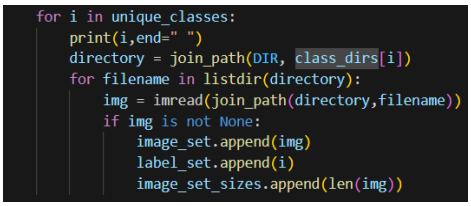
Assignment 2 Report

Student name: Omar Afifi
Student ID: 31105327
Student's Monash email: oafi0001@student.monash.edu
Your Kaggle ID: https://www.kaggle.com/omarafifi31105327
Submission date: 28/05/2023

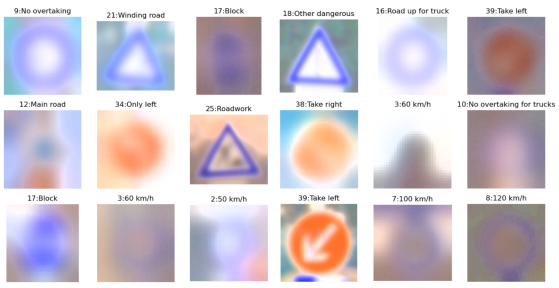
Part 1. Data analysis of signboard dataset

Task 1.1: Dataset preparation

We import all the images using opencv library, and visualize a random snippet of the data alone with their respective labels to make sure the data have been imported properly

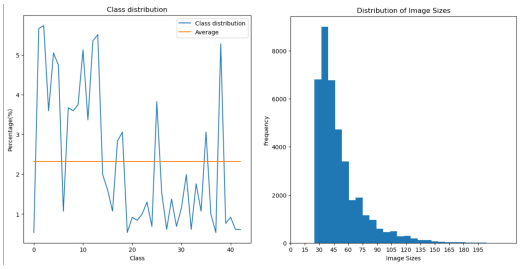
Explanation	Code snippet
1- We define the labels array which corresponds to the names of each of our classes 2- we define the directory folder of our data 3- we set the variable "class_dirs" for storing the last part of our data path to be able to get images from all the folders (folders 0 to 42)	
We iterate through all the folders and load all the images in each folder and store it in our images array, as well as their corresponding label. Now it is important to note that this tedious method was used instead of using an efficient built-in keras functions, due to the fact that we need to analyze the sizes of all of our images, and using keras functions wouldn't allow us to do that since they require to resize all images to one size when loading	

```
for i in unique_classes:
    print(i,end=" ")
    directory = join_path(DIR, class_dirs[i])
    for filename in listdir(directory):
        img = imread(join_path(directory,filename))
        if img is not None:
            image_set.append(img)
            label_set.append(i)
            image_set_sizes.append(len(img))
```

Discussion	Image
To the right are Two random snippets, as well as their corresponding labels, and we can see that they do match, which means our data have been imported properly and are ready for analysis. It is important to note that these images are shown in BRG as opposed to RGB, which has to do with how opencv reads images by default, however it is not a cause of concern for now as it will be addressed later on	

Task 1.2: Data analysis

We firstly visualize both size distribution and class distribution:

Discussion	Image
-On the rightWe can see that most of our data has a size between 30 to 45 pixels. Which means we will need to scale them up to a size of 64, which might lead to blurriness in our data. -On the left, we can see a massive discrepancy in our class distribution, with some classes 10 times larger than other classes, we can also see the orange line representing the average, which is simply the number of images in the dataset/ number of classes.	

So based on that, two things need to be done:

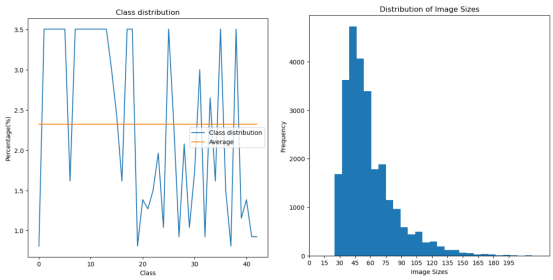
1- Undersampling the large classes by removing the smallest size images in each until we reach the desired average

2- Data augmentation to bring the number of images in small classes up by transforming copies of all the images in such classes randomly

Undersampling:

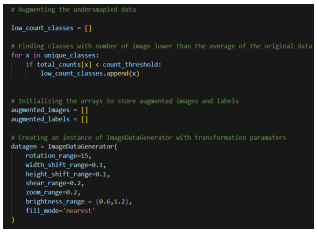
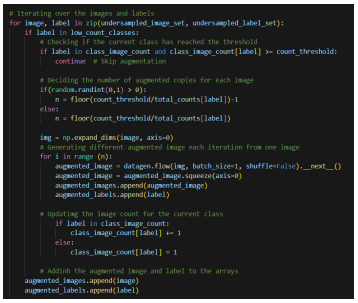
Explanation	Code snippet
Here we loop through all the classes and find out if they have a number of images bigger than a threshold (which would be the average we have shown in the graph earlier). if the class has larger number than the threshold then it sorts the image in that class by size, and removes the smallest images until the desired number of images is reached, and adds those images that are left to our augmented images array, as well as their corresponding labels to the augmented labels array, and the same is done for all classes	

```
count_threshold = int(total_num/NUM_CLASSES)
# Initializing arrays to store the undersampled data
undersampled_image_set = []
undersampled_label_set = []

for label in unique_classes:
    # Get the indices of images belonging to the certain class
    class_indices = np.where(label_set == label)[0]
    first_index = class_indices[0]
    num_images = total_counts[label]
    if num_images > count_threshold:
        # Sort the class images based on their size
        sorted_indices = sorted(class_indices, key=lambda i: len(image_set[i]))
        # Remove the smallest images until threshold is reached
        remove_indices = sorted_indices[:num_images - count_threshold]
        remove_indices = remove_indices - first_index
        class_indices = np.delete(class_indices, remove_indices)


    undersampled_image_set.extend([image_set[i] for i in class_indices])
    undersampled_label_set.extend([label_set[i] for i in class_indices])
```

Discussion	Image
On the right We can see that the number of images with sizes between 30 to 45 reduced massively, which is what we were looking for On the left we can also see that the class distribution is way more balanced compared to the original data	

Despite the better class balance, there is still a big difference, with some classes 4 times larger than other classes, so that brings us to the next step, data augmentation

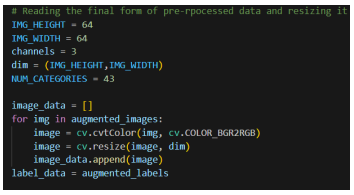
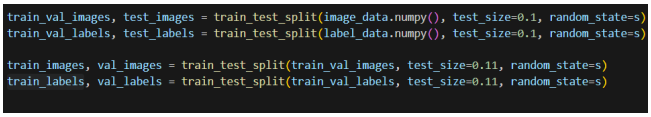
Data Augmentation

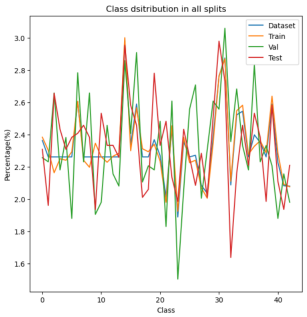
Explanation	Code snippet
Here we first find which classes have number of images below our threshold (which is defined as the same average we mentioned earlier) Then we create an instance of in Image data generator, which is a built in function is keras that we will be utilizing to introduce random transformation to our images	
Next we iterate through all the images in classes that are below the threshold in our under sampled data, and we decide how many transformed copies per image we need so that we can arrive at the threshold. Then we transform each image the required number of times and we add each transformed image to our augmented data arrays, as well as their corresponding label, and we make sure to also add the original image.	

Part 2. Developing a deep learning-based perception model for AV

Task 2.1: Input processing.

firstly, we need to resize the data, as well as convert the color channels of our images to RGB, then after that we split our data into train, validation and test

Explanation	Code snippet
Here we are setting up important parameters such as the image height and width, as well as the number of classes in our dataset. Then we loop through our augmented image dataset and resize and change the color channel of each image	
here we simply convert both our label data and image data to tensors to allow ease of use later on, not that we converted them to numpy arrays first to accelerate the conversion to tensors	
Next we split our data into training and testing data first, using a random shuffler since our data so far has been in order(all images from first class, then all images from second class, etc.) to ensure that class distribution is the same for all splits, then we split the training data into training data and validation data, with ratio 0.11, since 0.11*0.9 = 0.1	

Discussion	Image
Here we can see the class distribution of all of our splits, as well ass the original augmented data, and we can see theta re pretty close, with variations not more than 0.5%, which is exactly what we want to ensure no bias between classes exists when training, validating and testing our model	

Task 2.2: Deep learning model development.

We start with a very simply cnn model, with one convolution layer as our input layer, one max pooling layer, a flatten layer and a dense layer of 128, in addition to another dense layer of 43 neurons for our output

Explanation	Code snippet

Here we define our model architecture, and print a summary of it(which will be shown in the next part for comparison)	`simple_model = keras.Sequential([`
	`  keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(IMG_HEIGHT, IMG_WIDTH, channels)),# convolution layer 1 (1),`
	`  keras.layers.MaxPooling2D(pool_size=(2, 2)),# max pooling layer 1 (2)`
	`  keras.layers.Flatten(),  # flatten layer (3)`
	`  keras.layers.Dense(128, activation='relu'),  # dense layer 1 (4)`
	`  keras.layers.Dense(NUM_CATEGORIES, activation='softmax') # output layer (5)`
	`])`
	`print(simple_model.summary())`

We then define our hyperparameters and compile our model. the batch size was chosen for speed, since from observation, it doesn't seem to affect the training that much. and the compiling parameters were chosen since they are pretty standard, even though they do describe important hyper parameters such as learning rate, the effect of such hyperparameters on the performance of the cnn was not explored in this assignment to simplify the problem	`# Hyperparamaters`
	`num_epochs = 20`
	`batch_size = 64`
	
	`simple_model.compile(optimizer='adam',`
	`  loss='sparse_categorical_crossentropy',`
	`  metrics=['accuracy'])`

we then train our model and save it, so that we don't have to train it everytime we need to test its performance	`#training and saving the model`
	`simple_model.fit(`
	`  train_images, np.array(train_labels), validation_data=(val_images,np.array(val_labels)),epochs=num_epochs, batch_size = batch_size)`
	
	`simple_model.save('simple_model.h5')`

Then we load and test our model, and print the classification report to assess our model performance	`# Testing and evaluating the model`
	`simple_model = keras.models.load_model(r'C:\Users\Omar\Desktop\Uni\2023\CIV4100\Assignment 2\simple_model.h5')`
	
	`label_pred = simple_model.predict(test_images,batch_size = batch_size)`
	`label_pred = np.argmax(label_pred, axis=1)`
	
	`# showing preciosion, recall and f1 scores`
	`classification_rep = classification_report(test_labels, label_pred)`
	`print("Classification Report:")`
	`print(classification_rep)`

A small snippet of the precision, recall and f1 scores of our model for a select number of classes is shown on the top right(both classification reports will entirely be in the appendix)Thanks to data augmentation and undersampling, we see no discrepancy in performance that is related to the number of images in each class, since the number of images in each class is pretty close, and we also know that by looking at the correlation between precision and recall. And since we can see that generally speaking, precision follows recall, we can say that, there is no one specific class at a

	precision	recall	f1-score	support
0	0.70	0.49	0.58	95
1	0.26	0.44	0.32	82
2	0.35	0.26	0.29	98
3	0.39	0.28	0.33	82
4	0.42	0.22	0.29	96
5	0.12	0.11	0.12	91

16	0.56	0.43	0.49	117
17	0.82	0.97	0.89	91
18	0.53	0.69	0.60	85
19	0.32	0.38	0.35	98
20	0.57	0.56	0.56	81
21	0.53	0.13	0.21	77

"disadvantage". And we can further see the truth of this when comparing the top classification (with data augmentation and undersampling) report, with the bottom one(no data augmentation or undersampling), and we can clearly see on the bottom report the discrepancy in the performance that is closely related to the number of images in a class.

	precision	recall	f1-score	support
0	0.60	0.12	0.19	26
1	0.48	0.47	0.47	225
2	0.37	0.53	0.44	215
3	0.31	0.38	0.34	143
4	0.40	0.35	0.38	181
5	0.39	0.35	0.37	188

16	0.40	0.20	0.27	30
17	0.94	0.94	0.94	109
18	0.77	0.56	0.65	133
19	0.30	0.41	0.35	17
20	0.68	0.45	0.54	29
21	0.60	0.54	0.57	28

Another thing to note from the classification report is,, since we ruled out the possibility of the number of classes being responsible for the discrepancy in performance, and since discrepancy in performance is clear here, then this likely due to the fact that some signs are easier to learn to predict than others (eg. class 17 which is shown on the right, is easier to learn than class 16 which is shown on the left, since with class 17, the color of the background narrows down the choices to two, whereas with class 16 we need to learn more about the details of the sign, since most signs have a white background). So based on that, we can say that our simple model hasn't learned enough about the harder to learn classes, and so we'd need to address that when we fine tune our model.

Another thing we need to address is overfitting, even though our simple model didn't come close to overfitting, it still had some behaviours that are characteristic of overfitting. As we can see from the image on the right (the entire history of the model will be provided in the appendix), the training accuracy has increased between epochs 14 and 18, whereas our validation accuracy has not, and so this is also something to consider in the next step when designing a more complex model to ensure it doesn't overfit.

```
Epoch 14/20
503/503 [==============================] - 18s 36ms/step - loss: 1.3423 -
accuracy: 0.5704 - val_loss: 1.4412 - val_accuracy: 0.5587
Epoch 15/20
503/503 [==============================] - 19s 37ms/step - loss: 1.3171 -
accuracy: 0.5795 - val_loss: 1.6389 - val_accuracy: 0.5014
Epoch 16/20
503/503 [==============================] - 20s 39ms/step - loss: 1.3184 -
accuracy: 0.5792 - val_loss: 1.4735 - val_accuracy: 0.5522
Epoch 17/20
503/503 [==============================] - 20s 40ms/step - loss: 1.3902 -
accuracy: 0.5684 - val_loss: 1.4155 - val_accuracy: 0.5580
Epoch 18/20
503/503 [==============================] - 20s 40ms/step - loss: 1.3046 -
accuracy: 0.5861 - val_loss: 1.5888 - val_accuracy: 0.5467
```

And finally, the overall accuracy of our model as well as average precision, recall and f1 scores(respectively from left to right) can be seen to the right, again we can see here that our model hasn't learned enough.

accuracy			0.59	4015
macro avg	0.61	0.59	0.58	4015
weighted avg	0.61	0.59	0.59	4015

Task 2.3: Model improvement.

Based on our observation from the last part, there are three aspect in our model that we need to address:

- Helping our model to learn faster, so that it can predict harder to learn classes
- Chance of overfitting
- Overall structure

Helping our model learn:

A very simple way we can help our model learn faster is by normalizing data, that is limiting our pixel values to be between 0 to 1 instead of 0 to 255, and this can be clearly seen below, where we can see, just by normalizing our input data, the model performance increase significantly (the data below shows the average precision, recall and f1 scores respectively from left to right, as well as overall accuracy on the top)

With normalization	without normalization

accuracy			0.82	3921
macro avg	0.83	0.81	0.81	3921
weighted avg	0.83	0.82	0.82	3921

accuracy			0.62	3921
macro avg	0.63	0.58	0.59	3921
weighted avg	0.64	0.62	0.62	3921

There are also plenty of other ways to help our model learn, which will be addressed when we discuss the model architecture

Avoiding overfitting:

There are a few ways we can go about avoiding overfitting, one of which is terminating the learning process once the validation accuracy stops improving(which is known as early stopping). However we went with a different approach, since the validation accuracy might start improving again after a while, we'd be limiting how much our model can learn by using early stopping, so the approach that we used to address overfitting is by saving the model that had the best validation accuracy every time there is an improvement in the validation accuracy. and lucky for us, there is a built-in function in keras that allows us to do that

Explanation	Code snippet

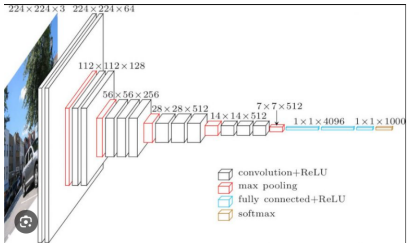
we define a variable "checkpoint" to be used as one of our callbacks when using the"fit()" function. this call back always checks the validation accuracy and compares it with the validation accuracy of the model in a given directory, if it is bigger, then the previous model in that given directory gets replaced by the model with the higher validation accuracy	`# Training the model`
	`checkpoint = keras.callbacks.ModelCheckpoint('finetuned_modelv2.h5',`
	`monitor='val_accuracy',`
	`mode='max',`
	`save_best_only=True,`
	`verbose=1)`
	
	`history = finetuned_model.fit(`
	`train_generator, validation_data=(norm_val_images, np.array(val_labels)), epochs=num_epochs, callbacks=[checkpoint],steps_per_epoch = len(train_images) // batch_size)`

Overall structure:

Deciding on the structure of our cnn, that is, how many convolution layers, number and size of filters in each one, number of max pooling layers, etc, might be the hardest part to finetune, since there is no size fits all, and it is one of the biggest areas of research on the topic of neural networks. That being said, there are some general guidelines that has been set by researches, and these guidelines have long been proven to be effective, to list a few:

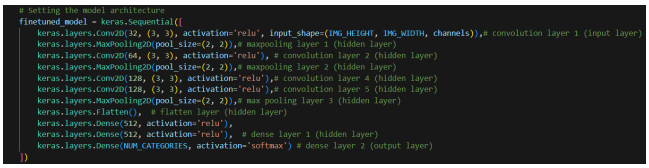
- Avoiding bottlenecks, that is the size of the inputs of our layers doesn't change significantly between two layers
- Balancing the width and the depth of the network, that is the number of filters in each convolution layer should be correlated with the number of convolution layers stack upon each other
- Spatial aggregation, that is applying filters using our convolution layers, is almost insensitive to reducing the dimension of the input data. And this point is the basis of the usefulness of using pooling layers

In addition, to these guidelines, we can mimic the architecture of a CNN that we know works, one of which is VGG16.

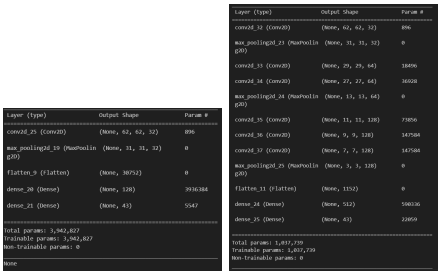


However copying a pasting the model blindly might have opposite results, and results on the model overfitting, since it would be too complex for our data, since we only 43 classes of 64*64 images, where VGG16 was used for classifying 1000 classes with 224*224 images. Nonetheless, we can definitely use VGG16 as an inspiration of how our model should look like.

Putting everything together:

Explanation	Code snippet
Here we try to mimic the architecture of VGG16, while also sticking with our guidelines and considering the simplicity of our problem. We are utilizing max pooling layers to help our model extract more data, while also reducing computation time. we also by have two convolution layers stacked on top of one another at the end of our CNN with 128 filter each, to keep a balance between depth and width	

```
# Setting the model architecture
finetuned_model = keras.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(IMG_HEIGHT, IMG_WIDTH, channels)),# convolution layer 1 (input layer)
    keras.layers.MaxPooling2D(pool_size=(2, 2)),# maxpooling layer 1 (hidden layer)
    keras.layers.Conv2D(64, (3, 3), activation='relu'), # convolution layer 2 (hidden layer)
    keras.layers.MaxPooling2D(pool_size=(2, 2)),# maxpooling layer 2 (hidden layer)
    keras.layers.Conv2D(128, (3, 3), activation='relu'),# convolution layer 4 (hidden layer)
    keras.layers.Conv2D(128, (3, 3), activation='relu'),# convolution layer 5 (hidden layer)
    keras.layers.MaxPooling2D(pool_size=(2, 2)),# max pooling layer 3 (hidden layer)
    keras.layers.Flatten(),  # flatten layer (hidden layer)
    keras.layers.Dense(512, activation='relu'),
    keras.layers.Dense(512, activation='relu'),  # dense layer 1 (hidden layer)
    keras.layers.Dense(NUM_CATEGORIES, activation='softmax') # dense layer 2 (output layer)
])
```

Discussion	Image
Based on the code above, and the image of our model summary to the right, we can also see that we barely have any bottlenecking compared to our model in the previous part shown on the left	

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_25 (Conv2D)           (None, 62, 62, 32)        896
max_pooling2d_19 (MaxPoolin  (None, 31, 31, 32)        0
g2D)
flatten_9 (Flatten)          (None, 30752)             0
dense_20 (Dense)             (None, 128)               3936384
dense_21 (Dense)             (None, 43)                5547
=================================================================
Total params: 3,942,827
Trainable params: 3,942,827
Non-trainable params: 0
_________________________________________________________________
None
```

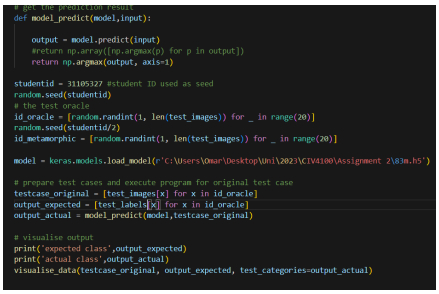
```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_32 (Conv2D)           (None, 62, 62, 32)        896
max_pooling2d_23 (MaxPoolin  (None, 31, 31, 32)        0
g2D)
conv2d_33 (Conv2D)           (None, 29, 29, 64)        18496
conv2d_34 (Conv2D)           (None, 27, 27, 64)        36928
max_pooling2d_24 (MaxPoolin  (None, 13, 13, 64)        0
g2D)
conv2d_35 (Conv2D)           (None, 11, 11, 128)       73856
conv2d_36 (Conv2D)           (None, 9, 9, 128)         147584
conv2d_37 (Conv2D)           (None, 7, 7, 128)         147584
max_pooling2d_25 (MaxPoolin  (None, 3, 3, 128)         0
g2D)
flatten_11 (Flatten)         (None, 1152)              0
dense_24 (Dense)             (None, 512)               590336
dense_25 (Dense)             (None, 43)                22059
=================================================================
Total params: 1,037,739
Trainable params: 1,037,739
Non-trainable params: 0
```

Model	overall accuracy	precision & recall	f1 score	kaggle score
Simple Model	59%	61% & 59%	59%	22%
Fine Tuned Model	95%	95% & 95%	95%	50%

Note: the value for kaggle score for the simple model was calculated manually, by comparing the prediction of 100 photos with eye, although from observation, the actual kaggle score tends to be lower than that of manual calculation

Part 3. Testing and validation of the developed perception system

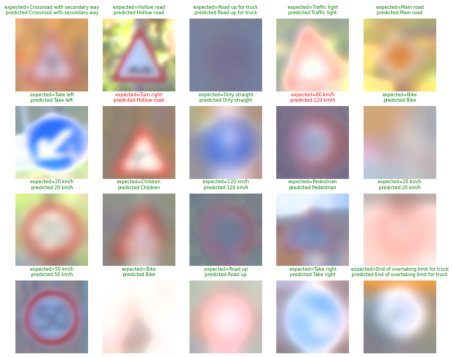
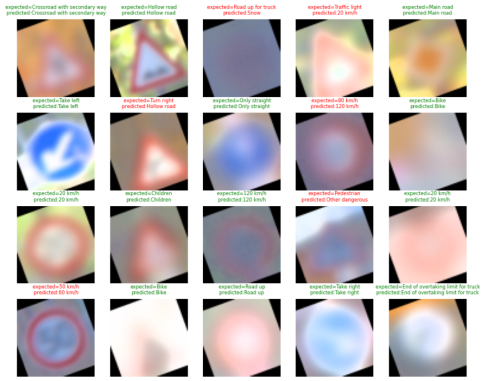
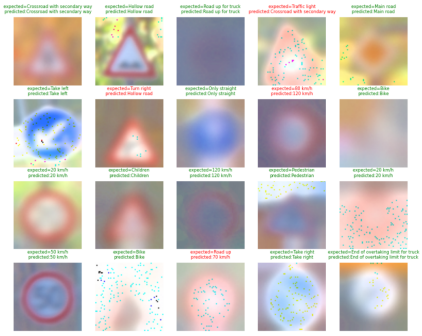
Task 3.1: Test case preparation.

Explanation	Code snippet
Here I am using my student id as a seed for generating 20 random indices for both the test oracle and the metamorphic test, these indices are then used to get the respective images and labels from our testing split	

```
# get the prediction result
def model_predict(model,input):

    output = model.predict(input)
    #return np.array([np.argmax(p) for p in output])
    return np.argmax(output, axis=1)

studentid = 31105327 #student ID used as seed
random.seed(studentid)
# the test oracle
id_oracle = [random.randint(1, len(test_images)) for _ in range(20)]
random.seed(studentid/2)
id_metamorphic = [random.randint(1, len(test_images)) for _ in range(20)]

model = keras.models.load_model(r'C:\Users\Omar\Desktop\Uni\2023\CIV4100\Assignment 2\83m.h5')

# prepare test cases and execute program for original test case
testcase_original = [test_images[x] for x in id_oracle]
output_expected = [test_labels[x] for x in id_oracle]
output_actual = model_predict(model,testcase_original)

# visualise output
print('expected class',output_expected)
print('actual class',output_actual)
visualise_data(testcase_original, output_expected, test_categories=output_actual)
```

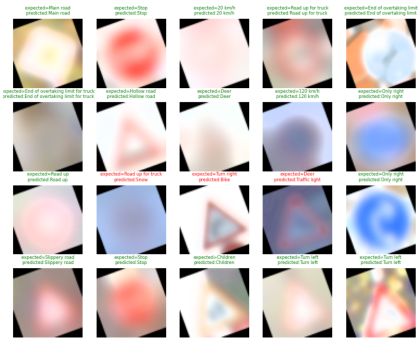
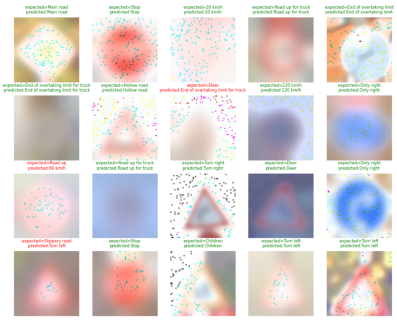
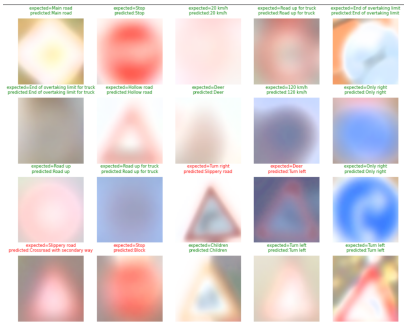
Task 3.2: Testing methods and results

With test oracle:

Discussion	Image
Here we see the original test_oracle, with both expected and actual output, and our model got 90% accuracy, which is expected from the results in the previous part	
Here we see the rotated test oracle, again along with the expected and actual output, and the accuracy of our reduced to 70%	
Here we see the noisy test oracle, again along with the expected and actual output, and the accuracy of our model reduced to 80%	

Here we see the zoomed in test oracle, again along with the expected and actual output, and the accuracy of our model reduced to 60%	

Metamorphic testing:

Discussion	**Image**
Here we see the rotated test case, shown with the predictions for both the original test case and the rotated test case, and we can see there is 85% accuracy.	
Here we see the noisy test case, shown with the predictions for both the original test case and the noisy test case, and we can see there is 85% accuracy.	
Here we see the zoomed in test case, shown with the predictions for both the original test case and the zoomed in test case, and we can see there is 80% accuracy.	

Testing method	Rotated	Noisy	Zoomed in
Traditional	70%	80%	60%
Metamorphic	85%	85%	80%

Both traditional and metamorphic testing show very close results although they mean slightly different things. in metamorphic testing, a small accuracy means high sensitivity to transformation, and vice versa with high accuracy, whereas with traditional testing, sensitivity to transformations could only be put into perspective when comparing the transformed accuracy with the original accuracy, but then again, we expect accuracies in both traditional and metamorphic testing to be correlated, which they are as shown in the above table.

Furthermore, this table tells us that our model is relatively resilient against rotation, which makes sense, since rotation was used as one of the transformations for augmenting our data as we mentioned at the first section of the report. Our model also seems to be relatively resilient against noise, because in a sense, we can think of the noise as the background of any of these images, which our model has been trained to ignore. And lastly, our model doesn't do very well with the zoomed in images, even though zooming in was also one of the parameters that were used for augmenting our data, however this is likely due to the fact that we used a maximum zoom of 1.1 when augmenting our data, whereas in our test case, we used 1.3, which is more than what our model is used to.

Areas of improvement:

To sum up, there are certain things we can do to improve on our model:

- Introduce more randomness in the data augmentation stage, to make the model more resilient against random data
- experiment with different learning rates, and the effects they have on overfitting and overall accuracy
- Regularization techniques such as L1 and L2 to help with generalization

References:

https://arxiv.org/pdf/1512.00567.pdf

https://arxiv.org/pdf/1405.3531.pdf

http://yann.lecun.org/exdb/publis/pdf/boureau-icml-10.pdf

https://stackoverflow.com/questions/60390707/how-to-choose-the-number-of-convolution-layers-and-filters-in-cnn

Appendix:

Simple model:

With Augmentation and under sampling used:

Classification Report:

	precision	recall	f1-score	support
0	0.70	0.49	0.58	95
1	0.26	0.44	0.32	82
2	0.35	0.26	0.29	98
3	0.39	0.28	0.33	82
4	0.42	0.22	0.29	96
5	0.12	0.11	0.12	91
6	0.48	0.80	0.60	116
7	0.24	0.35	0.29	88
8	0.70	0.51	0.59	97
9	0.83	0.83	0.83	87
10	0.93	0.98	0.95	94
11	0.49	0.49	0.49	85
12	0.94	0.98	0.96	102
13	1.00	0.99	0.99	77
14	0.93	0.86	0.89	118
15	0.41	0.55	0.47	94
16	0.56	0.43	0.49	117
17	0.82	0.97	0.89	91
18	0.53	0.69	0.60	85

19	0.32	0.38	0.35	98
20	0.57	0.56	0.56	81
21	0.53	0.13	0.21	77
22	0.38	0.35	0.37	102
23	0.56	0.23	0.33	100
24	0.33	0.52	0.40	85
25	0.64	0.55	0.59	87
26	0.48	0.61	0.53	76
27	0.31	0.42	0.35	77
28	0.51	0.43	0.47	83
29	0.46	0.37	0.41	98
30	0.33	0.62	0.43	103
31	0.49	0.27	0.35	113
32	0.61	0.61	0.61	79
33	0.90	0.91	0.90	108
34	0.82	0.83	0.82	92
35	0.86	0.90	0.88	92
36	0.84	0.84	0.84	98
37	0.95	0.86	0.90	86
38	0.88	0.84	0.86	100
39	0.93	0.92	0.93	115
40	0.86	0.79	0.82	91
41	0.78	0.58	0.66	92
42	0.62	0.53	0.57	87
accuracy			0.59	4015
macro avg	0.61	0.59	0.58	4015

weighted avg 0.61 0.59 0.59 4015

History:

Epoch 1/20

503/503 [==============================] - 20s 39ms/step - loss: 29.4503 - accuracy: 0.1799 - val_loss: 2.8782 - val_accuracy: 0.2518

Epoch 2/20

503/503 [==============================] - 19s 39ms/step - loss: 2.6987 - accuracy: 0.2741 - val_loss: 2.5177 - val_accuracy: 0.2893

Epoch 3/20

503/503 [==============================] - 20s 39ms/step - loss: 2.3397 - accuracy: 0.3330 - val_loss: 2.2316 - val_accuracy: 0.3585

Epoch 4/20

503/503 [==============================] - 20s 39ms/step - loss: 2.1549 - accuracy: 0.3757 - val_loss: 2.2239 - val_accuracy: 0.3849

Epoch 5/20

503/503 [==============================] - 19s 39ms/step - loss: 2.0176 - accuracy: 0.4002 - val_loss: 1.9797 - val_accuracy: 0.4113

Epoch 6/20

503/503 [==============================] - 20s 40ms/step - loss: 1.8524 - accuracy: 0.4337 - val_loss: 1.7325 - val_accuracy: 0.4626

Epoch 7/20

503/503 [==============================] - 19s 37ms/step - loss: 1.7015 - accuracy: 0.4687 - val_loss: 1.7229 - val_accuracy: 0.4604

Epoch 8/20

503/503 [==============================] - 18s 36ms/step - loss: 1.6409 - accuracy: 0.4891 - val_loss: 1.7556 - val_accuracy: 0.4707

Epoch 9/20

503/503 [==============================] - 20s 39ms/step - loss: 1.5951 - accuracy: 0.5016 - val_loss: 1.6391 - val_accuracy: 0.5029

Epoch 10/20

503/503 [==============================] - 19s 39ms/step - loss: 1.5240 - accuracy: 0.5208 - val_loss: 1.5475 - val_accuracy: 0.5308

Epoch 11/20

503/503 [==============================] - 19s 38ms/step - loss: 1.4629 - accuracy: 0.5357 - val_loss: 1.5774 - val_accuracy: 0.5006

Epoch 12/20

503/503 [==============================] - 19s 37ms/step - loss: 1.4358 - accuracy: 0.5451 - val_loss: 1.4798 - val_accuracy: 0.5379

Epoch 13/20

503/503 [==============================] - 19s 38ms/step - loss: 1.4211 - accuracy: 0.5508 - val_loss: 1.5724 - val_accuracy: 0.5125

Epoch 14/20

503/503 [==============================] - 18s 36ms/step - loss: 1.3423 - accuracy: 0.5704 - val_loss: 1.4412 - val_accuracy: 0.5587

Epoch 15/20

503/503 [==============================] - 19s 37ms/step - loss: 1.3171 - accuracy: 0.5795 - val_loss: 1.6389 - val_accuracy: 0.5014

Epoch 16/20

503/503 [==============================] - 20s 39ms/step - loss: 1.3184 - accuracy: 0.5792 - val_loss: 1.4735 - val_accuracy: 0.5522

Epoch 17/20

503/503 [==============================] - 20s 40ms/step - loss: 1.3902 - accuracy: 0.5684 - val_loss: 1.4155 - val_accuracy: 0.5580

Epoch 18/20

503/503 [==============================] - 20s 40ms/step - loss: 1.3046 - accuracy: 0.5861 - val_loss: 1.5888 - val_accuracy: 0.5467

Epoch 19/20

503/503 [==============================] - 20s 39ms/step - loss: 1.3004 - accuracy: 0.5876 - val_loss: 1.4042 - val_accuracy: 0.5889

Epoch 20/20

503/503 [==============================] - 20s 40ms/step - loss: 1.2615 - accuracy: 0.6021 - val_loss: 1.3377 - val_accuracy: 0.596

No augmentation or undersampling:

Classification Report:

	precision	recall	f1-score	support
0	0.60	0.12	0.19	26
1	0.48	0.47	0.47	225
2	0.37	0.53	0.44	215
3	0.31	0.38	0.34	143
4	0.40	0.35	0.38	181
5	0.39	0.35	0.37	188
6	0.51	0.52	0.52	42
7	0.34	0.21	0.26	150
8	0.60	0.58	0.59	140
9	0.74	0.77	0.76	159
10	0.88	0.93	0.90	200
11	0.70	0.57	0.63	128
12	0.90	0.97	0.93	208
13	0.96	0.83	0.89	213
14	0.92	0.95	0.93	80
15	0.37	0.45	0.41	69
16	0.40	0.20	0.27	30
17	0.94	0.94	0.94	109

18	0.77	0.56	0.65	133
19	0.30	0.41	0.35	17
20	0.68	0.45	0.54	29
21	0.60	0.54	0.57	28
22	0.41	0.53	0.46	43
23	0.57	0.59	0.58	75
24	0.42	0.39	0.41	28
25	0.50	0.77	0.60	152
26	0.73	0.79	0.76	56
27	0.75	0.28	0.41	32
28	0.61	0.67	0.64	54
29	0.62	0.43	0.51	30
30	0.47	0.52	0.49	44
31	0.51	0.33	0.40	85
32	0.62	0.20	0.30	25
33	0.85	0.90	0.87	68
34	0.92	0.78	0.84	45
35	0.92	0.77	0.84	117
36	0.87	0.77	0.82	35
37	0.87	0.83	0.85	24
38	0.76	0.90	0.83	178
39	0.90	0.84	0.87	31
40	0.48	0.75	0.58	40
41	0.69	0.38	0.49	24
42	0.50	0.55	0.52	22

accuracy			0.62	3921
macro avg	0.63	0.58	0.59	3921
weighted avg	0.64	0.62	0.62	3921

History:

Epoch 1/20

490/491 [==============================>.] - ETA: 0s - loss: 37.2268 - accuracy: 0.2454

Epoch 1: val_accuracy improved from -inf to 0.34621, saving model to v0_best_model.h5

491/491 [==============================] - 18s 36ms/step - loss: 37.1767 - accuracy: 0.2454 - val_loss: 2.2775 - val_accuracy: 0.3462

Epoch 2/20

490/491 [==============================>.] - ETA: 0s - loss: 2.1099 - accuracy: 0.3929

Epoch 2: val_accuracy improved from 0.34621 to 0.41654, saving model to v0_best_model.h5

491/491 [==============================] - 18s 36ms/step - loss: 2.1101 - accuracy: 0.3929 - val_loss: 2.0547 - val_accuracy: 0.4165

Epoch 3/20

490/491 [==============================>.] - ETA: 0s - loss: 1.8594 - accuracy: 0.4499

Epoch 3: val_accuracy improved from 0.41654 to 0.45286, saving model to v0_best_model.h5

491/491 [==============================] - 17s 35ms/step - loss: 1.8593 - accuracy: 0.4499 - val_loss: 1.9220 - val_accuracy: 0.4529

Epoch 4/20

490/491 [==============================>.] - ETA: 0s - loss: 1.6840 - accuracy: 0.4908

Epoch 4: val_accuracy improved from 0.45286 to 0.53194, saving model to v0_best_model.h5

491/491 [==============================] - 17s 35ms/step - loss: 1.6837 - accuracy: 0.4909 - val_loss: 1.5239 - val_accuracy: 0.5319

Epoch 5/20

490/491 [==============================>.] - ETA: 0s - loss: 1.5665 - accuracy: 0.5175

Epoch 5: val_accuracy did not improve from 0.53194

491/491 [==============================] - 18s 36ms/step - loss: 1.5667 - accuracy: 0.5174 - val_loss: 1.6755 - val_accuracy: 0.4786

Epoch 6/20

490/491 [==============================>.] - ETA: 0s - loss: 1.4932 - accuracy: 0.5335

Epoch 6: val_accuracy improved from 0.53194 to 0.53323, saving model to v0_best_model.h5

491/491 [==============================] - 18s 36ms/step - loss: 1.4926 - accuracy: 0.5338 - val_loss: 1.5070 - val_accuracy: 0.5332

Epoch 7/20

490/491 [==============================>.] - ETA: 0s - loss: 1.4229 - accuracy: 0.5524

Epoch 7: val_accuracy did not improve from 0.53323

491/491 [==============================] - 19s 38ms/step - loss: 1.4234 - accuracy: 0.5522 - val_loss: 1.5312 - val_accuracy: 0.5276

Epoch 8/20

491/491 [==============================] - ETA: 0s - loss: 1.4009 - accuracy: 0.5576

Epoch 8: val_accuracy improved from 0.53323 to 0.57805, saving model to v0_best_model.h5

491/491 [==============================] - 18s 37ms/step - loss: 1.4009 - accuracy: 0.5576 - val_loss: 1.4830 - val_accuracy: 0.5781

Epoch 9/20

490/491 [==============================>.] - ETA: 0s - loss: 1.3369 - accuracy: 0.5727

Epoch 9: val_accuracy did not improve from 0.57805

491/491 [==============================] - 19s 39ms/step - loss: 1.3366 - accuracy: 0.5729 - val_loss: 1.3956 - val_accuracy: 0.5595

Epoch 10/20

491/491 [==============================] - ETA: 0s - loss: 1.3139 - accuracy: 0.5786

Epoch 10: val_accuracy improved from 0.57805 to 0.58733, saving model to v0_best_model.h5

491/491 [==============================] - 20s 40ms/step - loss: 1.3139 - accuracy: 0.5786 - val_loss: 1.3117 - val_accuracy: 0.5873

Epoch 11/20

491/491 [==============================] - ETA: 0s - loss: 1.2522 - accuracy: 0.5953

Epoch 11: val_accuracy did not improve from 0.58733

491/491 [==============================] - 18s 37ms/step - loss: 1.2522 - accuracy: 0.5953 - val_loss: 1.3686 - val_accuracy: 0.5842

Epoch 12/20

491/491 [==============================] - ETA: 0s - loss: 1.2725 - accuracy: 0.5915

Epoch 12: val_accuracy improved from 0.58733 to 0.61231, saving model to v0_best_model.h5

491/491 [==============================] - 18s 37ms/step - loss: 1.2725 - accuracy: 0.5915 - val_loss: 1.2105 - val_accuracy: 0.6123

Epoch 13/20

491/491 [==============================] - ETA: 0s - loss: 1.2809 - accuracy: 0.5857

Epoch 13: val_accuracy did not improve from 0.61231

491/491 [==============================] - 18s 37ms/step - loss: 1.2809 - accuracy: 0.5857 - val_loss: 1.3938 - val_accuracy: 0.5732

Epoch 14/20

490/491 [==============================>.] - ETA: 0s - loss: 1.2728 - accuracy: 0.5920

Epoch 14: val_accuracy did not improve from 0.61231

491/491 [==============================] - 18s 37ms/step - loss: 1.2735 - accuracy: 0.5917 - val_loss: 1.2458 - val_accuracy: 0.5925

Epoch 15/20

490/491 [==============================>.] - ETA: 0s - loss: 1.2531 - accuracy: 0.5934

Epoch 15: val_accuracy did not improve from 0.61231

491/491 [==============================] - 18s 38ms/step - loss: 1.2536 - accuracy: 0.5933 - val_loss: 1.2734 - val_accuracy: 0.5791

Epoch 16/20

491/491 [==============================] - ETA: 0s - loss: 1.1910 - accuracy: 0.6140

Epoch 16: val_accuracy did not improve from 0.61231

491/491 [==============================] - 20s 40ms/step - loss: 1.1910 - accuracy: 0.6140 - val_loss: 1.2694 - val_accuracy: 0.5971

Epoch 17/20

491/491 [==============================] - ETA: 0s - loss: 1.1740 - accuracy: 0.6174

Epoch 17: val_accuracy improved from 0.61231 to 0.61721, saving model to v0_best_model.h5

491/491 [==============================] - 19s 39ms/step - loss: 1.1740 - accuracy: 0.6174 - val_loss: 1.2621 - val_accuracy: 0.6172

Epoch 18/20

490/491 [==============================>.] - ETA: 0s - loss: 1.1899 - accuracy: 0.6228

Epoch 18: val_accuracy did not improve from 0.61721

491/491 [==============================] - 18s 37ms/step - loss: 1.1903 - accuracy: 0.6229 - val_loss: 1.4620 - val_accuracy: 0.5559

Epoch 19/20

491/491 [==============================] - ETA: 0s - loss: 1.1814 - accuracy: 0.6224

Epoch 19: val_accuracy improved from 0.61721 to 0.63524, saving model to v0_best_model.h5

491/491 [==============================] - 18s 38ms/step - loss: 1.1814 - accuracy: 0.6224 - val_loss: 1.1185 - val_accuracy: 0.6352

Epoch 20/20

491/491 [==============================] - ETA: 0s - loss: 1.0835 - accuracy: 0.6445

Epoch 20: val_accuracy did not improve from 0.63524

491/491 [==============================] - 18s 38ms/step - loss: 1.0835 - accuracy: 0.6445 - val_loss: 1.2663 - val_accuracy: 0.6180

Fine tuned model:

Classification Report:

	precision	recall	f1-score	support
0	0.99	0.99	0.99	95
1	0.97	0.90	0.94	82
2	0.87	0.93	0.90	98
3	0.94	0.96	0.95	82
4	0.88	0.88	0.88	96
5	0.87	0.93	0.90	91
6	0.98	0.99	0.99	116
7	0.91	0.78	0.84	88
8	0.90	0.94	0.92	97
9	0.98	0.97	0.97	87
10	0.99	1.00	0.99	94
11	0.98	0.95	0.96	85
12	1.00	0.99	1.00	102
13	1.00	1.00	1.00	77
14	0.98	0.97	0.97	118
15	0.92	0.94	0.93	94
16	0.99	0.97	0.98	117
17	0.96	0.99	0.97	91
18	0.96	0.87	0.91	85
19	0.91	0.98	0.94	98
20	0.96	0.91	0.94	81
21	0.87	0.94	0.90	77
22	0.97	0.91	0.94	102
23	0.96	0.82	0.89	100
24	0.91	0.95	0.93	85
25	0.92	0.90	0.91	87
26	0.96	0.97	0.97	76
27	0.89	0.97	0.93	77
28	0.87	0.89	0.88	83

29	0.93	0.88	0.91	98
30	0.92	0.98	0.95	103
31	0.88	0.92	0.90	113
32	0.96	1.00	0.98	79
33	0.96	0.99	0.98	108
34	0.99	0.99	0.99	92
35	0.99	0.98	0.98	92
36	0.99	0.96	0.97	98
37	0.98	0.99	0.98	86
38	0.99	0.99	0.99	100
39	0.98	0.99	0.99	115
40	0.98	0.92	0.95	91
41	0.96	0.97	0.96	92
42	0.98	0.98	0.98	87
accuracy			0.95	4015
macro avg	0.95	0.95	0.95	4015
weighted avg	0.95	0.95	0.95	4015

History:

Epoch 1/20

502/502 [==============================] - ETA: 0s - loss: 1.9847 - accuracy: 0.4042

Epoch 1: val_accuracy improved from -inf to 0.56855, saving model to finetuned_modelv2.h5

502/502 [==============================] - 46s 90ms/step - loss: 1.9847 - accuracy: 0.4042 - val_loss: 1.3553 - val_accuracy: 0.5686

Epoch 2/20

502/502 [==============================] - ETA: 0s - loss: 1.0491 - accuracy: 0.6589

Epoch 2: val_accuracy improved from 0.56855 to 0.71799, saving model to finetuned_modelv2.h5

502/502 [==============================] - 49s 97ms/step - loss: 1.0491 - accuracy: 0.6589 - val_loss: 0.8973 - val_accuracy: 0.7180

Epoch 3/20

502/502 [==============================] - ETA: 0s - loss: 0.7000 - accuracy: 0.7694

Epoch 3: val_accuracy improved from 0.71799 to 0.78214, saving model to finetuned_modelv2.h5

502/502 [==============================] - 50s 100ms/step - loss: 0.7000 - accuracy: 0.7694 - val_loss: 0.6678 - val_accuracy: 0.7821

Epoch 4/20

502/502 [==============================] - ETA: 0s - loss: 0.4963 - accuracy: 0.8351

Epoch 4: val_accuracy improved from 0.78214 to 0.83723, saving model to finetuned_modelv2.h5

502/502 [==============================] - 50s 100ms/step - loss: 0.4963 - accuracy: 0.8351 - val_loss: 0.5117 - val_accuracy: 0.8372

Epoch 5/20

502/502 [==============================] - ETA: 0s - loss: 0.3703 - accuracy: 0.8772

Epoch 5: val_accuracy improved from 0.83723 to 0.86038, saving model to finetuned_modelv2.h5

502/502 [==============================] - 49s 97ms/step - loss: 0.3703 - accuracy: 0.8772 - val_loss: 0.4306 - val_accuracy: 0.8604

Epoch 6/20

502/502 [==============================] - ETA: 0s - loss: 0.2802 - accuracy: 0.9060

Epoch 6: val_accuracy improved from 0.86038 to 0.87119, saving model to finetuned_modelv2.h5

502/502 [==============================] - 50s 99ms/step - loss: 0.2802 - accuracy: 0.9060 - val_loss: 0.4072 - val_accuracy: 0.8712

Epoch 7/20

502/502 [==============================] - ETA: 0s - loss: 0.2390 - accuracy: 0.9195

Epoch 7: val_accuracy improved from 0.87119 to 0.90767, saving model to finetuned_modelv2.h5

502/502 [==============================] - 50s 100ms/step - loss: 0.2390 - accuracy: 0.9195 - val_loss: 0.3121 - val_accuracy: 0.9077

Epoch 8/20

502/502 [==============================] - ETA: 0s - loss: 0.2070 - accuracy: 0.9294

Epoch 8: val_accuracy did not improve from 0.90767

502/502 [==============================] - 51s 101ms/step - loss: 0.2070 - accuracy: 0.9294 - val_loss: 0.5317 - val_accuracy: 0.8463

Epoch 9/20

502/502 [==============================] - ETA: 0s - loss: 0.1755 - accuracy: 0.9389

Epoch 9: val_accuracy improved from 0.90767 to 0.91522, saving model to finetuned_modelv2.h5

502/502 [==============================] - 50s 100ms/step - loss: 0.1755 - accuracy: 0.9389 - val_loss: 0.3241 - val_accuracy: 0.9152

Epoch 10/20

502/502 [==============================] - ETA: 0s - loss: 0.1572 - accuracy: 0.9470

Epoch 10: val_accuracy improved from 0.91522 to 0.91623, saving model to finetuned_modelv2.h5

502/502 [==============================] - 50s 99ms/step - loss: 0.1572 - accuracy: 0.9470 - val_loss: 0.3200 - val_accuracy: 0.9162

Epoch 11/20

502/502 [==============================] - ETA: 0s - loss: 0.1299 - accuracy: 0.9561

Epoch 11: val_accuracy improved from 0.91623 to 0.92126, saving model to finetuned_modelv2.h5

502/502 [==============================] - 48s 96ms/step - loss: 0.1299 - accuracy: 0.9561 - val_loss: 0.2804 - val_accuracy: 0.9213

Epoch 12/20

502/502 [==============================] - ETA: 0s - loss: 0.1164 - accuracy: 0.9612

Epoch 12: val_accuracy did not improve from 0.92126

502/502 [==============================] - 51s 102ms/step - loss: 0.1164 - accuracy: 0.9612 - val_loss: 0.3028 - val_accuracy: 0.9190

Epoch 13/20

502/502 [==============================] - ETA: 0s - loss: 0.1127 - accuracy: 0.9623

Epoch 13: val_accuracy improved from 0.92126 to 0.93585, saving model to finetuned_modelv2.h5

502/502 [==============================] - 50s 100ms/step - loss: 0.1127 - accuracy: 0.9623 - val_loss: 0.2692 - val_accuracy: 0.9358

Epoch 14/20

502/502 [==============================] - ETA: 0s - loss: 0.1037 - accuracy: 0.9658

Epoch 14: val_accuracy improved from 0.93585 to 0.94340, saving model to finetuned_modelv2.h5

502/502 [==============================] - 51s 102ms/step - loss: 0.1037 - accuracy: 0.9658 - val_loss: 0.2371 - val_accuracy: 0.9434

Epoch 15/20

502/502 [==============================] - ETA: 0s - loss: 0.1039 - accuracy: 0.9657

Epoch 15: val_accuracy did not improve from 0.94340

502/502 [==============================] - 49s 97ms/step - loss: 0.1039 - accuracy: 0.9657 - val_loss: 0.4017 - val_accuracy: 0.9072

Epoch 16/20

502/502 [==============================] - ETA: 0s - loss: 0.0963 - accuracy: 0.9681

Epoch 16: val_accuracy did not improve from 0.94340

502/502 [==============================] - 49s 97ms/step - loss: 0.0963 - accuracy: 0.9681 - val_loss: 0.3140 - val_accuracy: 0.9291

Epoch 17/20

502/502 [==============================] - ETA: 0s - loss: 0.0789 - accuracy: 0.9739

Epoch 17: val_accuracy improved from 0.94340 to 0.94667, saving model to finetuned_modelv2.h5

502/502 [==============================] - 49s 98ms/step - loss: 0.0789 - accuracy: 0.9739 - val_loss: 0.2310 - val_accuracy: 0.9467

Epoch 18/20

502/502 [==============================] - ETA: 0s - loss: 0.0709 - accuracy: 0.9772

Epoch 18: val_accuracy did not improve from 0.94667

502/502 [==============================] - 52s 104ms/step - loss: 0.0709 - accuracy: 0.9772 - val_loss: 0.2722 - val_accuracy: 0.9328

Epoch 19/20

502/502 [==============================] - ETA: 0s - loss: 0.0815 - accuracy: 0.9733

Epoch 19: val_accuracy did not improve from 0.94667

502/502 [==============================] - 49s 98ms/step - loss: 0.0815 - accuracy: 0.9733 - val_loss: 0.2965 - val_accuracy: 0.9288

Epoch 20/20

362/502 [====================>.....] - ETA: 14s - loss: 0.0744 - accuracy: 0.9751W

Epoch 20: val_accuracy did not improve from 0.94667

502/502 [==============================] - 40s 79ms/step - loss: 0.0744 - accuracy: 0.9751 - val_loss: 0.2765 - val_accuracy: 0.9353