy: 0.6392 - val_output1_loss: 0.9353 - val_output2_a
ccuracy: 0.7682 - val_output2_loss: 0.6259 - learning_rate: 0.0010
Epoch 58/250
703/703 ━━━━━━━━━━━━━━━━━━━━ 0s 398ms/step - loss: 1.2267 - output1_accuracy: 0.7300
- output1_loss: 0.7011 - output2_accuracy: 0.8149 - output2_loss: 0.4998
```

The model did not achieve groundbreaking results with an average validation accuracy of 70.37%.

Conclusion

We experimented with different optimizers, starting learning rates, activation functions, data augmentations, regularizers, label smoothings, dropouts, batch norm, convolutional layers specific to each classifier, global average pooling, and softmax over 1x1x5 convolutions directly. To do so, we adopted good scientific procedure and either changed one configuration at a time or performed ablation to see which were helpful. For example, in the appendix we test the result of changing the activation function! The ease of testing was a significant milestone in our Machine Learning capabilities.

We learned that it is very hard to achieve the same results as are published in research papers. In the future, we would mitigate this by cross checking the papers which are cited to see if others were able to reproduce them.

Appendix

In [7]:
```python
def ACCNL():
    # CIFAR-10
    inputs = keras.Input(shape=(32,32,3))
    # Feature Extractor
    x = layers.Conv2D(96, (3,3), activation=LeakyReLU())(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(96, (3,3), activation=LeakyReLU())(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(96, (3,3), activation=LeakyReLU(), strides=2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation=LeakyReLU())(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation=LeakyReLU())(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation=LeakyReLU(), strides=2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (3,3), activation=LeakyReLU())(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(192, (1,1), activation=LeakyReLU())(x)
    x = layers.BatchNormalization()(x)
    # Classifers
    x = layers.GlobalAveragePooling2D()(x)
```

```
    output1 = layers.Dense(5, activation='softmax', name='output1')(x)
    output2 = layers.Dense(5, activation='softmax', name='output2')(x)
    # Model creation
    model = keras.Model(
        inputs,
        outputs={'output1':output1, 'output2':output2}, name='ACCNL',
        )
    return model
```

In [9]:
```
K.clear_session()
model = ACCNL()
model.compile(optimizer='rmsprop',
              loss={'output1': keras.losses.CategoricalCrossentropy(label_smoothing
              metrics={'output1': 'accuracy', 'output2': 'accuracy'})


model.summary()
```

Model: "ACCNL"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 32, 32, 3)	0	-
conv2d (Conv2D)	(None, 30, 30, 96)	2,688	input_layer[0][0]
batch_normalization (BatchNormalizatio…	(None, 30, 30, 96)	384	conv2d[0][0]
conv2d_1 (Conv2D)	(None, 28, 28, 96)	83,040	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 28, 28, 96)	384	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 13, 13, 96)	83,040	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 13, 13, 96)	384	conv2d_2[0][0]
conv2d_3 (Conv2D)	(None, 11, 11, 192)	166,080	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 11, 11, 192)	768	conv2d_3[0][0]
conv2d_4 (Conv2D)	(None, 9, 9, 192)	331,968	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 9, 9, 192)	768	conv2d_4[0][0]
conv2d_5 (Conv2D)	(None, 4, 4, 192)	331,968	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 4, 4, 192)	768	conv2d_5[0][0]
conv2d_6 (Conv2D)	(None, 2, 2, 192)	331,968	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 2, 2, 192)	768	conv2d_6[0][0]
conv2d_7 (Conv2D)	(None, 2, 2, 192)	37,056	batch_normalizat…
batch_normalizatio… (BatchNormalizatio…	(None, 2, 2, 192)	768	conv2d_7[0][0]
global_average_poo… (GlobalAveragePool…	(None, 192)	0	batch_normalizat…
output1 (Dense)	(None, 5)	965	global_average_p…

output2 (Dense)	(None, 5)	965	global_average_p…

Total params: 1,374,730 (5.24 MB)

Trainable params: 1,372,234 (5.23 MB)

Non-trainable params: 2,496 (9.75 KB)

In [13]:
```
early_stopping = keras.callbacks.EarlyStopping(monitor='val_loss',
                                patience=20,
                                mode='min',
                                restore_best_weights=True,
                                verbose=1)
reduce_lr = keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                               patience=5, min_lr=1e-9, verbose=1)
history = model.fit(
    traingen,            # Training data generator
    epochs=100,          # Number of epochs to train
    steps_per_epoch=len(cifar10_x_train)//batchsize,
    validation_data=valgen,  # Validation data generator
    validation_steps=len(cifar10_x_val)//batchsize,  # Steps per validation epoch (
    callbacks=[early_stopping,reduce_lr],  # Early stopping callback
    verbose=2
)
```

```
Epoch 1/100
703/703 - 9s - 13ms/step - loss: 2.7728 - output1_accuracy: 0.4325 - output1_loss:
1.4111 - output2_accuracy: 0.4694 - output2_loss: 1.3616 - val_loss: 2.6788 - val_ou
tput1_accuracy: 0.4545 - val_output1_loss: 1.3733 - val_output2_accuracy: 0.5018 - v
al_output2_loss: 1.3055 - learning_rate: 1.0000e-03
Epoch 2/100
703/703 - 7s - 10ms/step - loss: 2.4531 - output1_accuracy: 0.5029 - output1_loss:
1.2989 - output2_accuracy: 0.6022 - output2_loss: 1.1541 - val_loss: 2.6978 - val_ou
tput1_accuracy: 0.4647 - val_output1_loss: 1.5315 - val_output2_accuracy: 0.6068 - v
al_output2_loss: 1.1663 - learning_rate: 1.0000e-03
Epoch 3/100
703/703 - 7s - 9ms/step - loss: 2.3225 - output1_accuracy: 0.5398 - output1_loss: 1.
2418 - output2_accuracy: 0.6426 - output2_loss: 1.0807 - val_loss: 2.5916 - val_outp
ut1_accuracy: 0.4653 - val_output1_loss: 1.3899 - val_output2_accuracy: 0.5819 - val
_output2_loss: 1.2017 - learning_rate: 1.0000e-03
Epoch 4/100
703/703 - 7s - 9ms/step - loss: 2.2450 - output1_accuracy: 0.5617 - output1_loss: 1.
2035 - output2_accuracy: 0.6646 - output2_loss: 1.0415 - val_loss: 2.3295 - val_outp
ut1_accuracy: 0.5347 - val_output1_loss: 1.2657 - val_output2_accuracy: 0.6663 - val
_output2_loss: 1.0639 - learning_rate: 1.0000e-03
Epoch 5/100
703/703 - 7s - 9ms/step - loss: 2.1907 - output1_accuracy: 0.5774 - output1_loss: 1.
1808 - output2_accuracy: 0.6846 - output2_loss: 1.0099 - val_loss: 2.3734 - val_outp
ut1_accuracy: 0.5323 - val_output1_loss: 1.2791 - val_output2_accuracy: 0.6394 - val
_output2_loss: 1.0943 - learning_rate: 1.0000e-03
Epoch 6/100
703/703 - 7s - 10ms/step - loss: 2.1399 - output1_accuracy: 0.5938 - output1_loss:
1.1535 - output2_accuracy: 0.6988 - output2_loss: 0.9863 - val_loss: 2.2482 - val_ou
tput1_accuracy: 0.5817 - val_output1_loss: 1.1717 - val_output2_accuracy: 0.6548 - v
al_output2_loss: 1.0765 - learning_rate: 1.0000e-03
Epoch 7/100
703/703 - 7s - 9ms/step - loss: 2.0946 - output1_accuracy: 0.6060 - output1_loss: 1.
1299 - output2_accuracy: 0.7093 - output2_loss: 0.9647 - val_loss: 2.3367 - val_outp
ut1_accuracy: 0.5465 - val_output1_loss: 1.2475 - val_output2_accuracy: 0.6472 - val
_output2_loss: 1.0893 - learning_rate: 1.0000e-03
Epoch 8/100
703/703 - 7s - 9ms/step - loss: 2.0588 - output1_accuracy: 0.6219 - output1_loss: 1.
1105 - output2_accuracy: 0.7183 - output2_loss: 0.9483 - val_loss: 2.2221 - val_outp
ut1_accuracy: 0.5653 - val_output1_loss: 1.2254 - val_output2_accuracy: 0.7115 - val
_output2_loss: 0.9967 - learning_rate: 1.0000e-03
Epoch 9/100
703/703 - 7s - 10ms/step - loss: 2.0351 - output1_accuracy: 0.6279 - output1_loss:
1.1000 - output2_accuracy: 0.7270 - output2_loss: 0.9351 - val_loss: 2.7488 - val_ou
tput1_accuracy: 0.4046 - val_output1_loss: 1.5610 - val_output2_accuracy: 0.5952 - v
al_output2_loss: 1.1878 - learning_rate: 1.0000e-03
Epoch 10/100
703/703 - 7s - 10ms/step - loss: 1.9968 - output1_accuracy: 0.6375 - output1_loss:
1.0792 - output2_accuracy: 0.7359 - output2_loss: 0.9177 - val_loss: 2.1220 - val_ou
tput1_accuracy: 0.6018 - val_output1_loss: 1.1492 - val_output2_accuracy: 0.7127 - v
al_output2_loss: 0.9728 - learning_rate: 1.0000e-03
Epoch 11/100
703/703 - 7s - 10ms/step - loss: 1.9782 - output1_accuracy: 0.6454 - output1_loss:
1.0687 - output2_accuracy: 0.7414 - output2_loss: 0.9096 - val_loss: 2.1148 - val_ou
tput1_accuracy: 0.5899 - val_output1_loss: 1.1648 - val_output2_accuracy: 0.7216 - v
al_output2_loss: 0.9501 - learning_rate: 1.0000e-03
Epoch 12/100
```

```
703/703 - 6s - 9ms/step - loss: 1.9516 - output1_accuracy: 0.6539 - output1_loss: 1.
0543 - output2_accuracy: 0.7471 - output2_loss: 0.8973 - val_loss: 2.2559 - val_outp
ut1_accuracy: 0.5627 - val_output1_loss: 1.2334 - val_output2_accuracy: 0.6925 - val
_output2_loss: 1.0225 - learning_rate: 1.0000e-03
Epoch 13/100
703/703 - 7s - 9ms/step - loss: 1.9404 - output1_accuracy: 0.6570 - output1_loss: 1.
0489 - output2_accuracy: 0.7530 - output2_loss: 0.8915 - val_loss: 2.0160 - val_outp
ut1_accuracy: 0.6290 - val_output1_loss: 1.0880 - val_output2_accuracy: 0.7396 - val
_output2_loss: 0.9280 - learning_rate: 1.0000e-03
Epoch 14/100
703/703 - 7s - 10ms/step - loss: 1.9120 - output1_accuracy: 0.6680 - output1_loss:
1.0305 - output2_accuracy: 0.7580 - output2_loss: 0.8815 - val_loss: 2.2882 - val_ou
tput1_accuracy: 0.5449 - val_output1_loss: 1.2874 - val_output2_accuracy: 0.6953 - v
al_output2_loss: 1.0008 - learning_rate: 1.0000e-03
Epoch 15/100
703/703 - 7s - 9ms/step - loss: 1.8901 - output1_accuracy: 0.6710 - output1_loss: 1.
0229 - output2_accuracy: 0.7668 - output2_loss: 0.8672 - val_loss: 2.0696 - val_outp
ut1_accuracy: 0.6198 - val_output1_loss: 1.1390 - val_output2_accuracy: 0.7272 - val
_output2_loss: 0.9306 - learning_rate: 1.0000e-03
Epoch 16/100
703/703 - 6s - 9ms/step - loss: 1.8718 - output1_accuracy: 0.6798 - output1_loss: 1.
0151 - output2_accuracy: 0.7706 - output2_loss: 0.8567 - val_loss: 2.1047 - val_outp
ut1_accuracy: 0.5984 - val_output1_loss: 1.1508 - val_output2_accuracy: 0.7270 - val
_output2_loss: 0.9539 - learning_rate: 1.0000e-03
Epoch 17/100
703/703 - 7s - 9ms/step - loss: 1.8550 - output1_accuracy: 0.6856 - output1_loss: 1.
0032 - output2_accuracy: 0.7744 - output2_loss: 0.8517 - val_loss: 2.0999 - val_outp
ut1_accuracy: 0.5990 - val_output1_loss: 1.1675 - val_output2_accuracy: 0.7354 - val
_output2_loss: 0.9323 - learning_rate: 1.0000e-03
Epoch 18/100

Epoch 18: ReduceLROnPlateau reducing learning rate to 0.00020000000949949026.
703/703 - 7s - 10ms/step - loss: 1.8379 - output1_accuracy: 0.6910 - output1_loss:
0.9943 - output2_accuracy: 0.7766 - output2_loss: 0.8436 - val_loss: 2.1028 - val_ou
tput1_accuracy: 0.6210 - val_output1_loss: 1.1209 - val_output2_accuracy: 0.7091 - v
al_output2_loss: 0.9819 - learning_rate: 1.0000e-03
Epoch 19/100
703/703 - 7s - 10ms/step - loss: 1.7621 - output1_accuracy: 0.7115 - output1_loss:
0.9556 - output2_accuracy: 0.7995 - output2_loss: 0.8065 - val_loss: 1.8490 - val_ou
tput1_accuracy: 0.6749 - val_output1_loss: 1.0154 - val_output2_accuracy: 0.7877 - v
al_output2_loss: 0.8336 - learning_rate: 2.0000e-04
Epoch 20/100
703/703 - 7s - 10ms/step - loss: 1.7160 - output1_accuracy: 0.7237 - output1_loss:
0.9333 - output2_accuracy: 0.8112 - output2_loss: 0.7827 - val_loss: 1.8688 - val_ou
tput1_accuracy: 0.6769 - val_output1_loss: 1.0161 - val_output2_accuracy: 0.7742 - v
al_output2_loss: 0.8527 - learning_rate: 2.0000e-04
Epoch 21/100
703/703 - 7s - 10ms/step - loss: 1.7075 - output1_accuracy: 0.7280 - output1_loss:
0.9265 - output2_accuracy: 0.8134 - output2_loss: 0.7810 - val_loss: 1.8640 - val_ou
tput1_accuracy: 0.6743 - val_output1_loss: 1.0220 - val_output2_accuracy: 0.7843 - v
al_output2_loss: 0.8420 - learning_rate: 2.0000e-04
Epoch 22/100
703/703 - 7s - 10ms/step - loss: 1.6948 - output1_accuracy: 0.7322 - output1_loss:
0.9202 - output2_accuracy: 0.8168 - output2_loss: 0.7746 - val_loss: 1.8537 - val_ou
tput1_accuracy: 0.6747 - val_output1_loss: 1.0260 - val_output2_accuracy: 0.7881 - v
al_output2_loss: 0.8277 - learning_rate: 2.0000e-04
```

```
Epoch 23/100
703/703 - 7s - 9ms/step - loss: 1.6929 - output1_accuracy: 0.7307 - output1_loss: 0.
9211 - output2_accuracy: 0.8183 - output2_loss: 0.7718 - val_loss: 1.8341 - val_outp
ut1_accuracy: 0.6781 - val_output1_loss: 1.0214 - val_output2_accuracy: 0.7913 - val
_output2_loss: 0.8127 - learning_rate: 2.0000e-04
Epoch 24/100
703/703 - 7s - 10ms/step - loss: 1.6812 - output1_accuracy: 0.7356 - output1_loss:
0.9147 - output2_accuracy: 0.8212 - output2_loss: 0.7664 - val_loss: 1.8628 - val_ou
tput1_accuracy: 0.6647 - val_output1_loss: 1.0358 - val_output2_accuracy: 0.7877 - v
al_output2_loss: 0.8270 - learning_rate: 2.0000e-04
Epoch 25/100
703/703 - 7s - 9ms/step - loss: 1.6674 - output1_accuracy: 0.7384 - output1_loss: 0.
9065 - output2_accuracy: 0.8236 - output2_loss: 0.7609 - val_loss: 1.8874 - val_outp
ut1_accuracy: 0.6530 - val_output1_loss: 1.0552 - val_output2_accuracy: 0.7917 - val
_output2_loss: 0.8321 - learning_rate: 2.0000e-04
Epoch 26/100
703/703 - 6s - 9ms/step - loss: 1.6721 - output1_accuracy: 0.7373 - output1_loss: 0.
9096 - output2_accuracy: 0.8242 - output2_loss: 0.7625 - val_loss: 1.8605 - val_outp
ut1_accuracy: 0.6683 - val_output1_loss: 1.0377 - val_output2_accuracy: 0.7945 - val
_output2_loss: 0.8228 - learning_rate: 2.0000e-04
Epoch 27/100
703/703 - 6s - 9ms/step - loss: 1.6600 - output1_accuracy: 0.7427 - output1_loss: 0.
9017 - output2_accuracy: 0.8264 - output2_loss: 0.7582 - val_loss: 1.8551 - val_outp
ut1_accuracy: 0.6793 - val_output1_loss: 1.0142 - val_output2_accuracy: 0.7788 - val
_output2_loss: 0.8409 - learning_rate: 2.0000e-04
Epoch 28/100
703/703 - 6s - 9ms/step - loss: 1.6620 - output1_accuracy: 0.7439 - output1_loss: 0.
8995 - output2_accuracy: 0.8232 - output2_loss: 0.7625 - val_loss: 1.8312 - val_outp
ut1_accuracy: 0.6861 - val_output1_loss: 1.0133 - val_output2_accuracy: 0.7915 - val
_output2_loss: 0.8179 - learning_rate: 2.0000e-04
Epoch 29/100
703/703 - 7s - 9ms/step - loss: 1.6519 - output1_accuracy: 0.7453 - output1_loss: 0.
8981 - output2_accuracy: 0.8295 - output2_loss: 0.7538 - val_loss: 1.8502 - val_outp
ut1_accuracy: 0.6711 - val_output1_loss: 1.0368 - val_output2_accuracy: 0.8017 - val
_output2_loss: 0.8135 - learning_rate: 2.0000e-04
Epoch 30/100
703/703 - 6s - 9ms/step - loss: 1.6495 - output1_accuracy: 0.7471 - output1_loss: 0.
8928 - output2_accuracy: 0.8274 - output2_loss: 0.7567 - val_loss: 1.8205 - val_outp
ut1_accuracy: 0.6913 - val_output1_loss: 1.0001 - val_output2_accuracy: 0.7937 - val
_output2_loss: 0.8203 - learning_rate: 2.0000e-04
Epoch 31/100
703/703 - 6s - 9ms/step - loss: 1.6414 - output1_accuracy: 0.7477 - output1_loss: 0.
8894 - output2_accuracy: 0.8291 - output2_loss: 0.7519 - val_loss: 1.8476 - val_outp
ut1_accuracy: 0.6689 - val_output1_loss: 1.0203 - val_output2_accuracy: 0.7804 - val
_output2_loss: 0.8273 - learning_rate: 2.0000e-04
Epoch 32/100
703/703 - 6s - 9ms/step - loss: 1.6270 - output1_accuracy: 0.7518 - output1_loss: 0.
8851 - output2_accuracy: 0.8342 - output2_loss: 0.7419 - val_loss: 1.8156 - val_outp
ut1_accuracy: 0.6835 - val_output1_loss: 1.0004 - val_output2_accuracy: 0.7985 - val
_output2_loss: 0.8152 - learning_rate: 2.0000e-04
Epoch 33/100
703/703 - 6s - 9ms/step - loss: 1.6342 - output1_accuracy: 0.7521 - output1_loss: 0.
8854 - output2_accuracy: 0.8305 - output2_loss: 0.7489 - val_loss: 1.8650 - val_outp
ut1_accuracy: 0.6777 - val_output1_loss: 1.0294 - val_output2_accuracy: 0.7885 - val
_output2_loss: 0.8356 - learning_rate: 2.0000e-04
Epoch 34/100
```

```
703/703 - 6s - 9ms/step - loss: 1.6128 - output1_accuracy: 0.7572 - output1_loss: 0.
8739 - output2_accuracy: 0.8357 - output2_loss: 0.7389 - val_loss: 1.8649 - val_outp
ut1_accuracy: 0.6795 - val_output1_loss: 1.0229 - val_output2_accuracy: 0.7825 - val
_output2_loss: 0.8420 - learning_rate: 2.0000e-04
Epoch 35/100
703/703 - 6s - 9ms/step - loss: 1.6220 - output1_accuracy: 0.7531 - output1_loss: 0.
8799 - output2_accuracy: 0.8337 - output2_loss: 0.7421 - val_loss: 1.8895 - val_outp
ut1_accuracy: 0.6635 - val_output1_loss: 1.0403 - val_output2_accuracy: 0.7770 - val
_output2_loss: 0.8492 - learning_rate: 2.0000e-04
Epoch 36/100
703/703 - 7s - 9ms/step - loss: 1.6144 - output1_accuracy: 0.7559 - output1_loss: 0.
8774 - output2_accuracy: 0.8365 - output2_loss: 0.7370 - val_loss: 1.8359 - val_outp
ut1_accuracy: 0.6727 - val_output1_loss: 1.0251 - val_output2_accuracy: 0.7959 - val
_output2_loss: 0.8108 - learning_rate: 2.0000e-04
Epoch 37/100

Epoch 37: ReduceLROnPlateau reducing learning rate to 4.0000001899898055e-05.
703/703 - 6s - 9ms/step - loss: 1.6063 - output1_accuracy: 0.7611 - output1_loss: 0.
8730 - output2_accuracy: 0.8386 - output2_loss: 0.7333 - val_loss: 1.8288 - val_outp
ut1_accuracy: 0.6837 - val_output1_loss: 1.0079 - val_output2_accuracy: 0.7913 - val
_output2_loss: 0.8210 - learning_rate: 2.0000e-04
Epoch 38/100
703/703 - 6s - 9ms/step - loss: 1.5878 - output1_accuracy: 0.7662 - output1_loss: 0.
8596 - output2_accuracy: 0.8421 - output2_loss: 0.7282 - val_loss: 1.8379 - val_outp
ut1_accuracy: 0.6837 - val_output1_loss: 1.0138 - val_output2_accuracy: 0.7923 - val
_output2_loss: 0.8242 - learning_rate: 4.0000e-05
Epoch 39/100
703/703 - 6s - 9ms/step - loss: 1.5778 - output1_accuracy: 0.7690 - output1_loss: 0.
8552 - output2_accuracy: 0.8457 - output2_loss: 0.7227 - val_loss: 1.8109 - val_outp
ut1_accuracy: 0.6849 - val_output1_loss: 1.0074 - val_output2_accuracy: 0.8029 - val
_output2_loss: 0.8035 - learning_rate: 4.0000e-05
Epoch 40/100
703/703 - 6s - 9ms/step - loss: 1.5809 - output1_accuracy: 0.7661 - output1_loss: 0.
8591 - output2_accuracy: 0.8454 - output2_loss: 0.7217 - val_loss: 1.7943 - val_outp
ut1_accuracy: 0.6909 - val_output1_loss: 0.9969 - val_output2_accuracy: 0.8091 - val
_output2_loss: 0.7974 - learning_rate: 4.0000e-05
Epoch 41/100
703/703 - 6s - 9ms/step - loss: 1.5768 - output1_accuracy: 0.7714 - output1_loss: 0.
8525 - output2_accuracy: 0.8459 - output2_loss: 0.7243 - val_loss: 1.8095 - val_outp
ut1_accuracy: 0.6875 - val_output1_loss: 1.0071 - val_output2_accuracy: 0.8035 - val
_output2_loss: 0.8024 - learning_rate: 4.0000e-05
Epoch 42/100
703/703 - 6s - 9ms/step - loss: 1.5690 - output1_accuracy: 0.7730 - output1_loss: 0.
8488 - output2_accuracy: 0.8461 - output2_loss: 0.7202 - val_loss: 1.8219 - val_outp
ut1_accuracy: 0.7007 - val_output1_loss: 0.9885 - val_output2_accuracy: 0.7778 - val
_output2_loss: 0.8334 - learning_rate: 4.0000e-05
Epoch 43/100
703/703 - 6s - 9ms/step - loss: 1.5666 - output1_accuracy: 0.7733 - output1_loss: 0.
8478 - output2_accuracy: 0.8482 - output2_loss: 0.7189 - val_loss: 1.7952 - val_outp
ut1_accuracy: 0.7055 - val_output1_loss: 0.9830 - val_output2_accuracy: 0.7945 - val
_output2_loss: 0.8122 - learning_rate: 4.0000e-05
Epoch 44/100
703/703 - 6s - 9ms/step - loss: 1.5738 - output1_accuracy: 0.7728 - output1_loss: 0.
8522 - output2_accuracy: 0.8438 - output2_loss: 0.7216 - val_loss: 1.7895 - val_outp
ut1_accuracy: 0.6915 - val_output1_loss: 0.9969 - val_output2_accuracy: 0.8033 - val
_output2_loss: 0.7925 - learning_rate: 4.0000e-05
```

```
Epoch 45/100
703/703 - 6s - 9ms/step - loss: 1.5596 - output1_accuracy: 0.7754 - output1_loss: 0.
8429 - output2_accuracy: 0.8495 - output2_loss: 0.7167 - val_loss: 1.8257 - val_outp
ut1_accuracy: 0.6859 - val_output1_loss: 1.0110 - val_output2_accuracy: 0.7981 - val
_output2_loss: 0.8146 - learning_rate: 4.0000e-05
Epoch 46/100
703/703 - 6s - 9ms/step - loss: 1.5566 - output1_accuracy: 0.7729 - output1_loss: 0.
8421 - output2_accuracy: 0.8494 - output2_loss: 0.7145 - val_loss: 1.8000 - val_outp
ut1_accuracy: 0.6943 - val_output1_loss: 0.9865 - val_output2_accuracy: 0.7953 - val
_output2_loss: 0.8135 - learning_rate: 4.0000e-05
Epoch 47/100
703/703 - 6s - 9ms/step - loss: 1.5642 - output1_accuracy: 0.7757 - output1_loss: 0.
8464 - output2_accuracy: 0.8477 - output2_loss: 0.7179 - val_loss: 1.8108 - val_outp
ut1_accuracy: 0.6865 - val_output1_loss: 1.0007 - val_output2_accuracy: 0.8041 - val
_output2_loss: 0.8101 - learning_rate: 4.0000e-05
Epoch 48/100
703/703 - 6s - 9ms/step - loss: 1.5587 - output1_accuracy: 0.7736 - output1_loss: 0.
8479 - output2_accuracy: 0.8518 - output2_loss: 0.7108 - val_loss: 1.7916 - val_outp
ut1_accuracy: 0.6977 - val_output1_loss: 0.9894 - val_output2_accuracy: 0.8069 - val
_output2_loss: 0.8022 - learning_rate: 4.0000e-05
Epoch 49/100

Epoch 49: ReduceLROnPlateau reducing learning rate to 8.000000525498762e-06.
703/703 - 6s - 9ms/step - loss: 1.5552 - output1_accuracy: 0.7742 - output1_loss: 0.
8455 - output2_accuracy: 0.8526 - output2_loss: 0.7096 - val_loss: 1.8237 - val_outp
ut1_accuracy: 0.6943 - val_output1_loss: 1.0009 - val_output2_accuracy: 0.7961 - val
_output2_loss: 0.8228 - learning_rate: 4.0000e-05
Epoch 50/100
703/703 - 6s - 9ms/step - loss: 1.5514 - output1_accuracy: 0.7786 - output1_loss: 0.
8398 - output2_accuracy: 0.8515 - output2_loss: 0.7116 - val_loss: 1.8080 - val_outp
ut1_accuracy: 0.6975 - val_output1_loss: 0.9908 - val_output2_accuracy: 0.7971 - val
_output2_loss: 0.8172 - learning_rate: 8.0000e-06
Epoch 51/100
703/703 - 6s - 9ms/step - loss: 1.5434 - output1_accuracy: 0.7806 - output1_loss: 0.
8353 - output2_accuracy: 0.8543 - output2_loss: 0.7081 - val_loss: 1.7925 - val_outp
ut1_accuracy: 0.7053 - val_output1_loss: 0.9830 - val_output2_accuracy: 0.7961 - val
_output2_loss: 0.8095 - learning_rate: 8.0000e-06
Epoch 52/100
703/703 - 6s - 9ms/step - loss: 1.5576 - output1_accuracy: 0.7752 - output1_loss: 0.
8456 - output2_accuracy: 0.8491 - output2_loss: 0.7121 - val_loss: 1.7893 - val_outp
ut1_accuracy: 0.6911 - val_output1_loss: 0.9952 - val_output2_accuracy: 0.8081 - val
_output2_loss: 0.7941 - learning_rate: 8.0000e-06
Epoch 53/100
703/703 - 6s - 9ms/step - loss: 1.5462 - output1_accuracy: 0.7804 - output1_loss: 0.
8365 - output2_accuracy: 0.8517 - output2_loss: 0.7098 - val_loss: 1.7868 - val_outp
ut1_accuracy: 0.6895 - val_output1_loss: 0.9938 - val_output2_accuracy: 0.7997 - val
_output2_loss: 0.7931 - learning_rate: 8.0000e-06
Epoch 54/100
703/703 - 6s - 9ms/step - loss: 1.5485 - output1_accuracy: 0.7794 - output1_loss: 0.
8373 - output2_accuracy: 0.8516 - output2_loss: 0.7111 - val_loss: 1.8034 - val_outp
ut1_accuracy: 0.6929 - val_output1_loss: 0.9944 - val_output2_accuracy: 0.7971 - val
_output2_loss: 0.8090 - learning_rate: 8.0000e-06
Epoch 55/100
703/703 - 6s - 9ms/step - loss: 1.5425 - output1_accuracy: 0.7796 - output1_loss: 0.
8364 - output2_accuracy: 0.8540 - output2_loss: 0.7061 - val_loss: 1.8112 - val_outp
ut1_accuracy: 0.6853 - val_output1_loss: 0.9987 - val_output2_accuracy: 0.7943 - val
```

```
_output2_loss: 0.8125 - learning_rate: 8.0000e-06
Epoch 56/100
703/703 - 7s - 10ms/step - loss: 1.5471 - output1_accuracy: 0.7778 - output1_loss:
0.8371 - output2_accuracy: 0.8521 - output2_loss: 0.7100 - val_loss: 1.7995 - val_ou
tput1_accuracy: 0.6871 - val_output1_loss: 1.0029 - val_output2_accuracy: 0.8079 - v
al_output2_loss: 0.7966 - learning_rate: 8.0000e-06
Epoch 57/100
703/703 - 6s - 9ms/step - loss: 1.5485 - output1_accuracy: 0.7792 - output1_loss: 0.
8388 - output2_accuracy: 0.8538 - output2_loss: 0.7097 - val_loss: 1.7974 - val_outp
ut1_accuracy: 0.6989 - val_output1_loss: 0.9895 - val_output2_accuracy: 0.7989 - val
_output2_loss: 0.8079 - learning_rate: 8.0000e-06
Epoch 58/100

Epoch 58: ReduceLROnPlateau reducing learning rate to 1.6000001778593287e-06.
703/703 - 6s - 9ms/step - loss: 1.5441 - output1_accuracy: 0.7790 - output1_loss: 0.
8377 - output2_accuracy: 0.8537 - output2_loss: 0.7064 - val_loss: 1.8376 - val_outp
ut1_accuracy: 0.6799 - val_output1_loss: 1.0243 - val_output2_accuracy: 0.7967 - val
_output2_loss: 0.8133 - learning_rate: 8.0000e-06
Epoch 59/100
703/703 - 6s - 9ms/step - loss: 1.5478 - output1_accuracy: 0.7770 - output1_loss: 0.
8399 - output2_accuracy: 0.8518 - output2_loss: 0.7078 - val_loss: 1.8080 - val_outp
ut1_accuracy: 0.6907 - val_output1_loss: 0.9946 - val_output2_accuracy: 0.7959 - val
_output2_loss: 0.8134 - learning_rate: 1.6000e-06
Epoch 60/100
703/703 - 6s - 9ms/step - loss: 1.5494 - output1_accuracy: 0.7787 - output1_loss: 0.
8418 - output2_accuracy: 0.8529 - output2_loss: 0.7076 - val_loss: 1.7818 - val_outp
ut1_accuracy: 0.6949 - val_output1_loss: 0.9890 - val_output2_accuracy: 0.8077 - val
_output2_loss: 0.7928 - learning_rate: 1.6000e-06
Epoch 61/100
703/703 - 6s - 9ms/step - loss: 1.5423 - output1_accuracy: 0.7795 - output1_loss: 0.
8347 - output2_accuracy: 0.8518 - output2_loss: 0.7077 - val_loss: 1.8172 - val_outp
ut1_accuracy: 0.6809 - val_output1_loss: 1.0111 - val_output2_accuracy: 0.7993 - val
_output2_loss: 0.8061 - learning_rate: 1.6000e-06
Epoch 62/100
703/703 - 6s - 9ms/step - loss: 1.5431 - output1_accuracy: 0.7788 - output1_loss: 0.
8364 - output2_accuracy: 0.8537 - output2_loss: 0.7067 - val_loss: 1.8066 - val_outp
ut1_accuracy: 0.6917 - val_output1_loss: 1.0021 - val_output2_accuracy: 0.8013 - val
_output2_loss: 0.8046 - learning_rate: 1.6000e-06
Epoch 63/100
703/703 - 6s - 9ms/step - loss: 1.5451 - output1_accuracy: 0.7767 - output1_loss: 0.
8382 - output2_accuracy: 0.8548 - output2_loss: 0.7069 - val_loss: 1.8013 - val_outp
ut1_accuracy: 0.6931 - val_output1_loss: 1.0054 - val_output2_accuracy: 0.8031 - val
_output2_loss: 0.7959 - learning_rate: 1.6000e-06
Epoch 64/100
703/703 - 6s - 9ms/step - loss: 1.5483 - output1_accuracy: 0.7774 - output1_loss: 0.
8377 - output2_accuracy: 0.8514 - output2_loss: 0.7106 - val_loss: 1.7897 - val_outp
ut1_accuracy: 0.6927 - val_output1_loss: 0.9902 - val_output2_accuracy: 0.8077 - val
_output2_loss: 0.7994 - learning_rate: 1.6000e-06
Epoch 65/100

Epoch 65: ReduceLROnPlateau reducing learning rate to 3.200000264769187e-07.
703/703 - 6s - 9ms/step - loss: 1.5452 - output1_accuracy: 0.7764 - output1_loss: 0.
8414 - output2_accuracy: 0.8571 - output2_loss: 0.7038 - val_loss: 1.7841 - val_outp
ut1_accuracy: 0.6983 - val_output1_loss: 0.9867 - val_output2_accuracy: 0.8031 - val
_output2_loss: 0.7973 - learning_rate: 1.6000e-06
Epoch 66/100
```

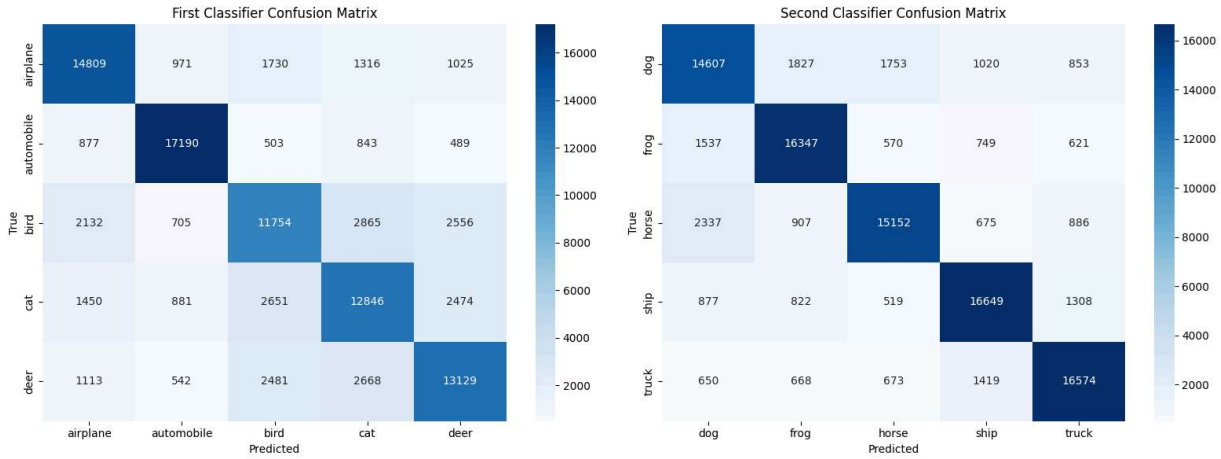
```
703/703 - 6s - 9ms/step - loss: 1.5428 - output1_accuracy: 0.7823 - output1_loss: 0.
8365 - output2_accuracy: 0.8524 - output2_loss: 0.7064 - val_loss: 1.7771 - val_outp
ut1_accuracy: 0.7013 - val_output1_loss: 0.9784 - val_output2_accuracy: 0.8061 - val
_output2_loss: 0.7987 - learning_rate: 3.2000e-07
Epoch 67/100
703/703 - 6s - 9ms/step - loss: 1.5456 - output1_accuracy: 0.7804 - output1_loss: 0.
8358 - output2_accuracy: 0.8523 - output2_loss: 0.7098 - val_loss: 1.7707 - val_outp
ut1_accuracy: 0.6961 - val_output1_loss: 0.9845 - val_output2_accuracy: 0.8085 - val
_output2_loss: 0.7862 - learning_rate: 3.2000e-07
Epoch 68/100
703/703 - 6s - 9ms/step - loss: 1.5455 - output1_accuracy: 0.7787 - output1_loss: 0.
8402 - output2_accuracy: 0.8542 - output2_loss: 0.7052 - val_loss: 1.8024 - val_outp
ut1_accuracy: 0.7001 - val_output1_loss: 0.9893 - val_output2_accuracy: 0.7953 - val
_output2_loss: 0.8131 - learning_rate: 3.2000e-07
Epoch 69/100
703/703 - 6s - 9ms/step - loss: 1.5477 - output1_accuracy: 0.7794 - output1_loss: 0.
8372 - output2_accuracy: 0.8527 - output2_loss: 0.7105 - val_loss: 1.8092 - val_outp
ut1_accuracy: 0.6821 - val_output1_loss: 1.0081 - val_output2_accuracy: 0.8067 - val
_output2_loss: 0.8011 - learning_rate: 3.2000e-07
Epoch 70/100
703/703 - 6s - 9ms/step - loss: 1.5414 - output1_accuracy: 0.7827 - output1_loss: 0.
8347 - output2_accuracy: 0.8549 - output2_loss: 0.7067 - val_loss: 1.7873 - val_outp
ut1_accuracy: 0.6967 - val_output1_loss: 0.9776 - val_output2_accuracy: 0.7967 - val
_output2_loss: 0.8097 - learning_rate: 3.2000e-07
Epoch 71/100
703/703 - 6s - 9ms/step - loss: 1.5456 - output1_accuracy: 0.7804 - output1_loss: 0.
8374 - output2_accuracy: 0.8516 - output2_loss: 0.7081 - val_loss: 1.7917 - val_outp
ut1_accuracy: 0.6955 - val_output1_loss: 0.9922 - val_output2_accuracy: 0.8005 - val
_output2_loss: 0.7995 - learning_rate: 3.2000e-07
Epoch 72/100

Epoch 72: ReduceLROnPlateau reducing learning rate to 6.400000529538374e-08.
703/703 - 6s - 9ms/step - loss: 1.5478 - output1_accuracy: 0.7789 - output1_loss: 0.
8381 - output2_accuracy: 0.8521 - output2_loss: 0.7096 - val_loss: 1.8064 - val_outp
ut1_accuracy: 0.6939 - val_output1_loss: 0.9989 - val_output2_accuracy: 0.7927 - val
_output2_loss: 0.8074 - learning_rate: 3.2000e-07
Epoch 73/100
703/703 - 6s - 9ms/step - loss: 1.5432 - output1_accuracy: 0.7802 - output1_loss: 0.
8365 - output2_accuracy: 0.8549 - output2_loss: 0.7067 - val_loss: 1.7929 - val_outp
ut1_accuracy: 0.6981 - val_output1_loss: 0.9881 - val_output2_accuracy: 0.7953 - val
_output2_loss: 0.8048 - learning_rate: 6.4000e-08
Epoch 74/100
703/703 - 6s - 9ms/step - loss: 1.5450 - output1_accuracy: 0.7782 - output1_loss: 0.
8377 - output2_accuracy: 0.8530 - output2_loss: 0.7073 - val_loss: 1.8223 - val_outp
ut1_accuracy: 0.6791 - val_output1_loss: 1.0166 - val_output2_accuracy: 0.8011 - val
_output2_loss: 0.8057 - learning_rate: 6.4000e-08
Epoch 75/100
703/703 - 6s - 9ms/step - loss: 1.5439 - output1_accuracy: 0.7767 - output1_loss: 0.
8390 - output2_accuracy: 0.8546 - output2_loss: 0.7049 - val_loss: 1.8129 - val_outp
ut1_accuracy: 0.6899 - val_output1_loss: 1.0097 - val_output2_accuracy: 0.8083 - val
_output2_loss: 0.8032 - learning_rate: 6.4000e-08
Epoch 76/100
703/703 - 6s - 9ms/step - loss: 1.5452 - output1_accuracy: 0.7817 - output1_loss: 0.
8368 - output2_accuracy: 0.8512 - output2_loss: 0.7084 - val_loss: 1.8055 - val_outp
ut1_accuracy: 0.6885 - val_output1_loss: 0.9995 - val_output2_accuracy: 0.8029 - val
_output2_loss: 0.8060 - learning_rate: 6.4000e-08
```

```
Epoch 77/100

Epoch 77: ReduceLROnPlateau reducing learning rate to 1.2800001059076749e-08.
703/703 - 6s - 9ms/step - loss: 1.5502 - output1_accuracy: 0.7800 - output1_loss: 0.
8380 - output2_accuracy: 0.8511 - output2_loss: 0.7122 - val_loss: 1.8012 - val_outp
ut1_accuracy: 0.6879 - val_output1_loss: 1.0028 - val_output2_accuracy: 0.8059 - val
_output2_loss: 0.7985 - learning_rate: 6.4000e-08
Epoch 78/100
703/703 - 6s - 9ms/step - loss: 1.5396 - output1_accuracy: 0.7843 - output1_loss: 0.
8314 - output2_accuracy: 0.8540 - output2_loss: 0.7082 - val_loss: 1.7964 - val_outp
ut1_accuracy: 0.6993 - val_output1_loss: 0.9856 - val_output2_accuracy: 0.7953 - val
_output2_loss: 0.8108 - learning_rate: 1.2800e-08
Epoch 79/100
703/703 - 6s - 9ms/step - loss: 1.5408 - output1_accuracy: 0.7811 - output1_loss: 0.
8339 - output2_accuracy: 0.8546 - output2_loss: 0.7069 - val_loss: 1.8117 - val_outp
ut1_accuracy: 0.6895 - val_output1_loss: 0.9992 - val_output2_accuracy: 0.7959 - val
_output2_loss: 0.8125 - learning_rate: 1.2800e-08
Epoch 80/100
703/703 - 6s - 9ms/step - loss: 1.5478 - output1_accuracy: 0.7769 - output1_loss: 0.
8392 - output2_accuracy: 0.8533 - output2_loss: 0.7086 - val_loss: 1.7777 - val_outp
ut1_accuracy: 0.6977 - val_output1_loss: 0.9853 - val_output2_accuracy: 0.8091 - val
_output2_loss: 0.7924 - learning_rate: 1.2800e-08
Epoch 81/100
703/703 - 7s - 9ms/step - loss: 1.5481 - output1_accuracy: 0.7797 - output1_loss: 0.
8393 - output2_accuracy: 0.8525 - output2_loss: 0.7088 - val_loss: 1.7903 - val_outp
ut1_accuracy: 0.6997 - val_output1_loss: 0.9880 - val_output2_accuracy: 0.8049 - val
_output2_loss: 0.8023 - learning_rate: 1.2800e-08
Epoch 82/100

Epoch 82: ReduceLROnPlateau reducing learning rate to 2.5600002118153498e-09.
703/703 - 6s - 9ms/step - loss: 1.5459 - output1_accuracy: 0.7783 - output1_loss: 0.
8368 - output2_accuracy: 0.8533 - output2_loss: 0.7091 - val_loss: 1.7933 - val_outp
ut1_accuracy: 0.6959 - val_output1_loss: 0.9898 - val_output2_accuracy: 0.7995 - val
_output2_loss: 0.8035 - learning_rate: 1.2800e-08
Epoch 83/100
703/703 - 6s - 9ms/step - loss: 1.5476 - output1_accuracy: 0.7779 - output1_loss: 0.
8366 - output2_accuracy: 0.8524 - output2_loss: 0.7110 - val_loss: 1.7854 - val_outp
ut1_accuracy: 0.7013 - val_output1_loss: 0.9851 - val_output2_accuracy: 0.7985 - val
_output2_loss: 0.8004 - learning_rate: 2.5600e-09
Epoch 84/100
703/703 - 6s - 9ms/step - loss: 1.5392 - output1_accuracy: 0.7819 - output1_loss: 0.
8322 - output2_accuracy: 0.8528 - output2_loss: 0.7070 - val_loss: 1.8184 - val_outp
ut1_accuracy: 0.6947 - val_output1_loss: 1.0076 - val_output2_accuracy: 0.8019 - val
_output2_loss: 0.8108 - learning_rate: 2.5600e-09
Epoch 85/100
703/703 - 6s - 9ms/step - loss: 1.5465 - output1_accuracy: 0.7790 - output1_loss: 0.
8370 - output2_accuracy: 0.8515 - output2_loss: 0.7095 - val_loss: 1.8023 - val_outp
ut1_accuracy: 0.6887 - val_output1_loss: 0.9996 - val_output2_accuracy: 0.8017 - val
_output2_loss: 0.8028 - learning_rate: 2.5600e-09
Epoch 86/100
703/703 - 6s - 9ms/step - loss: 1.5398 - output1_accuracy: 0.7793 - output1_loss: 0.
8377 - output2_accuracy: 0.8567 - output2_loss: 0.7021 - val_loss: 1.8152 - val_outp
ut1_accuracy: 0.6895 - val_output1_loss: 1.0002 - val_output2_accuracy: 0.7951 - val
_output2_loss: 0.8150 - learning_rate: 2.5600e-09
Epoch 87/100
```

```
Epoch 87: ReduceLROnPlateau reducing learning rate to 1e-09.
703/703 - 7s - 9ms/step - loss: 1.5463 - output1_accuracy: 0.7789 - output1_loss: 0.
8378 - output2_accuracy: 0.8526 - output2_loss: 0.7085 - val_loss: 1.8109 - val_outp
ut1_accuracy: 0.6859 - val_output1_loss: 0.9921 - val_output2_accuracy: 0.7933 - val
_output2_loss: 0.8189 - learning_rate: 2.5600e-09
Epoch 87: early stopping
Restoring model weights from the end of the best epoch: 67.
```

In [14]:
```
evaluate_model(model, testgen)
```

```
100%|██████████| 100/100 [00:31<00:00,  3.14it/s]
First Classifier Accuracy: 0.6973
Second Classifier Accuracy: 0.7933
Mean Accuracy: 0.7453 ± 0.0480
First Classifier F1 Score: 0.6965
Second Classifier F1 Score: 0.7931
Mean F1 Score: 0.7448 ± 0.0483
```



This model performs worse, so we did not use it. It does, however, illustrate the ease of use of experimentation in the Keras framework and with our code design. The reader is invited to try and improve on our results!