project3-final

December 8, 2024

0.1 Image Classification Exploration Through Neural Networks

This project attempts to explore and understand neural networks using a dataset of small images, CIFAR-10, provided through the University of Toronto.

0.1.1 What Are We Trying to Learn?

The specific learning problem we will be working on is image classification using neural network methods. Some time is spent examining image preprocessing and discussing effectuve methods to perform this. Afterward, the project describes the two major network types chosen; namely, the convolutional neural network and the vision transformer. The Hyperband tuner from the keras tuner package is used to perform hyperparameter search, and some experiences from the project's results and inter-model performance is explored.

0.1.2 Why Does It Matter?

This project is for the CU Boulder CSPB machine learning course. While regular course material is both relevent and plentiful, the realm of machine learning techniques and tools is incredibly vast and cannot be covered only by reading and standard homework material. An opportunity to explore is invaluable, because it involves attempting to answer unexpected questions and grows experience in a way which is more personable.

0.1.3 Data Link and Info

The CIFAR-10 dataset is a well known image dataset of sixty thousand 32 x 32 color images containing ten different classes. Originating from the Canadian Institute for Advanced Research (CIFAR) it was introduced by Alex Krizhevsky and other CIFAR collaborators in 2009. Its development was primarily driven by research in image classification and machine learning, aiming to provide a standardized benchmark for evaluating image recognition algorithms. The images were compiled from the 80 Million Tiny Images dataset, created by Antonio Torralba, however this original dataset is nolonger available. The data as it has been provided by Krizhevsky is stored as binary values, which can be unpickled (through the pickle python library) into numpy arrays. These arrays organize pixel values for each image as a flattened sequence of 8 bit integers ordered as red values, green values, and blue values.

Class labels included for the data are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each batch holds ten thousand observations stored as 'uint8's, meaning values range from 0 to 255 for any pixel. Default batching has done a good job of evenly distributing classes. However, we will lump together the training observations in order to examine them here, and later perform shuffling to better guarantee a lack of patterns in the observations during training.

0.1.4 Imports, Basic Info, and Some Example Observations

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
import copy
import scipy.stats as stats
import time
import itertools
from scipy.stats import zscore
from sklearn.utils import shuffle
from sklearn.metrics import confusion_matrix, roc_auc_score, roc_curve, auc
from sklearn.metrics import RocCurveDisplay
from sklearn.preprocessing import LabelBinarizer
from sklearn.preprocessing import StandardScaler, MinMaxScaler

import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '1'

import tensorflow as tf
from tensorflow import keras
import keras_tuner
from tensorflow.keras.models import Sequential
from keras.utils import to_categorical
from tensorflow.keras.layers import (Layer, Dense, Activation, Conv2D,␣
 ↪MaxPool2D, Flatten, GaussianNoise, Reshape,
                                     Embedding, LayerNormalization, Add,␣
 ↪Dropout, MultiHeadAttention)
from tensorflow.keras.optimizers import Adam, SGD
from tensorflow.keras.regularizers import l2
from tensorflow.keras.metrics import categorical_crossentropy
from tensorflow.keras.callbacks import EarlyStopping

SEED = 42

def unpickle(path):
    with open(path, 'rb') as file:
        dict = pickle.load(file, encoding='bytes')
    return dict

data_batch_1 = unpickle('data/cifar-10-batches-py/data_batch_1')
data_batch_2 = unpickle('data/cifar-10-batches-py/data_batch_2')
data_batch_3 = unpickle('data/cifar-10-batches-py/data_batch_3')
data_batch_4 = unpickle('data/cifar-10-batches-py/data_batch_4')
data_batch_5 = unpickle('data/cifar-10-batches-py/data_batch_5')
```

```
test_batch = unpickle('data/cifar-10-batches-py/test_batch')
```

0.1.5 The Data as Provided

[2]:
```
print(data_batch_1.keys())
print(data_batch_1[b'data'])
```

```
dict_keys([b'batch_label', b'labels', b'data', b'filenames'])
[[ 59  43  50 … 140  84  72]
 [154 126 105 … 139 142 144]
 [255 253 253 …  83  83  84]
 …
 [ 71  60  74 …  68  69  68]
 [250 254 211 … 215 255 254]
 [ 62  61  60 … 130 130 131]]
```

[3]:
```
print("What balance does train or test data have?")

tr_labels = np.concatenate((data_batch_1[b'labels'],
                        data_batch_2[b'labels'],
                        data_batch_3[b'labels'],
                        data_batch_4[b'labels'],
                        data_batch_5[b'labels']
                            ), axis=0)
te_labels = test_batch[b'labels']

tr_encoded_labels, tr_counts = np.unique(tr_labels, return_counts=True)
te_encoded_labels, te_counts = np.unique(te_labels, return_counts=True)

print(95 * "_")
print('data\tairplane  ␣
 ↪automobile\tbird\tcat\tdeer\tdog\tfrog\thorse\tship\ttruck')
form = "{:9s} {:.1f}%     {:.1f}%       {:.1f}%   {:.1f}%   {:.1f}%\t{:.1f}%\t{:
 ↪.1f}%\t{:.1f}%\t{:.1f}%\t{:.1f}%"
tr_counts = ['train'] + list(tr_counts / len(tr_labels) * 100)
te_counts = ['test'] + list(te_counts / len(te_labels) * 100)
print(form.format(*tr_counts))
print(form.format(*te_counts))
print(95 * "_")
```

```
What balance does train or test data have?
_______________________________________________________________________________
---------------
data    airplane    automobile    bird    cat     deer    dog     frog    horse
ship    truck
train     10.0%     10.0%       10.0%   10.0%   10.0%   10.0%   10.0%   10.0%
```

```
10.0%   10.0%
test      10.0%     10.0%       10.0%   10.0%   10.0%   10.0%   10.0%   10.0%
10.0%   10.0%
--------------------------------------------------------------------------------
--------------
```

[4]:
```python
def perchannel_histogram(data, name="", r=(0, 255), channels_last=False):
    if channels_last == False:
        red_ch = data[:, :1, :, :]
        green_ch = data[:, 1:2, :, :]
        blue_ch = data[:, 2:3, :, :]
    else:
        red_ch = data[:, :, :, :1]
        green_ch = data[:, :, :, 1:2]
        blue_ch = data[:, :, :, 2:3]

    red_hist, bins = np.histogram(red_ch, bins=2560, range=r)
    green_hist, _ = np.histogram(green_ch, bins=2560, range=r)
    blue_hist, _ = np.histogram(blue_ch, bins=2560, range=r)

    plt.figure(figsize=(10, 6))
    plt.plot(bins[:-1], red_hist, color='red', alpha=.9, label='Red Channel')
    plt.plot(bins[:-1], green_hist, color='green', alpha=.7, label='Green␣
 ↪Channel')
    plt.plot(bins[:-1], blue_hist, color='blue', alpha=.5, label='Blue Channel')
    plt.xlabel("Pixel Value")
    plt.ylabel("Frequency")
    plt.legend()
    plt.title(f"Per-Channel Histograms {name}")
    plt.show()


def zero_one(data, channels_last=False):
    if channels_last == False:
        data = data - data.min(axis=(2,3))[:, :, np.newaxis, np.newaxis]
        data = data / (data.max(axis=(2,3)) - data.min(axis=(2,3)))[:, :, np.
 ↪newaxis, np.newaxis]
        return np.moveaxis(data, 1, -1)
    else:
        data = data - data.min(axis=(1,2))[:, np.newaxis, np.newaxis, :]
        data = data / (data.max(axis=(1,2)) - data.min(axis=(1,2)))[:, np.
 ↪newaxis, np.newaxis, :]
        return data

def imageplot(arr):
    fig, axes = plt.subplots(1, 10, figsize=(20,20))
    axes = axes.flatten()
```

```
    if arr.min() <= 0 and arr.max() <= 1:
        arr = (arr * 255).astype(np.uint8)
    for img, ax in zip(arr, axes):
        ax.imshow(img)
        ax.axis('off')
    plt.tight_layout()
    plt.show()


def historyplot(history, item):
    plt.plot(history.history[item], label=item)
    plt.plot(history.history["val_" + item], label="val_" + item)
    plt.xlabel("Epochs")
    plt.ylabel(item)
    plt.title(f"Train and Validation {item} Over Epochs")
    plt.legend()
    plt.grid()
    plt.show()

def high_pass_filter(data, width=0.01):
    images, channels, rows, cols = data.shape
    filtered_images = np.zeros_like(data)

    x, y = np.meshgrid(np.arange(cols), np.arange(rows))

    x = zscore(x, axis=None)
    y = zscore(y, axis=None)

    gaus3d = (1 - np.exp(-(x**2 + y**2) / (2 * width**2)))[np.newaxis, np.
 ↪newaxis, :, :]

    ft_data = np.fft.fftshift(np.fft.fft2(data))

    filtered_data = ft_data * gaus3d

    ift_data = np.fft.ifft2(np.fft.ifftshift(filtered_data))

    real_data = np.real(ift_data)

    return real_data


def low_pass_filter(data, width=0.01):
    images, channels, rows, cols = data.shape
    filtered_images = np.zeros_like(data)

    x, y = np.meshgrid(np.arange(cols), np.arange(rows))
```

```
    x = zscore(x, axis=None)
    y = zscore(y, axis=None)

    gaus3d = (np.exp(-(x**2 + y**2) / (2 * width**2)))[np.newaxis, np.newaxis, :
 ↪, :]

    ft_data = np.fft.fftshift(np.fft.fft2(data))

    filtered_data = ft_data * gaus3d

    ift_data = np.fft.ifft2(np.fft.ifftshift(filtered_data))

    real_data = np.real(ift_data)

    return real_data

def log_filter(data, c=0.01):

    ft_data = np.fft.fftshift(np.fft.fft2(data))

    filtered_data = log_magnitude = np.log1p(np.abs(ft_data))

    ift_data = np.fft.ifft2(np.fft.ifftshift(filtered_data))

    real_data = np.real(ift_data)

    return real_data


orig_sample = np.moveaxis(data_batch_1[b'data'][:10].reshape(10, 3, 32, 32), 1,␣
 ↪-1)
```

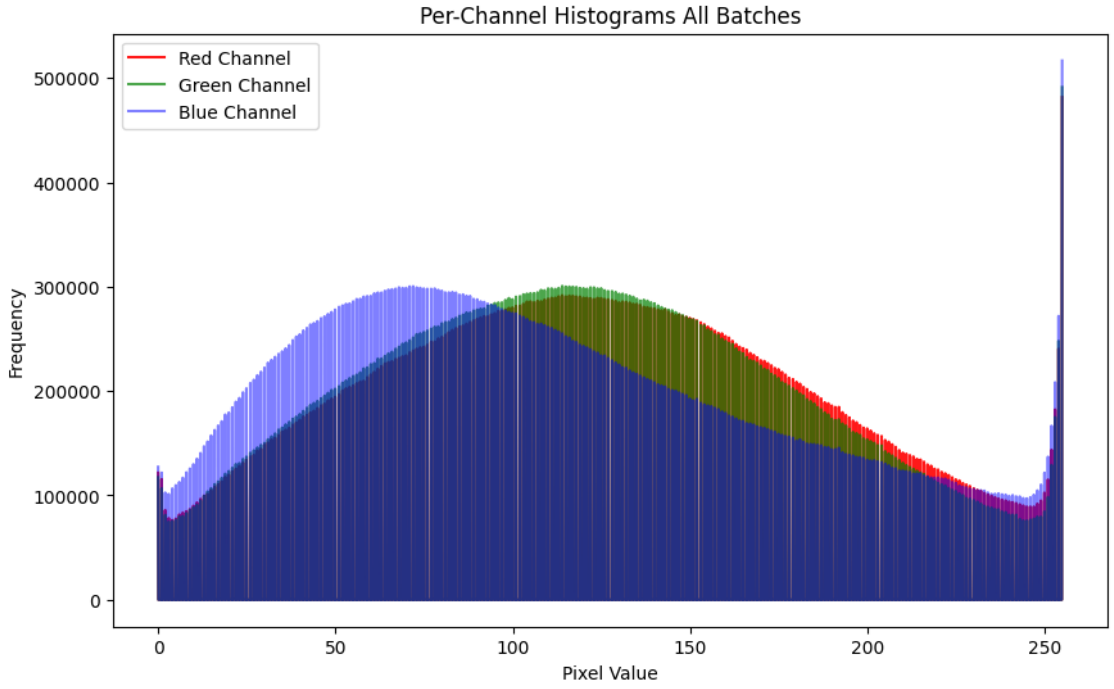
0.1.6 Examining the Dataset Values Collectively

Below we plot histograms for the pixel values of all the training images. Doing so reveals a number of characteristics about the data collectively. This set of lecture slides was very helpful for making some of these interpretations. The overall behavior across each channel is very normal (distribution shape) although we can see that there exist peaks to the intensity distributions at the extreme ends of the value range; this is mostly at the right-hand end where colors would be brightest. This apparent "squash" in the distribution is probably caused by heightened saturation. Saturation with digital images is when illumination values outside the original camera sensor's range are set to its min or max values. Although this effect can also be imposed via post-processing. We can also see that for any channel there exists a pattern of gaps in values. These defects should actually be expected. Recurring gaps in intensity values is a direct consequence of image compression, which makes sense given that the actual image sizes of the dataset are quite small. These results lead to the conclusion that characteristics we can observe in the histogram here are the result of

collective processing performed when the images were originally amassed by the 80 Million Tiny Images dataset project.

These characteristics will have an impact on our project results. For example, the process by which a convolutional neural network alters image data into a smaller set of values involves applying a filter (values are adjusted by a set of weights and their position relative to other pixels) and some scheme to pool resulting values together. This shrinks the number of values for an observation image iteratively until few enough values are in play that they can be processed by a feed forward network to perform classification. The condition of this data resembles what we might see by performing one or more filtering and pooling steps, only without the benefit of access to the associated weights for our training. Another challenge presented by the images is visible further down. Due to the nature of web scraping, objects are captured in a somewhat random fashion. This lack of consistency means that features identified (visible characteristics of a particular object) by our classifiers will inevitably be unuseful for some portion of images for any given class. Although this seems inevitably true for any image dataset.

[5]:
```
raw = np.concatenate((data_batch_1[b'data'],
                      data_batch_2[b'data'],
                      data_batch_3[b'data'],
                      data_batch_4[b'data'],
                      data_batch_5[b'data']), axis=0)
raw = raw.reshape(len(raw), 3, 32, 32)

perchannel_histogram(raw, name="All Batches")
```



0.1.7 A Route for Further Preprocessing Possibilities

In terms of analysis, what may be the greatest issue for image preprocessing is the infinite possibilities. With a short limit on available time it is best to rely on standard technique. In our case, training will use pixel values which are normalized to lie between zero and one, which is consistently the approach used for examples referenced by this project. However, with the goal of exploration there is still value in examining how more involved preprocessing could be used. Effectiveness of different data transformations for this specific dataset where applied to similar learning methods could prove limited due to small image size, but without the benefit of lots of experimentation there are simple limits what we can know will improve performance for certain.

For image data analysis, the most effective approach found by this project is to use Fourier transform. The Fourier transform is a mathematical tool that represents a signal (i.e. what is typically a one-dimensional sequence of values, either continuous or discrete) as a complex-valued function of frequency, decomposing it into its constituent sinusoidal components. For processing we treat our images as a two-dimensional signal or a bi-variate function of x and y spatial dimensions. This two-dimensional Fourier transform then constructs a pair of real values, these are the coefficients of a complex number, from variations in the intensity of pixel values. The result of the transformation is said to map from the "spatial domain" (or the "time domain" in the case of one-dimensional data) to the "frequency domain". Literally, the two coefficients (per pixel) that result are interpreted as "amplitude spectrum" and "phase spectrum", where the amplitude represents the strength of each frequency component (the "components" are the output values retrieved from FT) and the phase encodes the positional alignment or spatial structure of those components (pixel intensities) in the original signal.

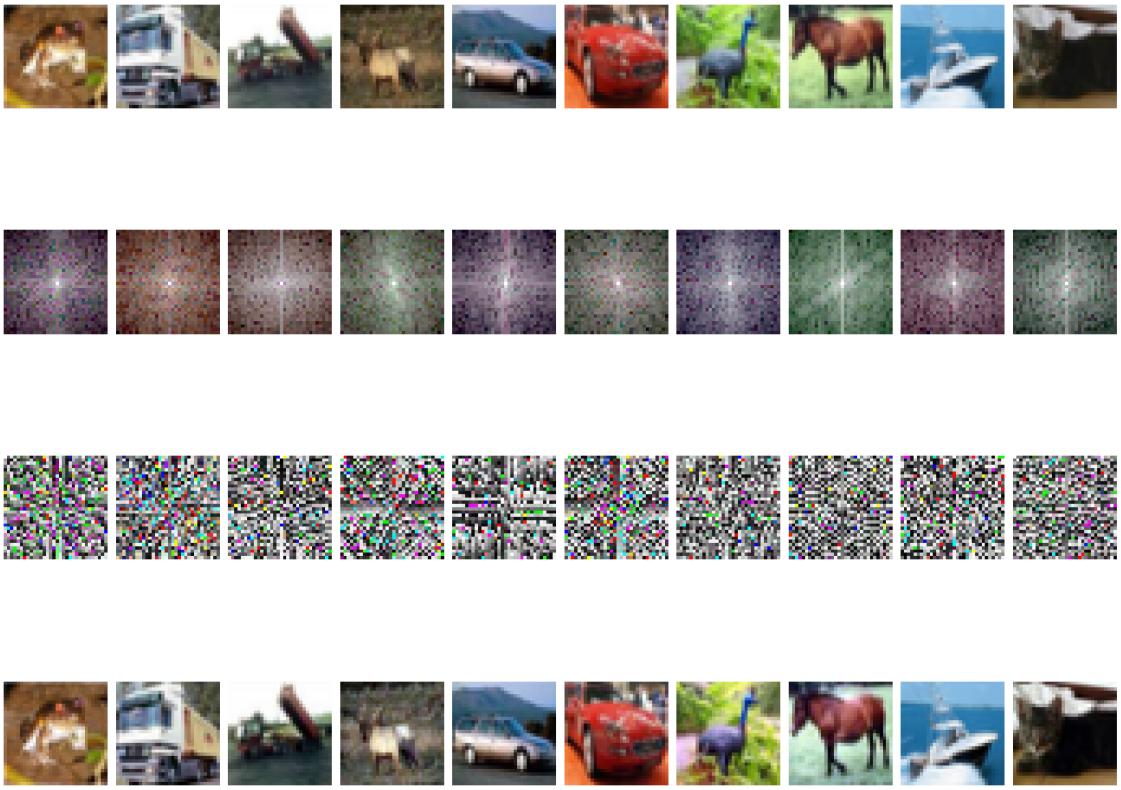
The amplitude spectrum reveals the dominant patterns or features in the image, such as edges or repeating textures, by showing how much each frequency contributes to the overall signal. Fascinatingly, the amplitude representation of the original spatial features dominates the center of the transformed data, localizing information from what were global patterns.

In contrast, the phase spectrum retains the critical information needed to reconstruct the image's spatial arrangement. Without the phase information, reconstruction would fail to reproduce the original signal accurately, even if the amplitudes were preserved.

Finally, by performing the inverse Fourier transform, the processed signal in the frequency domain is converted back to the spatial domain, effectively reconstructing the original image (or an altered version if filtering was applied). The inverse transform combines the amplitude and phase components to recreate the original intensities and spatial arrangement of the image pixels. This ability to toggle between spatial and frequency domains allows the Fourier transform to have very useful applications, which we explore below.

[6]:
```
samp = data_batch_1[b'data'][:10].reshape(10, 3, 32, 32)
samp = np.fft.fft2(samp)
samp = np.fft.fftshift(samp)

data_ampl = np.log(np.abs(samp) + 1)
data_phase = np.angle(samp)

imageplot(orig_sample)
imageplot(zero_one(data_ampl))
```

```
imageplot(zero_one(data_phase))
imageplot(zero_one(np.real(np.fft.ifft2(np.fft.ifftshift(samp)))))
```



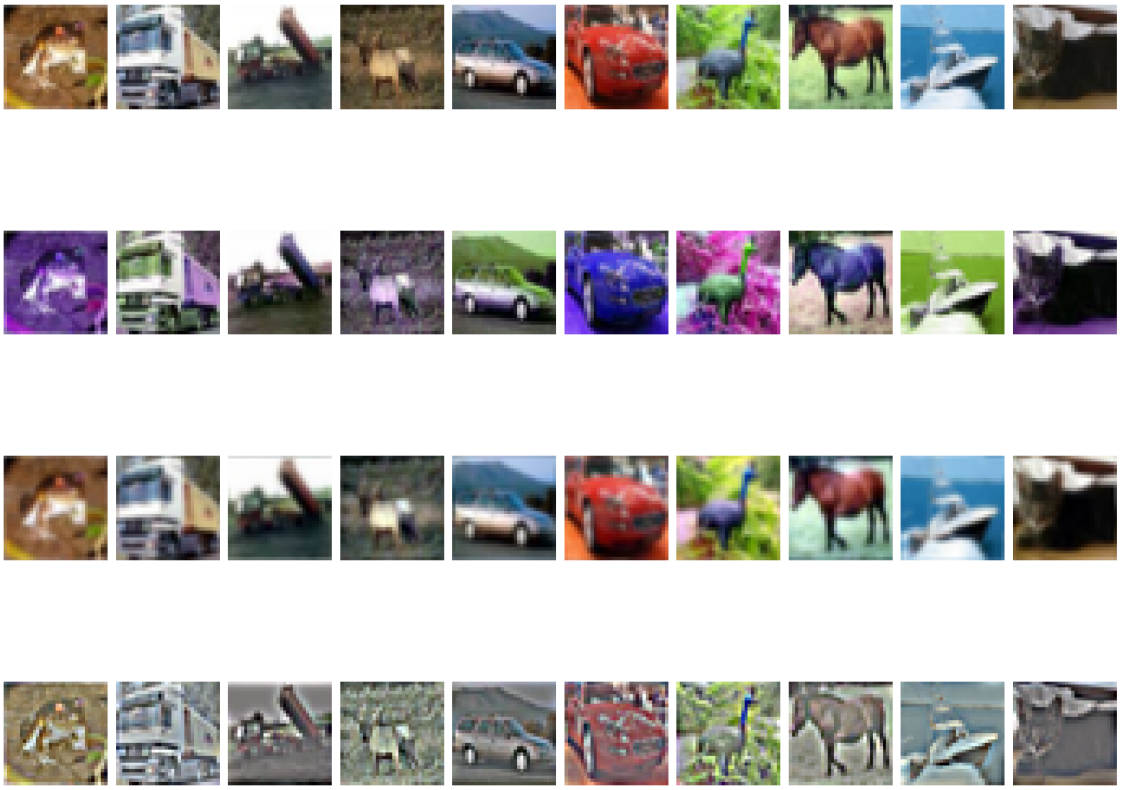
Above we see unaltered original data, followed by its amplitude, followed by its phase, and finally the inverse transformation returning the data back to its orignal composition. In the vast majority of cases neither phase nor amplitude are human interpretable. Although, in extremely simple cases the relationship between spatial features and spectum characteristics can be seen easily. However these frequency spectrum depictions are not themselves the typical goal of frequency analysis. Most commonly, Fourier transform serves as a computational exploit to impose new alterations to signals (in our case the images) cheaply.

In the same way we have learned that convolution of an image can apply some filter to, for example, blur or sharpen it, we actually apply one or several filters more easily and quickly by incorporating the extra steps of fast Fourier transform and inverse transform. Rather than applying traditional convolution to an image, which is obviously costly, we can instead consruct a filter as a matrix of same dimension (rather than a small 3 x 3 window) to the original image and perform simple elementwise multiplication across our filter and the complex, fourier transformed matrix. The savings becomes especially pronounced for increasing image sizes.

Filter results, for individual or successive filters, are also interpretable as simple mathematical functions. This can be seen with the included 'low_pass_filter' and 'high_pass_filter' functions above. Consider the examples below. We can easily effect an unlimited (hopefully advantageous) variety of image transformations to use in conjunction to learning methods. There is also the

application of frequency domain representations themselves along with an embedding step in deep learning methods in order to take advantage of the two representations directly as an alternative to traditional convolution layers.

[7]:
```
filt_l = low_pass_filter(data_batch_1[b'data'][:10].reshape(10, 3, 32, 32),␣
 ↪width=1)
filt_h = high_pass_filter(data_batch_1[b'data'][:10].reshape(10, 3, 32, 32),␣
 ↪width=.35)

imageplot(orig_sample)
imageplot(zero_one(np.real(np.fft.ifft2(np.fft.fftshift(samp)))))
imageplot(zero_one(filt_l))
imageplot(zero_one(filt_h))
```



[8]:
```
obs = len(data_batch_1[b'data']) * 5
data = np.concatenate((data_batch_1[b'data'],
                       data_batch_2[b'data'],
                       data_batch_3[b'data'],
                       data_batch_4[b'data'],
                       data_batch_5[b'data']), axis=0).reshape(obs, 3, 32, 32)
labels = to_categorical(np.concatenate((data_batch_1[b'labels'],
                                        data_batch_2[b'labels'],
```

```
                                        data_batch_3[b'labels'],
                                        data_batch_4[b'labels'],
                                        data_batch_5[b'labels'])),␣
 ↪num_classes=10)
data = data.astype('float32')

test_data = test_batch[b'data'].reshape(len(test_batch[b'data']), 3, 32, 32).
 ↪astype('float32')
test_labels = to_categorical(test_batch[b'labels'])

# Kernel gets broken by doing this all at once. Must be parted out.
twentieth = len(data) // 20
for i in range(20):
    lidx = i*twentieth
    hidx = (i+1)*twentieth if (i+1) < 20 else len(data)
    data[lidx:hidx] = data[lidx:hidx] / 255.0
fourth = len(test_data) // 4
for i in range(4):
    lidx = i*fourth
    hidx = (i+1)*fourth if (i+1) < 4 else len(data)
    test_data[lidx:hidx] = test_data[lidx:hidx] / 255.0

data = np.moveaxis(data, 1, -1)
test_data = np.moveaxis(test_data, 1, -1)

np.random.seed(SEED)
train_labels, train_data = shuffle(labels, data)
```

To move things forward, we should discuss the learning methods considered with the project. These are the convolutional neural network and vision transformer. Both methods are developed for learning goals like image classification, but their designs and backgrounds are extremely different.

0.1.8 Convolutional Neural Networks

Deep learning methods should not be described as "simple". However, we can argue that convolutional networks are at least comparitively straight-forward. The basic structure is a sequence of convolution followed by pooling for some depth of layers until the input size is effectively reduced. After this process, the resulting values are flattened into one-dimensional arrays and pushed to a feed forward network for classification. This can include various approaches to normalize data mid-process and/or regularize through dropout layers. From the first convolutional layer it is necessary for the model to use some predetermined amount of filters resulting in the subsequent shape of the data, and that the number of applied filters for subsequent layers will grow larger. Rather than explode the overall number of values, pooling (typically taking the maximum from some window size across the data) is used to shrink the amount of values being worked with until flattening occurs.

CNNs are the more traditional choice with image data. Although convolution is expensive, these models train quite quickly and also benefit from a fairly modest variety in terms of their architectural

choices. This greatly benefits the hyperparameter searching process, which will be explored further. For now, to refer to model design choices the project uses terminology found from Tensorflow. This means that "model hyperparameters" will refer to architectual considerations like the number and width of hidden layers and "algorithm hyperparameters" will refer to elements which control for speed and quality of learning, like the learning rate or optimizer choice. Either way, these are all considered "hyperparameters".

0.1.9 Vision Transformer

Transformers are a modern, deep learning architecture which was originally developed for Natural Language Processing problems, but has proven extremely versatile. Consequently, a form of transformer has been developed to perform image classification tasks specifically, which resembles the transformer's original language processing form quite closely.

Transformers are built on a few key components: positional encoding, self-attention, and feed-forward layers. In the original transformers - used for text - input sequences are "tokenized" into smaller units (words can become subwords) and converted into numeric representations combined with "positional embeddings". A positional embedding is a numeric representation added to tokens to encode information about their position or order in the sequence. Since transformers process input as a whole without a built-in understanding of sequence, positional embeddings provide a way to capture the arrangement of tokens. These positionally encoded tokens are then processed by the "self-attention" mechanism, which allows the model to focus on different parts of the input sequence depending on context. To achieve this, skip layers are used with "multihead attention" to improve network performance by addressing issues like vanishing gradients (the natural consequence of extreme depth in a network) and ensuring smoother gradient flow during training. In transformers, a skip connection adds the original input of a layer to its output before passing it to the next layer. For multihead attention, this works as follows: the input to the attention mechanism (a sequence of token embeddings) is processed by the multihead attention layer, which computes context-aware representations. Before these processed outputs move to the next stage (a final feed-forward layer), they are added element-wise to the original input embeddings. This combined result is then normalized using layer normalization. This forms a cycle, which can be extended for some predefined number of iterations, growing the transformer depth significantly.

Vision Transformers (ViTs) adapt this architecture for images by replacing the traditional word tokenization step with a system of creating "patches" from the input image. Instead of tokenizing text, an image is divided into smaller, fixed-size patches (e.g. 16x16 pixels). Each patch is flattened into a vector and treated like a token. These "patch embeddings" are combined with positional encodings to retain spatial information about where the patches are located in the original image. The resulting sequence of patch embeddings is then processed by the transformer's attention and feed-forward phases, enabling the model to analyze the relationships between different parts of the image. This patch-based approach allows transformers to effectively handle images while leveraging the same core principles used for text processing.

[9]:
```
def build_cnn(optimizer, loss, metrics, input_shape, filt_values, hl_units,␣
↪dropout_rate,
              kernel_sizes, activations, padding, regularizer, pool_size,␣
↪pool_stride):
    cnn = Sequential(name="CNN")
    for i in range(len(filt_values)):
```

```
        if i == 0:
            cnn.add(Conv2D(filters=filt_values[i],
                           kernel_size=kernel_sizes[i],
                           activation=activations[i],
                           padding=padding,
                           input_shape=input_shape))
            cnn.add(MaxPool2D(pool_size=pool_size, strides=pool_stride))

        else:
            cnn.add(Conv2D(filters=filt_values[i],
                           kernel_size=kernel_sizes[i],
                           activation=activations[i],
                           padding=padding))
            cnn.add(MaxPool2D(pool_size=pool_size, strides=pool_stride))

    cnn.add(Flatten())
    cnn.add(Dense(units=hl_units, activation=activations[-2]))
    cnn.add(Dropout(dropout_rate))
    cnn.add(Dense(units=10, activation=activations[-1]))

    cnn.compile(optimizer=optimizer, loss=loss, metrics=metrics)
    return cnn


def tune_cnn(hp):
    input_shape = (32,32,3)
    conv_layers = hp.Int('layers', min_value=1, max_value=3, step=1)
    filt_values = []
    for i in range(conv_layers): # Grow units using 2**i to conrol step␣
 ↪explosion
        units = hp.Int(f"units_layer_{i}", min_value=(32 * 2**i),␣
 ↪max_value=(256 * 2**i), step=(32 * 2**i))
        filt_values.append(units)

    kernel_sizes = [3,3,3]
    act = hp.Choice("activation", ["relu", "elu"])
    activations = [act]*(conv_layers+1) + ['softmax']

    hl_units = hp.Int("clsf_hl_units", min_value=512, max_value=2048,␣
 ↪default=1024, step=512)

    dropout_rate = hp.Float("clsf_dropout_rate", min_value=0.0, max_value=0.4,␣
 ↪step=0.1)

    optimizer_choice = hp.Choice('optimizer', values=['adam', 'sgd'])
```

```
    learning_rate = hp.Float("lrate", min_value=1e-5, max_value=1e-3,␣
 ↪default=1e-4, sampling="log")

    if optimizer_choice == 'adam':
        optimizer = Adam(learning_rate=learning_rate)
    elif optimizer_choice == 'sgd':
        optimizer = SGD(learning_rate=learning_rate)

    loss = 'categorical_crossentropy'
    metrics = ['accuracy']

    model = build_cnn(optimizer, loss, metrics, input_shape, filt_values,␣
 ↪hl_units, dropout_rate,
                      kernel_sizes, activations, padding='same',␣
 ↪regularizer=None, pool_size=2, pool_stride=2)
    return model


tuner = keras_tuner.Hyperband(tune_cnn, objective='val_accuracy',␣
 ↪max_epochs=30, factor=3,
                     hyperband_iterations=1, directory='cnn_dir1',␣
 ↪project_name='cnn')
```

```
Reloading Tuner from cnn_dir1\cnn\tuner0.json
```

0.1.10 Hyperband

Common methods for hyperparameter search include grid search and random searching. Where fitting a model can be performed very quickly, these naive approaches can scour the optimization surface associated with the range of possible choices for hyperparameters. However, neural networks can require very many epochs of fitting the training data, with each epoch tending to require much more time to fit than simpler classifiers, such as decision trees. Because of longer optimization times and the significant added complexity of having many additional points for exploration from model hyperparameters, more intelligent methods of polling a hyperparameter surface have been developed. This includes the Hyperband algorithm, which was selected for the project and is descibed briefly.

To employ the algorithm of the same name we use the Hyperband tuner from keras tuner package. Based on random search, the tuner randomly polls the provided range of choices for some number of combinations, but uses tournament style elimination to prevent the need for fitting every one for the full amount of epochs. The initial round begins with only a small budget of epochs per "trial", before elmiminating the worse performing half, then a subsequent round, and so on. This is far more efficient, and can be exploited to poll an even wider range of hyperparameter combinations, or otherwise save on resources. Similar to the keras fit method for models, the tuner allows for validation split and testing, data shuffling, and (perhaps most importantly) early stopping.

[63]:
```
early_stop = EarlyStopping(monitor='val_loss', patience=5)
```

```
tuner.search(train_data, train_labels, epochs=50, validation_split=0.1,␣
 ↪callbacks=[early_stop])

best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]
```

```
Trial 90 Complete [00h 07m 46s]
val_accuracy: 0.3346000015735626

Best val_accuracy So Far: 0.7654000520706177
Total elapsed time: 01h 52m 01s
```

[68]:
```
hypermodel = tuner.hypermodel.build(best_hps)

early_stop_vacc = EarlyStopping(monitor='val_accuracy', patience=15,␣
 ↪restore_best_weights=True)

history = hypermodel.fit(train_data, train_labels, batch_size=16, epochs=50,␣
 ↪verbose=0,
                         validation_split=0.1, callbacks=[early_stop_vacc])
```

[71]:
```
print("Best Hyperparameters:")
for param, value in best_hps.values.items():
    print(f"{param}: {value}")
```

```
Best Hyperparameters:
layers: 2
units_layer_0: 256
activation: relu
clsf_hl_units: 2048
clsf_dropout_rate: 0.4
optimizer: adam
lrate: 0.00015852467948888102
units_layer_1: 512
units_layer_2: 384
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 3
tuner/round: 2
tuner/trial_id: 0037
```

0.1.11 CNN Hyperparameter Search

For each iteration of the Hyperband algorithm ninety "trials" are performed. For one of these iterations we explore a wide variety of options. This includes a couple choices of optimizer, a couple of activation function choices, many orientations of shape for the convolution phase, and various sizes of hidden layer and dropout rates for the classification phase. The fitting process is demanding of machine hardware, but is able to complete fairly fast. Depending on randomization, hyperband can complete an iteration in an hour or two.

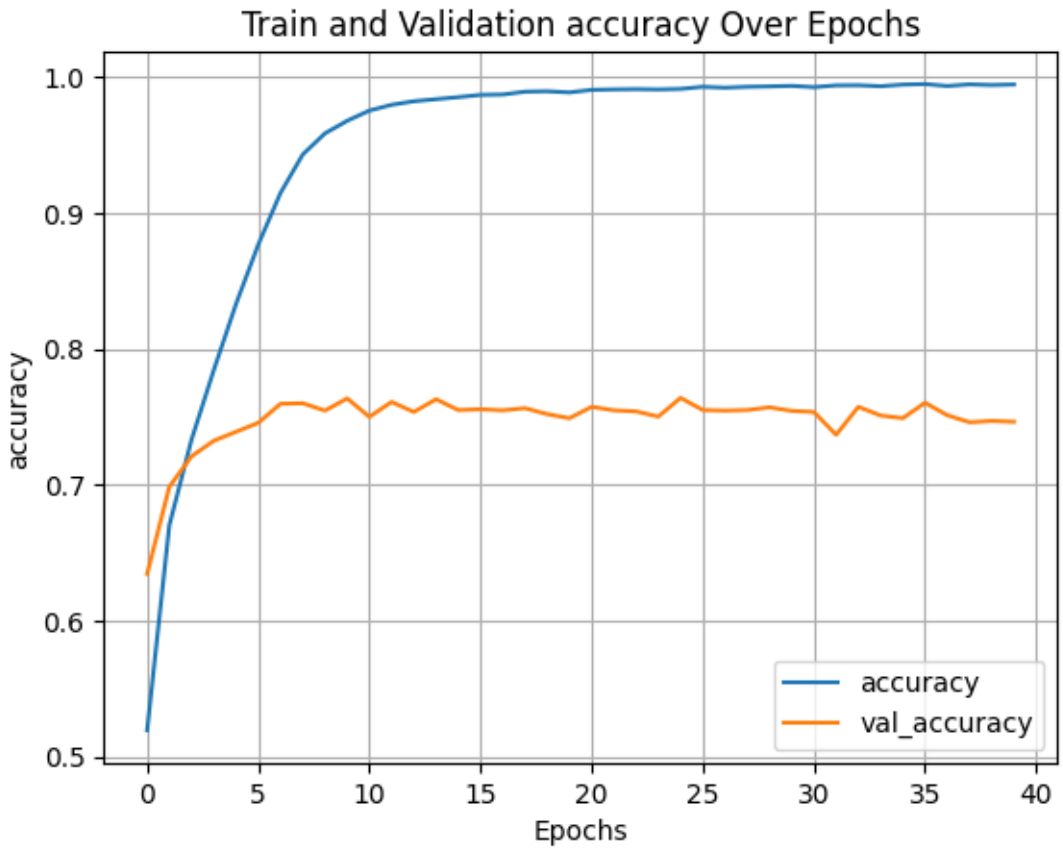
In this case, the best performer resulting from search performs three stages of convolution (units_layer_0: 256, units_layer_1: 512, units_layer_2: 384), and has as large of a hidden classifier layer as is possible from the allowed range. Looking to the performance charts below, training performance behaves exactly as we would hope and tops out in accuracy quickly. Results of validation loss are very telling though. For this combination of hyperparameters the learning rate happens to be extremely high.

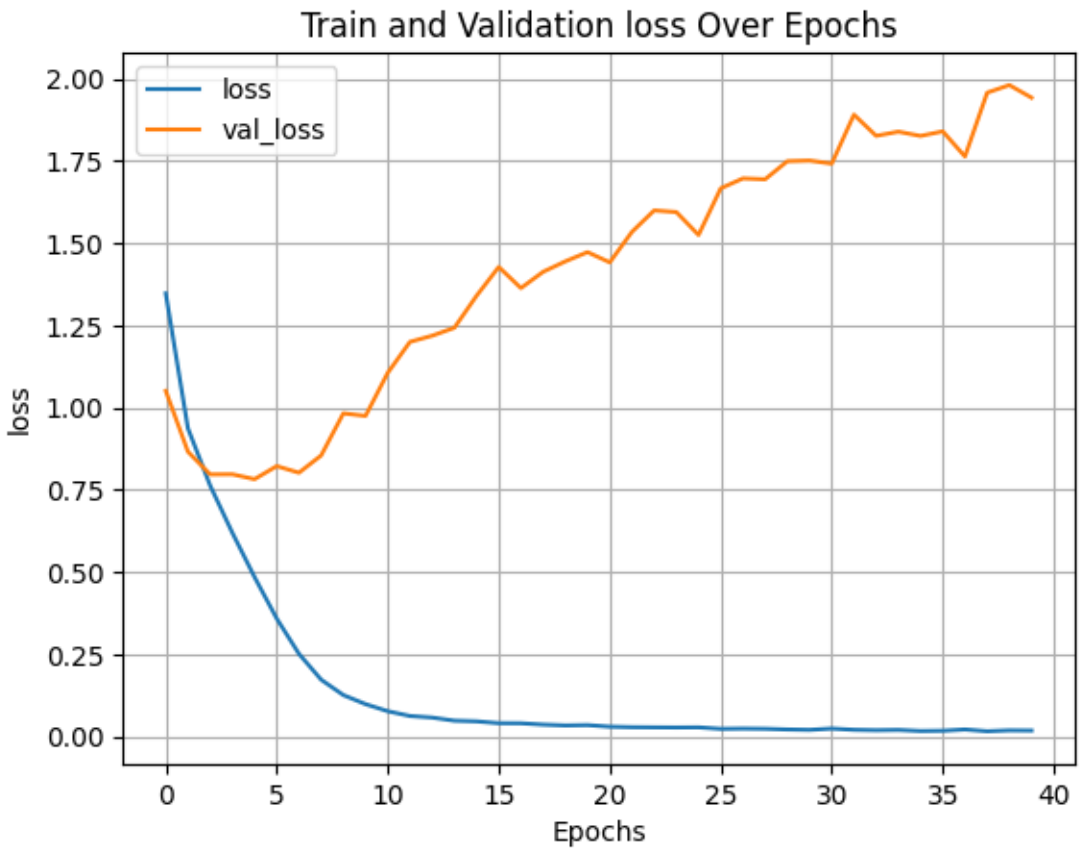
Where we take the time to check how the model performs with testing data, the result looks fairly good. Numerically, test accuracy also corresponds closely with what was seen with validation accuracy when we use early stopping to restore model weights automatically through keras tuner callbacks. The confusion matrix shows that the model generalizes best with certain vehicle images and gets confused the most between cats and dogs.

[167]:
```
def historyplot(history, item):
    plt.plot(history.history[item], label=item)
    plt.plot(history.history["val_" + item], label="val_" + item)
    plt.xlabel("Epochs")
    plt.ylabel(item)
    plt.title(f"Train and Validation {item} Over Epochs")
    plt.legend()
    plt.grid()
    plt.show()

def resultplot(model, history):
    val_acc_per_epoch = history.history['val_accuracy']

    best_epoch = val_acc_per_epoch.index(max(val_acc_per_epoch)) + 1
    print(model.summary())
    print(f'Best epoch: {best_epoch}')

    historyplot(history, 'accuracy')
    historyplot(history, 'loss')

resultplot(hypermodel, history)
```

```
Model: "CNN"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 32, 32, 256)       7168

 max_pooling2d_3 (MaxPooling  (None, 16, 16, 256)      0
 2D)

 conv2d_4 (Conv2D)           (None, 16, 16, 512)       1180160

 max_pooling2d_4 (MaxPooling  (None, 8, 8, 512)        0
 2D)
```

```
 flatten_2 (Flatten)         (None, 32768)             0

 dense_4 (Dense)             (None, 2048)              67110912

 dropout_2 (Dropout)         (None, 2048)              0

 dense_5 (Dense)             (None, 10)                20490

=================================================================
Total params: 68,318,730
Trainable params: 68,318,730
Non-trainable params: 0
_________________________________________________________________
None
Best epoch: 25
```

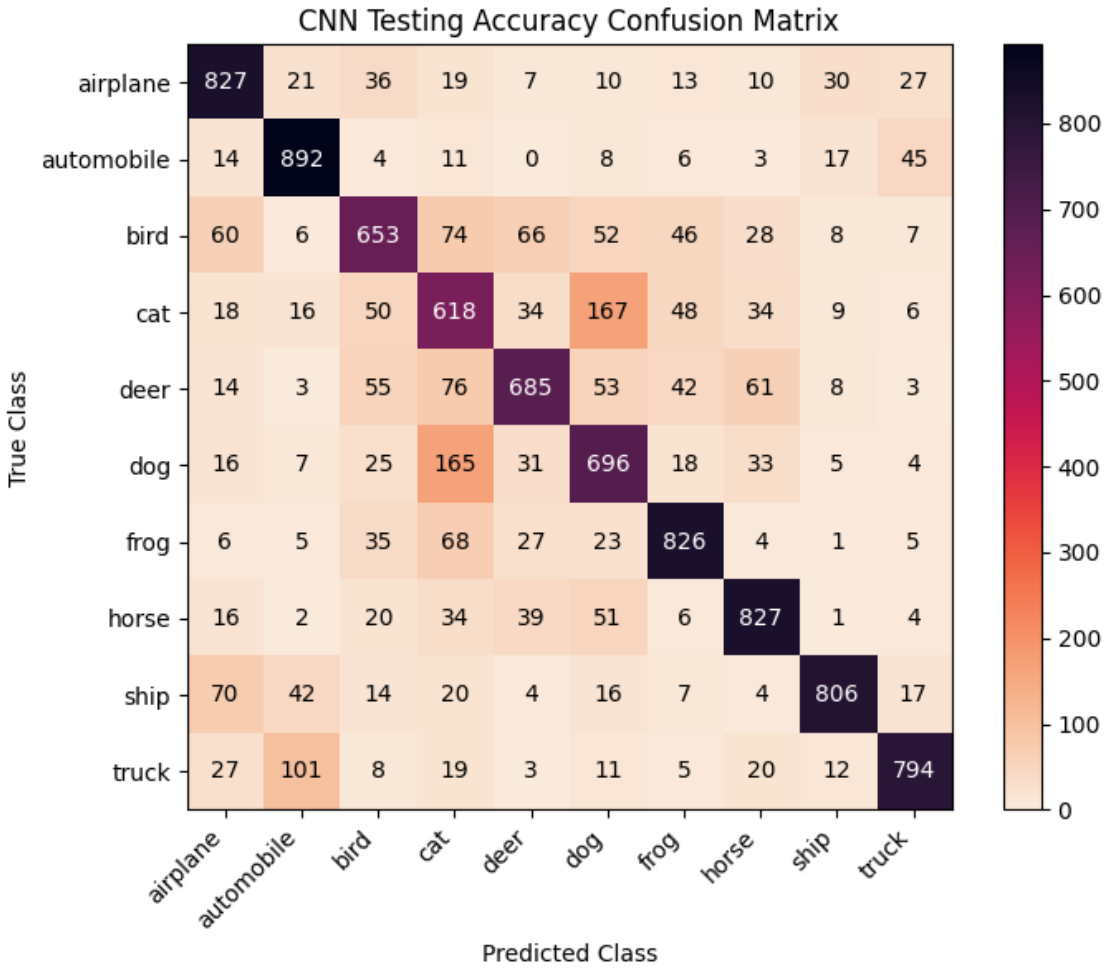




[183]:
```
def testing_accuracy_matrix(model, te_data):
    pred_labels = np.argmax(model.predict(te_data), axis=1).astype('int32')
    true_labels = np.array(test_batch[b'labels'])
    class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',␣
 ↪'frog', 'horse', 'ship', 'truck']
    cm = confusion_matrix(y_true=true_labels, y_pred=pred_labels)
    fig, ax = plt.subplots(figsize=(8, 6))
    #plt.figure(figsize=(10,8))
    im = ax.imshow(cm, cmap="rocket_r")

    # Add a colorbar
    cbar = ax.figure.colorbar(im, ax=ax)
    # Annotate each cell
    rows, cols = cm.shape
    for i in range(rows):
        for j in range(cols):
            if i == j:
                ax.text(j, i, f"{cm[i, j]}", ha="center", va="center",␣
 ↪color="white")
            else:
```

```
                ax.text(j, i, f"{cm[i, j]}", ha="center", va="center",␣
 ↪color="black")

    # Add labels and title
    ax.set_xlabel("Predicted Class")
    ax.set_ylabel("True Class")
    ax.set_title(f"{model.name} Testing Accuracy Confusion Matrix")
    plt.xticks(range(cols), labels=class_names, rotation=45, ha='right')
    plt.yticks(range(rows), labels=class_names)

    plt.show()

def testperf(model, te_data, te_labels):
    eval_result = model.evaluate(te_data, te_labels)
    print("[test loss, test accuracy]:", eval_result)
    testing_accuracy_matrix(model, te_data)

testperf(hypermodel, test_data, test_labels)
```

```
313/313 [==============================] - 11s 34ms/step - loss: 1.5152 -
accuracy: 0.7624
[test loss, test accuracy]: [1.5152196884155273, 0.7624000310897827]
313/313 [==============================] - 10s 31ms/step
```



[15]:
```
class Patches(Layer):
    def __init__(self, patch_size):
        super().__init__()
        self.patch_size = patch_size

    def call(self, images):
        input_shape = tf.shape(images)
        batch_size = input_shape[0]
        height = input_shape[1]
        width = input_shape[2]
        channels = input_shape[3]
        num_patches_h = height // self.patch_size
        num_patches_w = width // self.patch_size
        patches = tf.image.extract_patches(images,
                                           sizes=[1, self.patch_size, self.
 ↪patch_size, 1],
```

```
                                        strides=[1, self.patch_size, self.
↪patch_size, 1],
                                        rates=[1, 1, 1, 1],
                                        padding='VALID')
        patches = tf.reshape(
            patches,
            (
                batch_size,
                num_patches_h * num_patches_w,
                self.patch_size * self.patch_size * channels,
            ),
        )
        return patches

    def get_config(self):
        config = super().get_config()
        config.update({"patch_size": self.patch_size})
        return config


def build_vit(optimizer, loss, metrics, input_shape, num_classes, patch_size,␣
↪num_patches, projection_dim,
              transformer_dropout, transformer_units, num_heads,␣
↪transformer_layers):
    inputs = keras.Input(shape=input_shape)

    patches = Patches(patch_size)(inputs)

    positions = tf.range(start=0, limit=num_patches, delta=1)
    projection = Dense(units=projection_dim)(patches)
    position_embedding = Embedding(input_dim=num_patches,␣
↪output_dim=projection_dim)(positions)
    encoded_patches = projection + position_embedding

    # Transformer blocks
    for _ in range(transformer_layers):
        # Multi-head attention
        x1 = LayerNormalization(epsilon=1e-6)(encoded_patches)
        attention_output = MultiHeadAttention(num_heads=num_heads,␣
↪key_dim=projection_dim)(x1, x1)
        x2 = Add()([attention_output, encoded_patches])

        # Multilayer perceptron block
        mlp = LayerNormalization(epsilon=1e-6)(x2)
        for units in transformer_units:
            mlp = Dense(units, activation="gelu")(mlp)
            mlp = Dropout(transformer_dropout)(mlp)
```

```
        encoded_patches = Add()([mlp, x2])

    # Classification head
    representation = LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = Flatten()(representation)
    representation = Dropout(0.5)(representation)

    mlp = representation
    for units in transformer_units:
        mlp = Dense(units, activation="gelu")(mlp)
        mlp = Dropout(0.5)(mlp)

    outputs = Dense(num_classes, activation="softmax")(mlp)

    vit = keras.Model(name="VisionTransformer", inputs=inputs, outputs=outputs)
    vit.compile(optimizer=optimizer, loss=loss, metrics=metrics)
    return vit


def tune_vit(hp):
    input_shape = (32,32,3)
    num_classes = 10

    patch_size = hp.Int("patch_size", min_value=8, max_value=16, step=2)
    num_patches = (input_shape[0] // patch_size) ** 2

    projection_dim = hp.Int("projection_dim", min_value=64, max_value=128,␣
 ↪step=64)

    transformer_dropout = hp.Float("transf_mlp_drop_rt", min_value=0.1,␣
 ↪max_value=0.4, step=0.1)

    mlp_layers = hp.Int("mlp_layers", min_value=1, max_value=3, step=1)

    transformer_units = [(projection_dim * 2**i) for i in␣
 ↪range(mlp_layers-1,-1,-1)]

    num_heads = hp.Int("num_heads", min_value=4, max_value=8, step=4)

    transformer_layers = hp.Int("transformer_layers", min_value=2, max_value=6,␣
 ↪step=2)

    learning_rate = hp.Float("lrate", min_value=1e-5, max_value=1e-3,␣
 ↪default=1e-4, sampling="log")
```

```
    momentum = hp.Float('momentum', min_value=0.0, max_value=0.9, step=0.3,␣
 ↪default=0.9)

    optimizer_choice = hp.Choice('optimizer', values=['adam',
                                                      'sgd',
                                                      'rmsprop'
                                                     ])

    if optimizer_choice == 'adam':
        optimizer = Adam(learning_rate=learning_rate)
    elif optimizer_choice == 'sgd':
        optimizer = SGD(learning_rate=learning_rate, momentum=momentum)
    elif optimizer_choice == 'rmsprop':
        optimizer = SGD(learning_rate=learning_rate, momentum=momentum)

    loss = 'categorical_crossentropy'
    metrics = ['accuracy']

    model = build_vit(optimizer, loss, metrics, input_shape, num_classes,␣
 ↪patch_size, num_patches,
                      projection_dim, transformer_dropout, transformer_units,␣
 ↪num_heads, transformer_layers)

    return model


vtuner = keras_tuner.Hyperband(tune_vit, objective='val_accuracy',␣
 ↪max_epochs=30, factor=3,
                     hyperband_iterations=2, directory='vit_dir1',␣
 ↪project_name='vit')
```

```
Reloading Tuner from vit_dir1\vit\tuner0.json
```

[178]:
```
class LimitTrainingTime(tf.keras.callbacks.Callback):
    def __init__(self, max_epoch_time_s, max_step_time_s):
        super().__init__()
        self.max_time_s = max_epoch_time_s
        self.max_step_time_s = max_step_time_s
        self.start_time = None
        self.num_batches = 0

    def on_epoch_begin(self, epoch, logs=None):
        self.start_time = time.time()
        self.num_batches = 1

    def on_train_batch_end(self, batch, logs):
        time_taken = time.time() - self.start_time
```

```
        if (time_taken > self.max_time_s) or (
            self.num_batches > 40 and time_taken / self.num_batches > self.
 ↪max_step_time_s):
            self.model.stop_training = True
        self.num_batches += 1


#train_data_0255 = (train_data * 255).astype(np.uint8) # using original data␣
 ↪format while retaining shuffled order

go_fast = LimitTrainingTime(max_epoch_time_s=120, max_step_time_s=0.6)
vit_early_stop = EarlyStopping(monitor='val_loss', patience=5)
keep_growing = EarlyStopping(monitor='accuracy', min_delta=0.02, patience=5)

vtuner.search(train_data, train_labels, epochs=50, batch_size=512,␣
 ↪validation_split=0.1, callbacks=[vit_early_stop, go_fast, keep_growing])

vit_best_hps=vtuner.get_best_hyperparameters(num_trials=1)[0]
```

```
Trial 180 Complete [00h 01m 41s]
val_accuracy: 0.09300000220537186

Best val_accuracy So Far: 0.17080001533031464
Total elapsed time: 04h 30m 50s
```

[179]:
```
vit_hypermodel = vtuner.hypermodel.build(vit_best_hps)

vit_early_stop_vacc = EarlyStopping(monitor='val_accuracy', patience=15,␣
 ↪restore_best_weights=True)

vit_history = vit_hypermodel.fit(train_data, train_labels, batch_size=512,␣
 ↪epochs=100, verbose=0,
                          validation_split=0.1, callbacks=[vit_early_stop_vacc])
```

[180]:
```
print("Best Hyperparameters:")
for param, value in vit_best_hps.values.items():
    print(f"{param}: {value}")
```

```
Best Hyperparameters:
patch_size: 16
projection_dim: 128
transf_mlp_drop_rt: 0.2
mlp_layers: 2
num_heads: 4
transformer_layers: 4
lrate: 2.0221406050328464e-05
momentum: 0.6
```

```
optimizer: adam
tuner/epochs: 4
tuner/initial_epoch: 2
tuner/bracket: 3
tuner/round: 1
tuner/trial_id: 0019
```

0.1.12 ViT Hyperparameter Search

Once certain issues with searching the space of hyperparameters for vision transformers could be better worked out, the time taken for each iteration of hyperband was improved dramatically. This created the ability to perform a couple of iterations at a time, though this still required several hours to finish. To clarify, for the model and algorithm hyperparameters explored here and this representation of the data the hyperparameter surface which results is laden with combinations which go absolutely nowhere. In some of these cases, a combination (often but not always a large architecture) might take up to an hour for each epoch to complete. Regardless of the speed, in some cases training accuracy never takes off regardless of the number of epochs allowed; these models are stuck at "just guessing" performance levels, i.e. around 10% accuracy. Dealing with this required exploring keras tuner callbacks further. Not only is validation accuracy monitored as before, but where accuracy refuses to climb the tuner will ditch out on the model. Also, to deal with slow models both the speed at which training batches complete and the overall epoch time is tracked. Monitoring both characteristics is sufficient to cover when models are not reasonably fast enough and the most extreme cases of all (where training each batch is so slow that establishing a good batch speed baseline becomes impossible and the tuner wastes many minutes on a model which is totally stuck), respectively. Thankfully, for the huge increase in time required to fit vision transformers and search hypreparameters the demand on hardware considerably lighter. Running the machine at full-bore for the very many hours needed would not have been reasonable.

Several search iterations were performed in the time available, with the result of the most recent shown. In this instance, the best performer creates large patches (which would divide the image into quadrants in this case). Each feed-forward network per transformer layer (and also in the classifier layer) has depth two, with four transformer layers overal. The optimizer is Adam, with fairly low learning rate (this optimizer also means that momentum should be ignored in this case). The model then tops out at about one-and-a-half million weights. Training and validation performance have consistently looked good with these models. Behaviors in their trend-lines correspond well enough to indicate that the normalization layers and regularization from dropout used are working to prevent overfitting.

Looking to test performance, it is apparent that although the model is functioning above random guessing but performing poorly. Worse, interpetability of what to change is difficult here as there are so many moving parts involved. The model identifies ships fairly well, but is completely unable to distinguish cats and dogs; generally it confuses animals as a rule.

[181]:
```
resultplot(vit_hypermodel, vit_history)
```

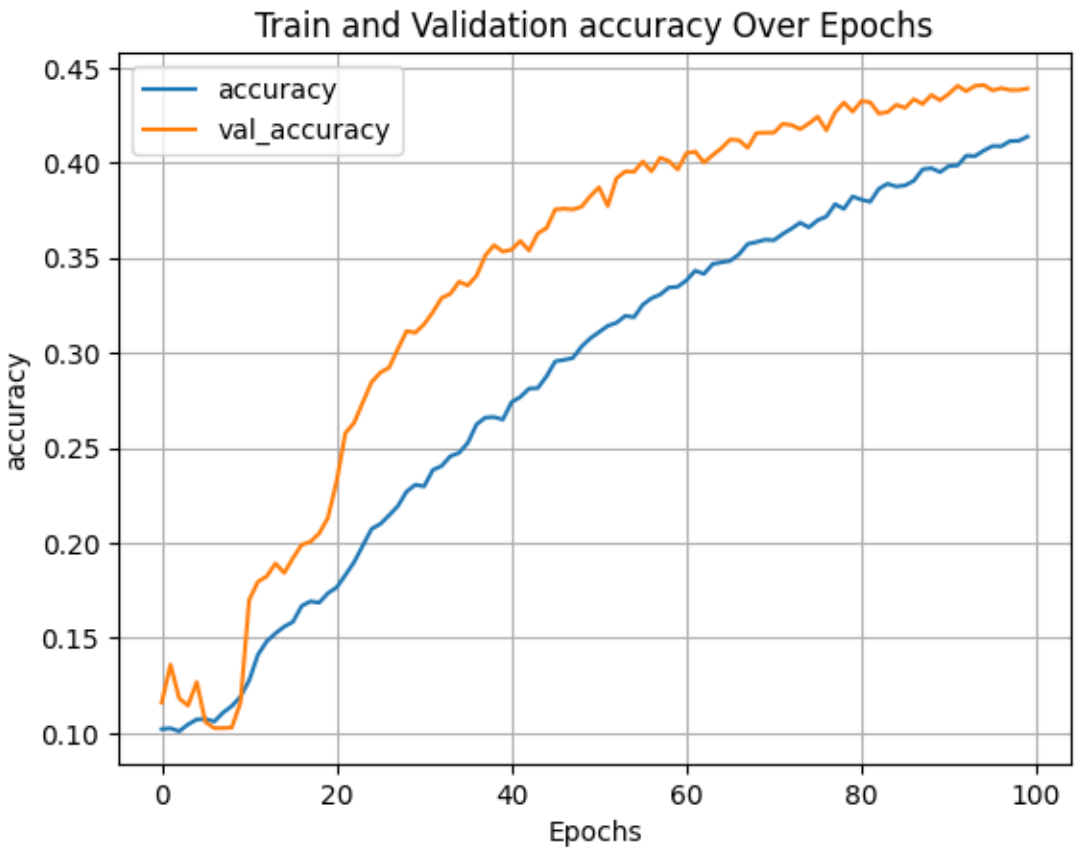
```
Model: "VisionTransformer"
__________________________________________________________________________________
__________________
 Layer (type)                 Output Shape              Param #     Connected to
```

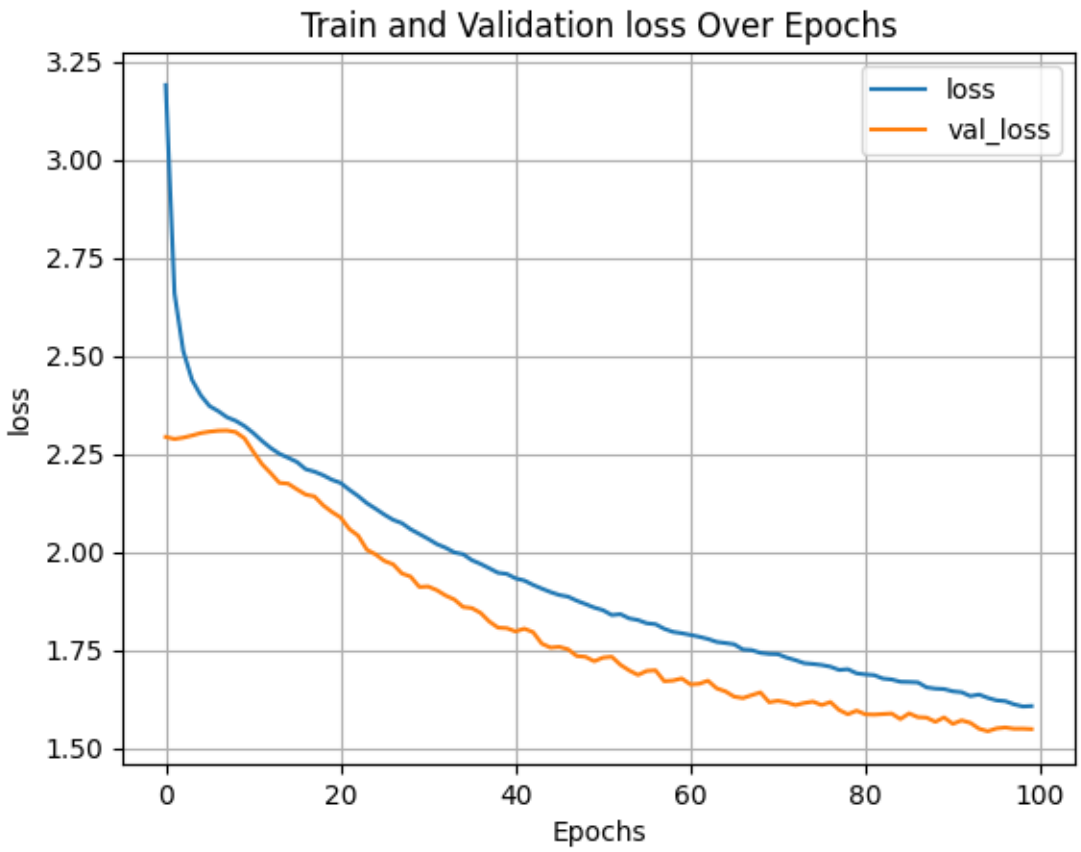
```
==================================================================================
==================
 input_2 (InputLayer)           [(None, 32, 32, 3)]  0           []

 patches_1 (Patches)            (None, 4, 768)       0
['input_2[0][0]']

 dense_12 (Dense)               (None, 4, 128)       98432
['patches_1[0][0]']

 tf.__operators__.add_1 (TFOpLa  (None, 4, 128)      0
['dense_12[0][0]']
 mbda)

 layer_normalization_9 (LayerNo  (None, 4, 128)      256
['tf.__operators__.add_1[0][0]']
 rmalization)

 multi_head_attention_4 (MultiH  (None, 4, 128)      263808
['layer_normalization_9[0][0]',
 eadAttention)
'layer_normalization_9[0][0]']

 add_8 (Add)                    (None, 4, 128)       0
['multi_head_attention_4[0][0]',
'tf.__operators__.add_1[0][0]']

 layer_normalization_10 (LayerN  (None, 4, 128)      256         ['add_8[0][0]']
 ormalization)

 dense_13 (Dense)               (None, 4, 256)       33024
['layer_normalization_10[0][0]']

 dropout_11 (Dropout)           (None, 4, 256)       0
['dense_13[0][0]']

 dense_14 (Dense)               (None, 4, 128)       32896
['dropout_11[0][0]']

 dropout_12 (Dropout)           (None, 4, 128)       0
['dense_14[0][0]']

 add_9 (Add)                    (None, 4, 128)       0
['dropout_12[0][0]',
                                                                  'add_8[0][0]']

 layer_normalization_11 (LayerN  (None, 4, 128)      256         ['add_9[0][0]']
 ormalization)
```

```
 multi_head_attention_5 (MultiH  (None, 4, 128)      263808
['layer_normalization_11[0][0]',
 eadAttention)
'layer_normalization_11[0][0]']

 add_10 (Add)                   (None, 4, 128)       0
['multi_head_attention_5[0][0]',
                                                                 'add_9[0][0]']

 layer_normalization_12 (LayerN  (None, 4, 128)      256
['add_10[0][0]']
 ormalization)

 dense_15 (Dense)               (None, 4, 256)       33024
['layer_normalization_12[0][0]']

 dropout_13 (Dropout)           (None, 4, 256)       0
['dense_15[0][0]']

 dense_16 (Dense)               (None, 4, 128)       32896
['dropout_13[0][0]']

 dropout_14 (Dropout)           (None, 4, 128)       0
['dense_16[0][0]']

 add_11 (Add)                   (None, 4, 128)       0
['dropout_14[0][0]',
'add_10[0][0]']

 layer_normalization_13 (LayerN  (None, 4, 128)      256
['add_11[0][0]']
 ormalization)

 multi_head_attention_6 (MultiH  (None, 4, 128)      263808
['layer_normalization_13[0][0]',
 eadAttention)
'layer_normalization_13[0][0]']

 add_12 (Add)                   (None, 4, 128)       0
['multi_head_attention_6[0][0]',
'add_11[0][0]']

 layer_normalization_14 (LayerN  (None, 4, 128)      256
['add_12[0][0]']
 ormalization)

 dense_17 (Dense)               (None, 4, 256)       33024
```

```
['layer_normalization_14[0][0]']

 dropout_15 (Dropout)           (None, 4, 256)       0
['dense_17[0][0]']

 dense_18 (Dense)               (None, 4, 128)       32896
['dropout_15[0][0]']

 dropout_16 (Dropout)           (None, 4, 128)       0
['dense_18[0][0]']

 add_13 (Add)                   (None, 4, 128)       0
['dropout_16[0][0]',
'add_12[0][0]']

 layer_normalization_15 (LayerN  (None, 4, 128)      256
['add_13[0][0]']
 ormalization)

 multi_head_attention_7 (MultiH  (None, 4, 128)      263808
['layer_normalization_15[0][0]',
 eadAttention)
'layer_normalization_15[0][0]']

 add_14 (Add)                   (None, 4, 128)       0
['multi_head_attention_7[0][0]',
'add_13[0][0]']

 layer_normalization_16 (LayerN  (None, 4, 128)      256
['add_14[0][0]']
 ormalization)

 dense_19 (Dense)               (None, 4, 256)       33024
['layer_normalization_16[0][0]']

 dropout_17 (Dropout)           (None, 4, 256)       0
['dense_19[0][0]']

 dense_20 (Dense)               (None, 4, 128)       32896
['dropout_17[0][0]']

 dropout_18 (Dropout)           (None, 4, 128)       0
['dense_20[0][0]']

 add_15 (Add)                   (None, 4, 128)       0
['dropout_18[0][0]',
'add_14[0][0]']
```

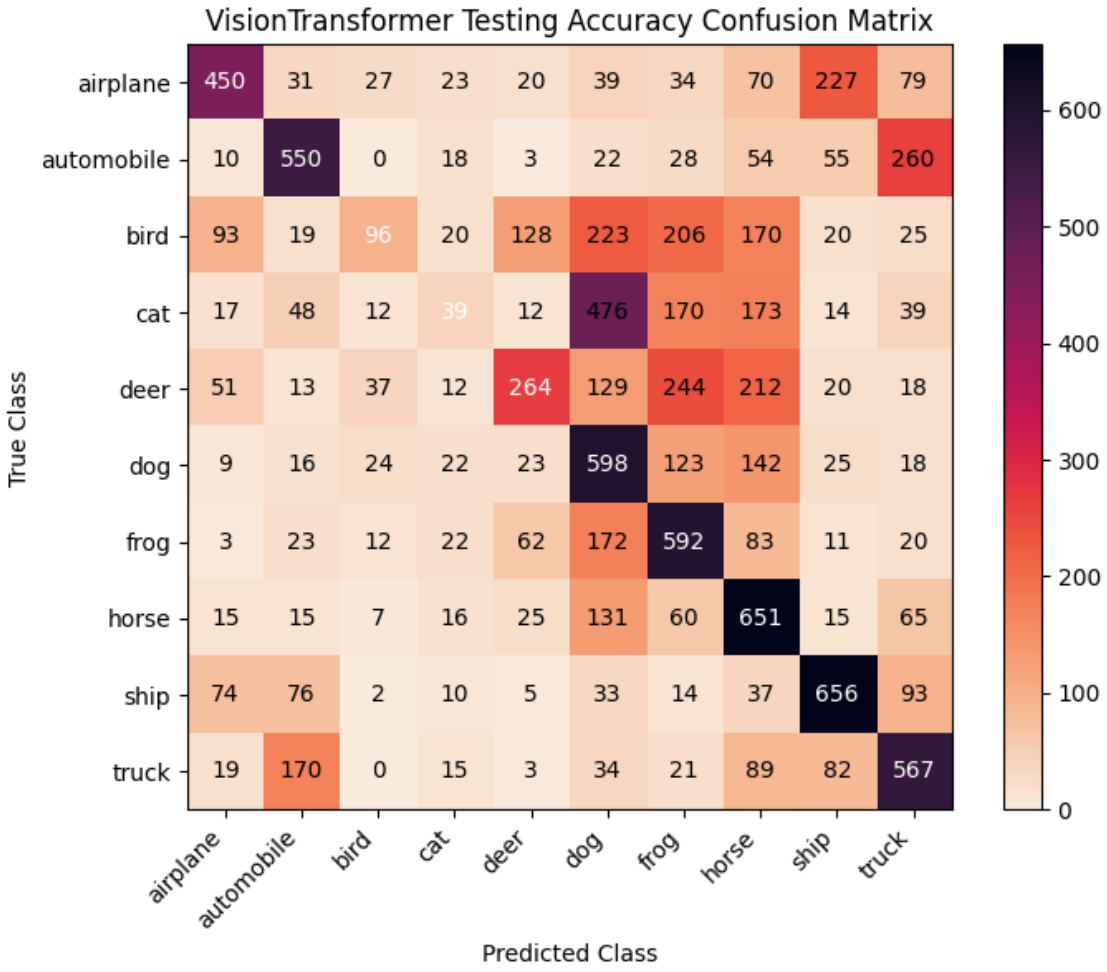
```
 layer_normalization_17 (LayerN  (None, 4, 128)      256
['add_15[0][0]']
 ormalization)

 flatten_1 (Flatten)            (None, 512)          0
['layer_normalization_17[0][0]']

 dropout_19 (Dropout)           (None, 512)          0
['flatten_1[0][0]']

 dense_21 (Dense)               (None, 256)          131328
['dropout_19[0][0]']

 dropout_20 (Dropout)           (None, 256)          0
['dense_21[0][0]']

 dense_22 (Dense)               (None, 128)          32896
['dropout_20[0][0]']

 dropout_21 (Dropout)           (None, 128)          0
['dense_22[0][0]']

 dense_23 (Dense)               (None, 10)           1290
['dropout_21[0][0]']

==================================================================================
==================
Total params: 1,585,162
Trainable params: 1,585,162
Non-trainable params: 0
----------------------------------------------------------------------------------
------------------
None
Best epoch: 95
```


Train and Validation accuracy Over Epochs
accuracy
val_accuracy
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
accuracy
0
20
40
60
80
100
Epochs



[185]:
```
testperf(vit_hypermodel, test_data, test_labels)
```

```
313/313 [==============================] - 90s 288ms/step - loss: 1.5364 -
accuracy: 0.4463
[test loss, test accuracy]: [1.53644859790802, 0.44630002975463867]
313/313 [==============================] - 96s 303ms/step
```



[ ]:

0.1.13 Performance Improvements

Where learning methods are simple enough to search hyperparameters comprehensively, we potentially do not need to impose further granularity on searching or manual hyperparameter tweeks. For methods like neural networks, further tuning can still be worthwhile because hyperband as a search method is designed to explore more efficiently than precisely. Assuming the dimensionality or cardinality of hyperparameter choices is high, the best-performing configurations identified by hyperband are unlikely to have been optimized fully. Additional tuning can refine these configurations by focusing more computational resources on fine-tuning key hyperparameters, exploring their local neighborhood more thoroughly, or using additional search strategy.

[10]:
```
optimizer = Adam(learning_rate=.00006)
loss = 'categorical_crossentropy'
metrics = ['accuracy']
input_shape = (32,32,3)
```

```
filt_values = [256,512,348]
hl_units = 2048
dropout_rate = 0.4
kernel_sizes = [3,3,3]
activations = ['relu','relu','relu','softmax']

cnn_test_model = build_cnn(optimizer, loss, metrics, input_shape, filt_values,␣
 ↪hl_units, dropout_rate,
                      kernel_sizes, activations, padding='same',␣
 ↪regularizer=None, pool_size=2, pool_stride=2)

early_stop_vacc = EarlyStopping(monitor='val_accuracy', patience=15,␣
 ↪restore_best_weights=True)

cnn_test_history = cnn_test_model.fit(train_data, train_labels, batch_size=16,␣
 ↪epochs=20, verbose=0,
                          validation_split=0.1, callbacks=[early_stop_vacc])
```

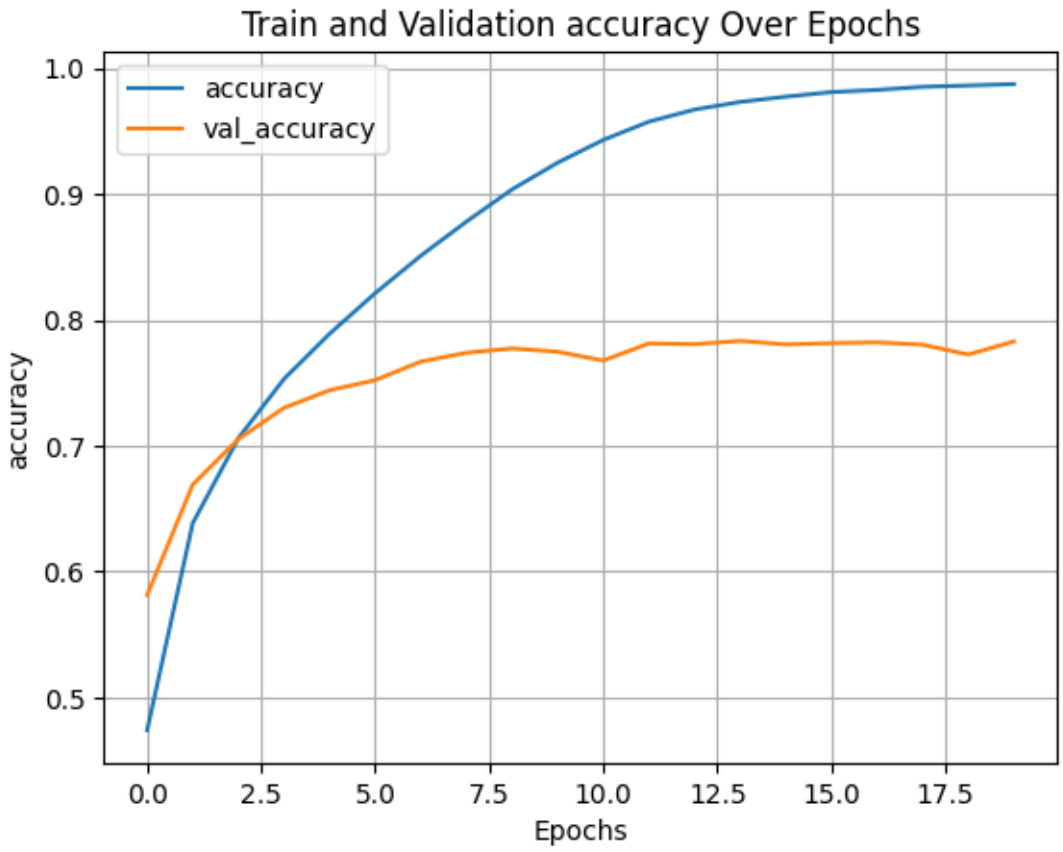
0.1.14 CNN Test Model

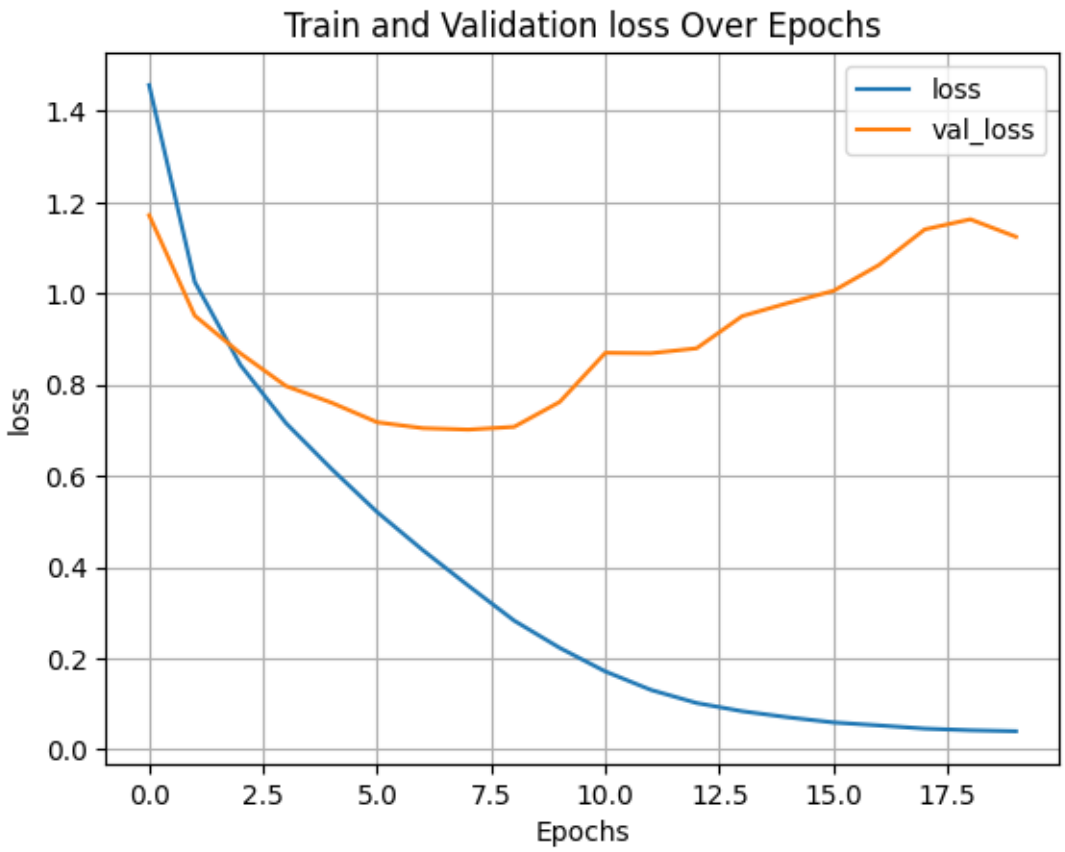
Using the "Best Hyperparameters" printout from earlier, we can build the model again after the fact but take the opportunity to manually tune a bit further. We identified fom validation loss that the learning rate for the model was too high. After a few iterations of fitting with lower learning rate, a better choice of hyperparameter was found. The validation loss curve is less extreme and the testing performance of this model is at least marginally better. Where we look to the test data confusion matrix below, class confusion in the model has become less pronounced.

[13]:
```
resultplot(cnn_test_model, cnn_test_history)
```

```
Model: "CNN"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 256)       7168

 max_pooling2d (MaxPooling2D  (None, 16, 16, 256)      0
 )

 conv2d_1 (Conv2D)           (None, 16, 16, 512)       1180160

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 512)        0
 2D)

 conv2d_2 (Conv2D)           (None, 8, 8, 348)         1603932

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 348)        0
 2D)
```

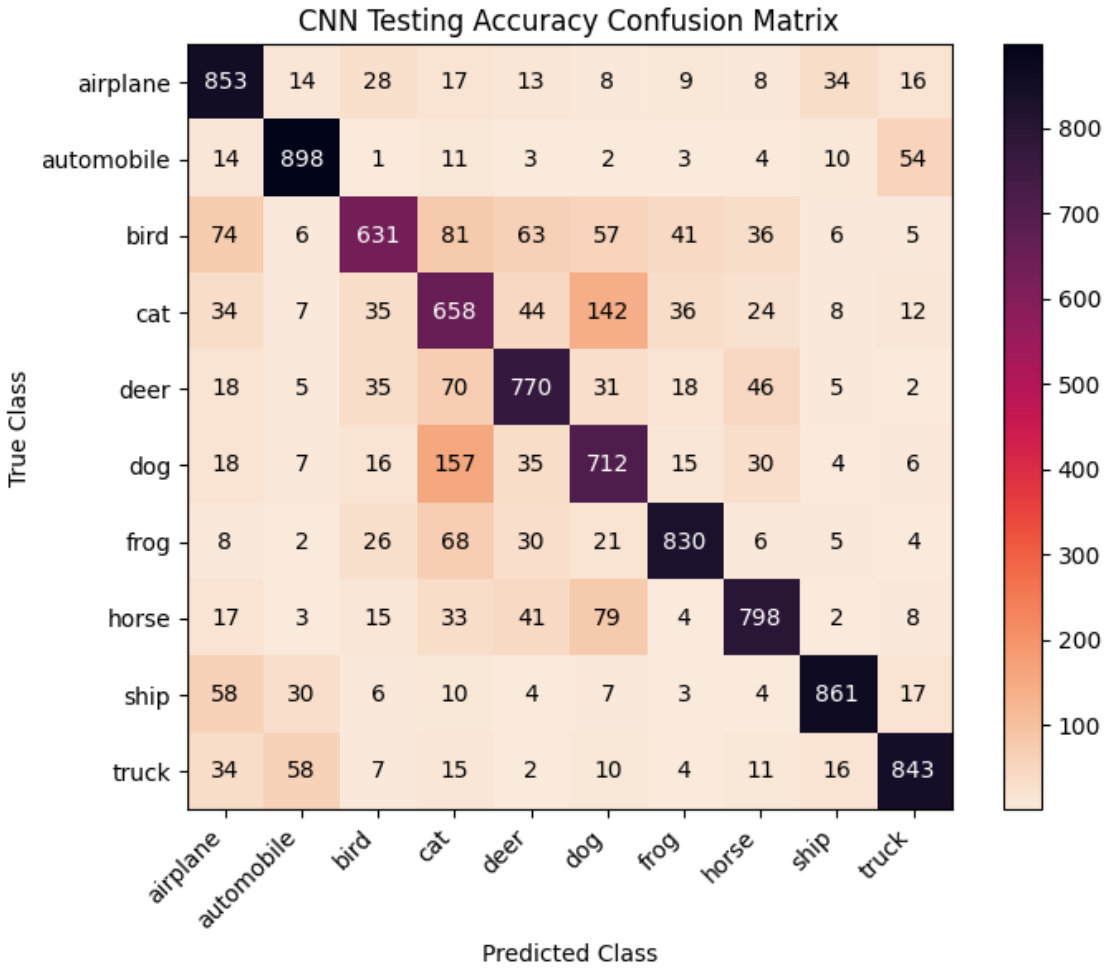
```
 flatten (Flatten)           (None, 5568)              0

 dense (Dense)               (None, 2048)              11405312

 dropout (Dropout)           (None, 2048)              0

 dense_1 (Dense)             (None, 10)                20490

=================================================================
Total params: 14,217,062
Trainable params: 14,217,062
Non-trainable params: 0
_________________________________________________________________
None
Best epoch: 14
```





[14]:
```
testperf(cnn_test_model, test_data, test_labels)
```

```
313/313 [==============================] - 1s 4ms/step - loss: 1.0946 -
accuracy: 0.7854
[test loss, test accuracy]: [1.0945510864257812, 0.7854000329971313]
313/313 [==============================] - 1s 3ms/step
```



[ ]:

[18]:
```
vtuner3 = keras_tuner.Hyperband(tune_vit, objective='val_accuracy',␣
 ↪max_epochs=30, factor=3,
                    hyperband_iterations=1, directory='vit_dir3',␣
 ↪project_name='vit')

vtuner3.search(train_data, train_labels, epochs=50, batch_size=512,␣
 ↪validation_split=0.1, callbacks=[vit_early_stop, go_fast, keep_growing])

vit3_best_hps = vtuner3.get_best_hyperparameters(num_trials=1)[0]
```

```
Reloading Tuner from vit_dir3\vit\tuner0.json
```

[19]:
```
vit3_hypermodel = vtuner3.hypermodel.build(vit3_best_hps)
```

```
vit_early_stop_vacc = EarlyStopping(monitor='val_accuracy', patience=15,␣
 ↪restore_best_weights=True)

vit3_history = vit3_hypermodel.fit(train_data, train_labels, batch_size=512,␣
 ↪epochs=100, verbose=0,
                          validation_split=0.1, callbacks=[vit_early_stop_vacc])
```

[20]:
```
print("Best Hyperparameters:")
for param, value in vit3_best_hps.values.items():
    print(f"{param}: {value}")
```

```
Best Hyperparameters:
patch_size: 8
projection_dim: 128
transf_mlp_drop_rt: 0.4
mlp_layers: 1
num_heads: 4
transformer_layers: 2
lrate: 0.00020056946471786024
optimizer: adam
tuner/epochs: 30
tuner/initial_epoch: 10
tuner/bracket: 2
tuner/round: 2
tuner/trial_id: 0067
momentum: 0.9
```

0.1.15 Many Searches for Vision Transformer

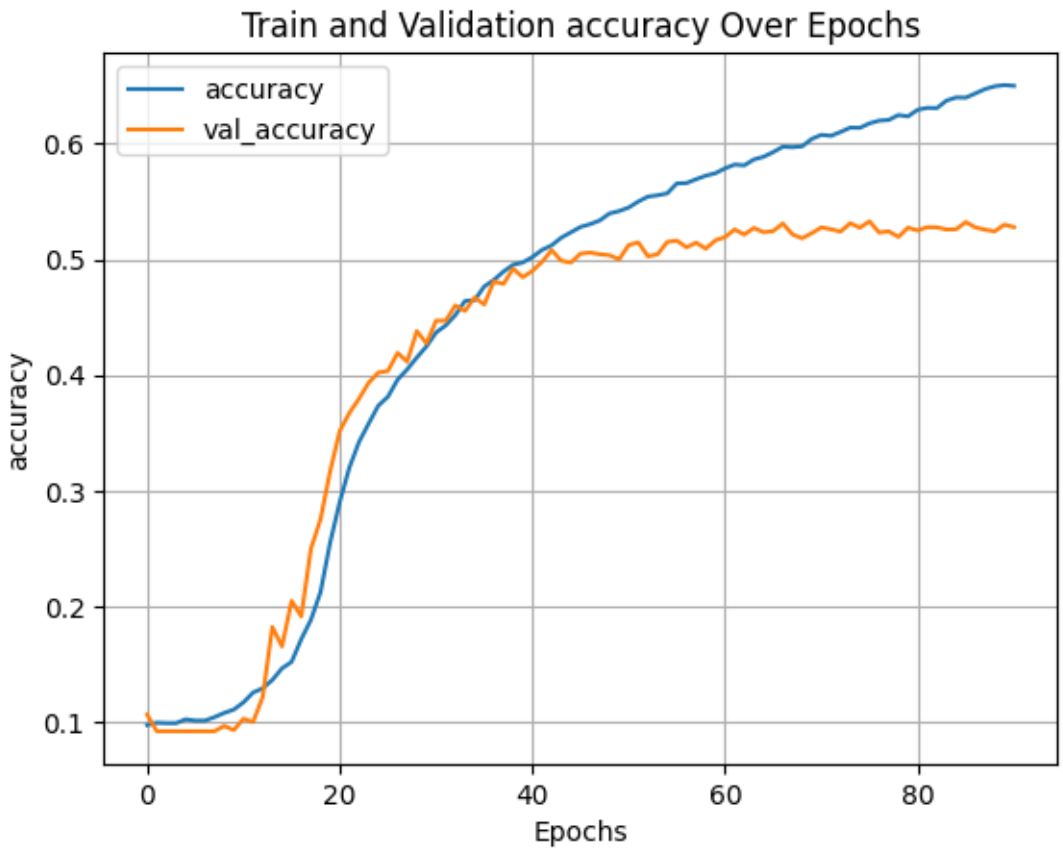
As mentioned previously, so many hyperparameters exist for transformers that interpreting a choice of adjustment which will improve performance is extremely challenging. With this in mind it seemed best to take advantage of the fact that multiple Hyperband search iterations could be used, and to select a best performer among them. This previous Hyperband search yielded the model above, which performed somewhat better with testing data. To take performance a bit further, while it was necessary to restrict searching to a max of thirty epochs to find a best choice, when we consider how cleanly train and validation accuracy/loss aligned across epochs (this was consistent throughout Hyperband search best perfromers) it is clear that vision transformers can still have room to grow. This is quite different than the characteristics seen with our best CNN models which always peaked out abruptly. Exploring transformers shows that there are a number of further extensions we could potentially implement to improve models given enough time to try them, but liberal use of search and growth constraints have still helped here. This ViT model shows similar shortcommings to what we saw before, but improves its accuracy with animals noticeably.

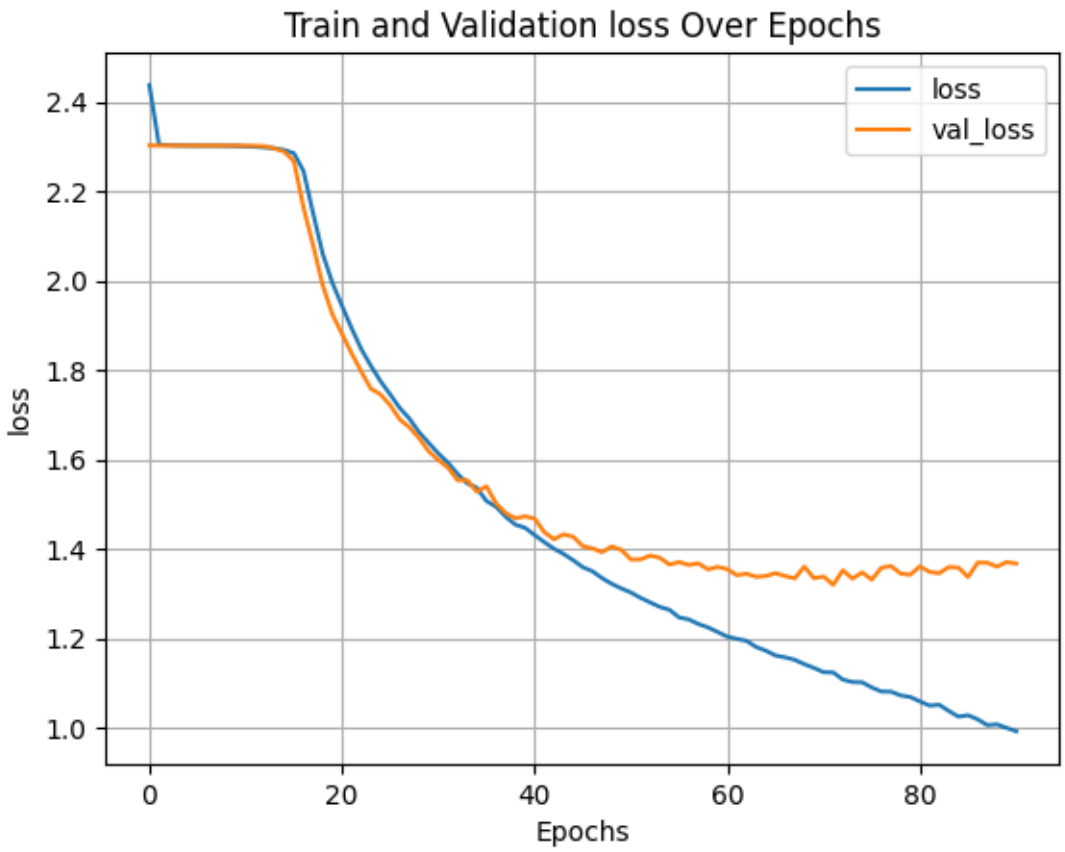
[23]:
```
resultplot(vit3_hypermodel, vit3_history)
```

```
Model: "VisionTransformer"
_________________________________________________________________________________
_________________
```

```
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================
==================
 input_1 (InputLayer)           [(None, 32, 32, 3)]  0           []

 patches (Patches)              (None, 16, 192)      0
['input_1[0][0]']

 dense_2 (Dense)                (None, 16, 128)      24704
['patches[0][0]']

 tf.__operators__.add (TFOpLamb  (None, 16, 128)     0
['dense_2[0][0]']
 da)

 layer_normalization (LayerNorm  (None, 16, 128)     256
['tf.__operators__.add[0][0]']
 alization)

 multi_head_attention (MultiHea  (None, 16, 128)     263808
['layer_normalization[0][0]',
 dAttention)
'layer_normalization[0][0]']

 add (Add)                      (None, 16, 128)      0
['multi_head_attention[0][0]',
'tf.__operators__.add[0][0]']

 layer_normalization_1 (LayerNo  (None, 16, 128)     256         ['add[0][0]']
 rmalization)

 dense_3 (Dense)                (None, 16, 128)      16512
['layer_normalization_1[0][0]']

 dropout_1 (Dropout)            (None, 16, 128)      0
['dense_3[0][0]']

 add_1 (Add)                    (None, 16, 128)      0
['dropout_1[0][0]',
                                                                  'add[0][0]']

 layer_normalization_2 (LayerNo  (None, 16, 128)     256         ['add_1[0][0]']
 rmalization)

 multi_head_attention_1 (MultiH  (None, 16, 128)     263808
['layer_normalization_2[0][0]',
 eadAttention)
'layer_normalization_2[0][0]']
```

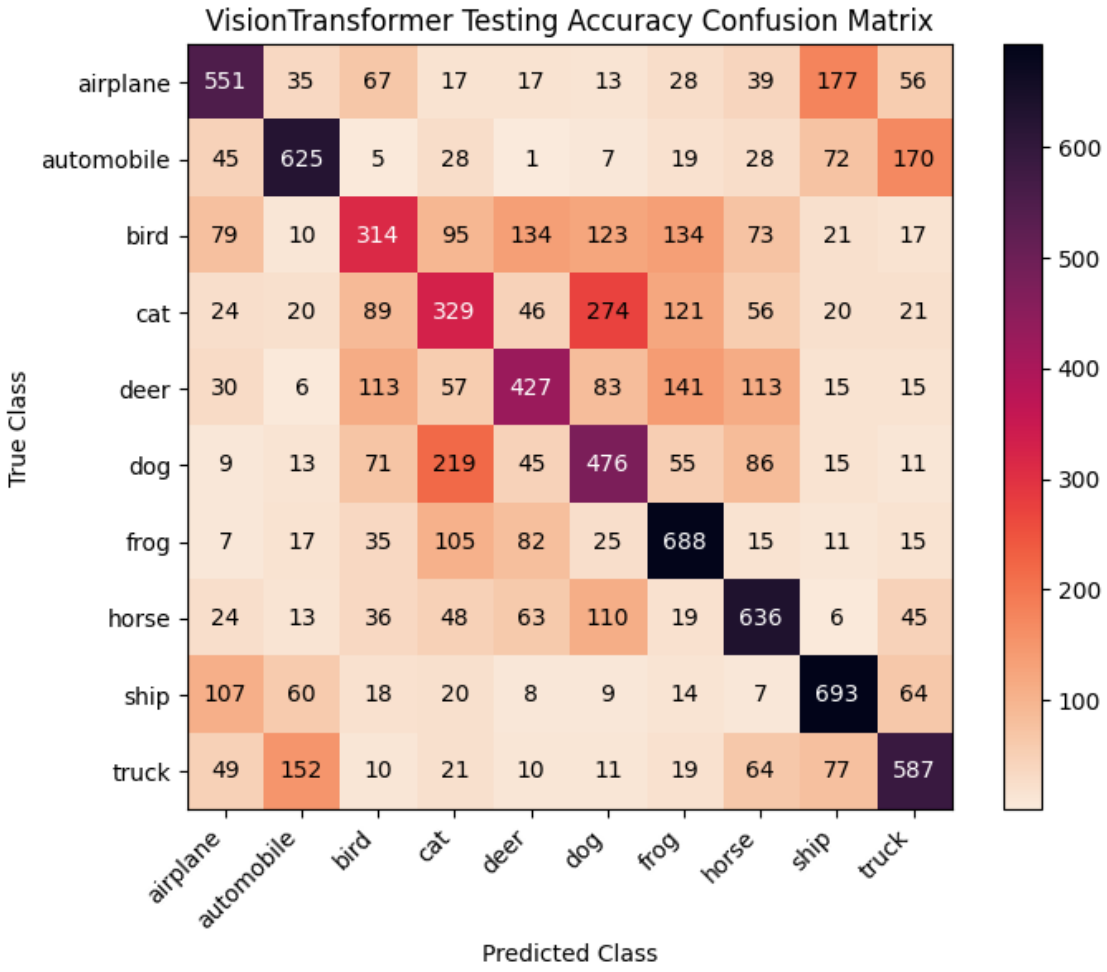
```
 add_2 (Add)                    (None, 16, 128)      0
['multi_head_attention_1[0][0]',
                                                                  'add_1[0][0]']

 layer_normalization_3 (LayerNo  (None, 16, 128)     256         ['add_2[0][0]']
 rmalization)

 dense_4 (Dense)                (None, 16, 128)      16512
['layer_normalization_3[0][0]']

 dropout_2 (Dropout)            (None, 16, 128)      0
['dense_4[0][0]']

 add_3 (Add)                    (None, 16, 128)      0
['dropout_2[0][0]',
                                                                  'add_2[0][0]']

 layer_normalization_4 (LayerNo  (None, 16, 128)     256         ['add_3[0][0]']
 rmalization)

 flatten_1 (Flatten)            (None, 2048)         0
['layer_normalization_4[0][0]']

 dropout_3 (Dropout)            (None, 2048)         0
['flatten_1[0][0]']

 dense_5 (Dense)                (None, 128)          262272
['dropout_3[0][0]']

 dropout_4 (Dropout)            (None, 128)          0
['dense_5[0][0]']

 dense_6 (Dense)                (None, 10)           1290
['dropout_4[0][0]']

==================================================================================
==================
Total params: 850,186
Trainable params: 850,186
Non-trainable params: 0
----------------------------------------------------------------------------------
------------------
None
Best epoch: 76
```


Train and Validation accuracy Over Epochs
accuracy
val_accuracy
0.6
0.5
0.4
0.3
0.2
0.1
accuracy
0
20
40
60
80
Epochs



[22]:
```
testperf(vit3_hypermodel, test_data, test_labels)
```

```
313/313 [==============================] - 7s 23ms/step - loss: 1.3248 -
accuracy: 0.5326
[test loss, test accuracy]: [1.3248401880264282, 0.5326000452041626]
313/313 [==============================] - 7s 23ms/step
```



[ ]:

0.1.16 ROC AUC Analysis

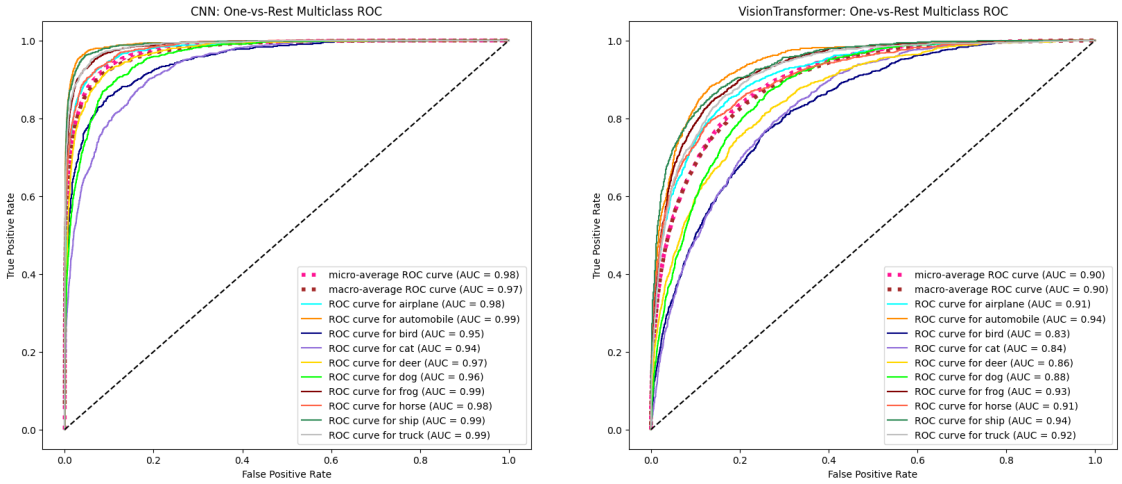
To compare our improved models more directly, we use One-vs-Rest ROC AUC plotting. This page from sklearn was helpful to create multiclass ROC curve plots for final models. While the problem type may be the same for these models, we have seen that they work very differently fundamentally. Loss curves across epochs and test data confusion matrices are not necessarily enough to distinguish a classifier model. To examine the results in a way more particular to classification, we can plot ROC AUC. This is typically used for binary classification tasks, where each curve on a plot represents the behavior of an individual model, but we can extend this. One-vs-Rest allows for plotting each class in a multiclass model independently, as well as overall ROC scores per model. Each plot line then represents transition between True Positive rate and False across thresholds. Overall ROC is represented through micro and macro averaging, where macro averaging is an unweighted average across all the classes and micro averaging aggregates the individual error contributions from all the classes. Micro average is preferable in cases where classes are imbalanced, though micro and macro will be closely aligned in our case. This makes sense given that the class distribution in our data

is extremely balanced.

[24]:
```
models_list = [cnn_test_model, vit3_hypermodel]
names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',␣
 ↪'horse', 'ship', 'truck']
K = len(models_list)
n_classes = 10
y_train = np.argmax(train_labels, axis=1)
y_test = test_batch[b'labels']
fig, ax = plt.subplots(1, K, figsize=(20, 8))


for k in range(K):

    y_score = models_list[k].predict(test_data)

    label_binarizer = LabelBinarizer().fit(y_train)
    y_onehot_test = label_binarizer.transform(y_test)
    #y_onehot_test.shape  # (n_samples, n_classes)

    # store the fpr, tpr, and roc_auc for all averaging strategies
    fpr, tpr, roc_auc = dict(), dict(), dict()
    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = roc_curve(y_onehot_test.ravel(), y_score.
 ↪ravel())
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])


    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_onehot_test[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    fpr_grid = np.linspace(0.0, 1.0, 1000)

    # Interpolate all ROC curves at these points
    mean_tpr = np.zeros_like(fpr_grid)

    for i in range(n_classes):
        mean_tpr += np.interp(fpr_grid, fpr[i], tpr[i])  # linear interpolation

    # Average it and compute AUC
    mean_tpr /= n_classes

    fpr["macro"] = fpr_grid
    tpr["macro"] = mean_tpr
    roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])
```

```
    macro_roc_auc_ovr = roc_auc_score(y_true=y_test, y_score=y_score,␣
 ↪multi_class="ovr", average="macro",)

    print(f"Micro-averaged and Macro-Averaged One-vs-Rest ROC AUC score for␣
 ↪{models_list[k].name} \
classifier: \n{roc_auc['micro']:.2f}, {macro_roc_auc_ovr:.2f}")

    ax[k].plot(fpr["micro"],tpr["micro"],label=f"micro-average ROC curve (AUC =␣
 ↪{roc_auc['micro']:.2f})",
        color="deeppink",linestyle=":",linewidth=4,
    )

    ax[k].plot(fpr["macro"],tpr["macro"],label=f"macro-average ROC curve (AUC =␣
 ↪{roc_auc['macro']:.2f})",
        color="brown",linestyle=":",linewidth=4,
    )

    colors = ["aqua", "darkorange", "navy", "mediumpurple", "gold", "lime",␣
 ↪"maroon", "tomato", "seagreen", "silver"]
    for class_id, color in zip(range(n_classes), colors):
        RocCurveDisplay.from_predictions(
            y_onehot_test[:, class_id],
            y_score[:, class_id],
            name=f"ROC curve for {names[class_id]}",
            color=color,
            ax=ax[k],
        )

    _ = ax[k].set(xlabel="False Positive Rate",ylabel="True Positive Rate",
        title=f"{models_list[k].name}: One-vs-Rest Multiclass ROC",
    )

    ax[k].plot([0, 1], [0, 1], 'k--', label='Chance level')
```

```
313/313 [==============================] - 1s 4ms/step
Micro-averaged and Macro-Averaged One-vs-Rest ROC AUC score for CNN classifier:
0.98, 0.97
313/313 [==============================] - 6s 19ms/step
Micro-averaged and Macro-Averaged One-vs-Rest ROC AUC score for
VisionTransformer classifier:
0.90, 0.90
```



0.1.17 Interpreting Results

We can see that our final Vision Transformer model still needs development. Good ROC would mean curves are less bow-shaped and more full. There is also quite a bit more of variance amongst the classes. Cat and bird classes proved the most difficult, and automobile and ship were learned best for the ViT. Our clear winner is the convolutional neural network. Its micro and macro averages are quite high, and the area and steepness of class curves are relatively full and pointed. Its best quality is how progression in its True Positive rate produces very little give on average along the axis of False Positive Rate. This is very ideal behavior. Perhaps unexpectedly, despite the major differences in design and performance between models the relative ordering on which classes are learned more or less effectively is quite similar. We could interpret this as telling us about which of these classes are inherently challenging for image classifiers.

While vision transformers have demonstrated themselves as highly performant in real world settings, in this project they have also demonstrated that their use comes at a strict cost of complexity, interpretability, and development time where weights and architecture are independently constructed and techniques such as transfer learning are not able to be exploited.

0.1.18 Wrapping Up

For the range of hyperparameters explored convolutional neural networks were able to be found through Hyperband search which performed fairly well. We can infer that the filtering and pooling process used by the CNNs was able to learn image features (characteristics) which served to identify classes fairly accurately. By comparison vision transformers proved capable of the task, but with a steep increase in challenge and complexity. This is most easily illustrated by the fact that hyperparameter searching grew so much more complicated, and required many more iterations and time to get as far as was managed. We saw from the ROC plot that there are clear similarities between which classes the two models learn better and worse. It may be then that - for hyperparameters tried - they are learning to identify many of the same image features, but that CNNs in this case must be learning those features in a more effective way, or perhaps are better equipped to learn more of them. Either way, CNNs proved a fast, more interpretable, and accessible learning method with good performance.

With all this in mind, it is still important to recognize that transformers currently make up the state of the art for machine learning. Vision transformers really should have done better here, and there are likely to be many paths to achieving better performance from ViTs which could have been implemented given enough time. More careful exploration of optmizers like SGD, and RMSProp could help them to be more competitive. This means exploring controls like L2 regularization to help keep validation performance consistent, more complex techniques of controlling momentum to allow optimizers to traverse the optimization surface better, and hyperparameter searching with smaller batch size and looser time constraints to help the amount of variance in the gradients of the optimization surface in case a significantly more suitable minimum can be found. Additionally, randomized image augmentation, e.g. horizontal flipping, and/or image preprocessing methods used with popular openly available networks like VGG-16 or ResNet-50 seems to hold the best potential for significant improvemnts to performance given that including techinques like these are so common. Approaches like these seem certain to bare results, although they continue with the trend observed in the project of how transformers represent a powerful learning method, albeit one which comes at a significant cost.

0.1.19 References

Krizhevsky, A. (2009a). CIFAR-10 and CIFAR-100 datasets. Toronto.edu. https://www.cs.toronto.edu/~kriz/cifar.html

Krizhevsky, A. (2009b). Learning Multiple Layers of Features from Tiny Images. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

Lu, M. (2020). Keras with TensorFlow Course - Python Deep Learning and Neural Networks for Beginners Tutorial. Www.youtube.com; freeCodeCamp.org. https://www.youtube.com/watch?v=qFJeN9V1ZsI

Keras Team. (n.d.). Models API. https://keras.io/2.18/api/

Keras Team. (n.d.). KerasTuner API documentation. https://keras.io/keras_tuner/api/

Keras Team. (n.d.). Getting started with KerasTuner. https://keras.io/keras_tuner/getting_started/

Agu, E. (2014). Digital image processing (CS/ECE 545): Lecture 2: Histograms and point operations [Lecture slides]. https://web.cs.wpi.edu/~emmanuel/courses/cs545/S14/slides/lecture02.pdf

Edeza, T. (2020). Image processing with Python: Application of Fourier transformation. Towards Data Science. https://towardsdatascience.com/image-processing-with-python-application-of-fourier-transformation-5a8584dc175b

Fisher, R., Perkins, S., Walker, A., & Wolfart, E. (n.d.). Intensity histogram. https://homepages.inf.ed.ac.uk/rbf/HIPR2/histgram.htm

Fisher, R., Perkins, S., Walker, A., & Wolfart, E. (n.d.). Histogram equalization. https://homepages.inf.ed.ac.uk/rbf/HIPR2/histeq.htm

Cohen, M. (2017). How the 2D FFT works. youtube.com; Mike X Cohen. https://www.youtube.com/watch?v=v743U7gvLq0

Dror, R. (2018). Fourier transforms and convolution (without the agonizing pain) [Lecture notes]. Stanford University.

https://web.stanford.edu/class/archive/cs/cs279/cs279.1182/lectures/lecture9-annot.pdf

Gros, C. (2022). Visual transformer and the MNIST dataset. https://itp.uni-frankfurt.de/~gros/StudentProjects/WS22_23_VisualTransformer/

Salama, K. (2021). Image classification with Vision Transformer - Keras. https://keras.io/examples/vision/image_classification_with_vision_transformer/#introduction

TensorFlow Team. (n.d.). Introduction to the Keras Tuner. https://www.tensorflow.org/tutorials/keras/keras_tuner

Brownlee, J. (2019). How to use learning curves to diagnose machine learning model performance. https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

TensorFlow Team. (n.d.). Migrate early stopping. https://www.tensorflow.org/guide/migrate/early_stopping

Li, L., & Jamieson, K. G. (2016). Hyperband: A novel bandit-based approach to hyperparameter optimization [Preprint]. arXiv. https://arxiv.org/pdf/1603.06560

O'Malley, T. (2020). Hyperparameter tuning with Keras Tuner. TensorFlow Blog. https://blog.tensorflow.org/2020/01/hyperparameter-tuning-with-keras-tuner.html

Brownlee, J. (2020). Object classification with CNNs using the Keras deep learning library. https://machinelearningmastery.com/object-recognition-convolutional-neural-networks-keras-deep-learning-library/

Keras Team. (n.d.). Hyperband Tuner - Keras.io documentation. https://keras.io/keras_tuner/api/tuners/hyperband/

Scikit-learn Developers. (n.d.). Multiclass receiver operating characteristic (ROC). https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html#sphx-glr-auto-examples-model-selection-plot-roc-py

[ ]: