Configuration Manual

MSc Research Project
Data Analytics

Maaz Ahmad
Student ID: 21134308

School of Computing
National College of Ireland

Supervisor: Michael Bradford

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Maaz Ahmad
Student ID:	21134308
Programme:	Data Analytics
Year:	2023
Module:	MSc Research Project
Supervisor:	Michael Bradford
Submission Due Date:	25/04/2023
Project Title:	Configuration Manual
Word Count:	1997
Page Count:	25

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Maaz Ahmad
Date:	27th May 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Maaz Ahmad
21134308

1 Introduction

This manual's purpose is to help users set up their own machines and get the results they want. The associated documents contain thorough information about the required hardware and software to build the environment. The manual includes code snippets, graphics from exploratory data analysis, and model assessments.

2 Environment

The environment needed to run the code solution is thoroughly described in this section. The setup for the Google Colaboratory, the required Python libraries and packages, and other crucial components are all covered in this section. This information is essential for making sure the code solution runs without a hitch and for making sure users can easily reproduce the study's findings.

2.1 Hardware Required

The computational tasks for the current research project were carried out using a specific hardware specification, though other requirements were also taken into account. The hardware specifications are compiled in Table 1 to give a thorough overview of the system configuration. Figure 1 also provides a thorough illustration of the machine's system configuration, improving understanding of the used hardware components.

Table 1: Hardware Specification

Hardware	Used in this Project	Alternative
System	VivoBook ASUS	Any Windows/Mac/Linux machine
OS	Windows 11	Any Windows/Macos/Linux Distribution
RAM	8GB	>4GB
Processor	Intel(R) Core(TM) i5-8265U	Any Intel/AMD/Apple Silicon
Hard Disk	258GB	>10GB
GPU	Nvidia	Any integrated/Nvidia/AMD

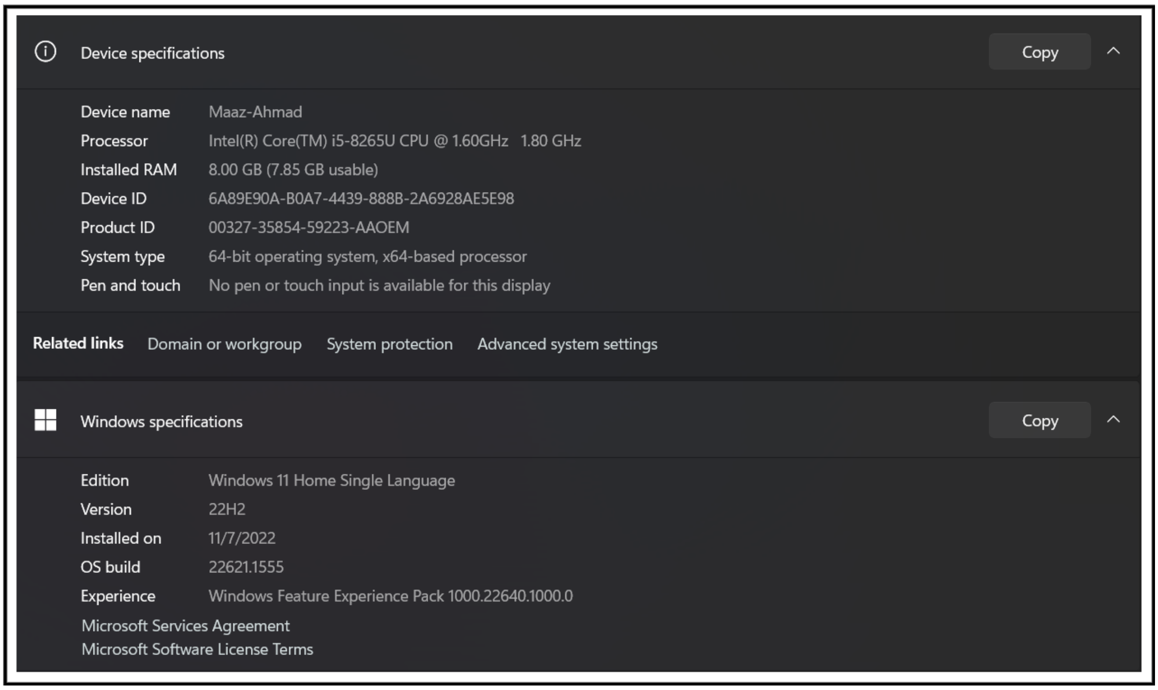


Figure 1: Hardware Specification

2.2 Setting Up Google Colab Environment

Python was the programming language used for the research project, and Google Colab was used to carry it out, as shown in Figure 2. Colab, a hosted Jupyter notebook service, provides an easy-to-use platform for running Python code directly in a web browser, making it an excellent choice for deep learning and machine learning tasks. For a limited time, Colab offers free access to computing resources, including GPUs, without the need for configuration. However, users must upgrade to Colab Pro in order to access TPU.

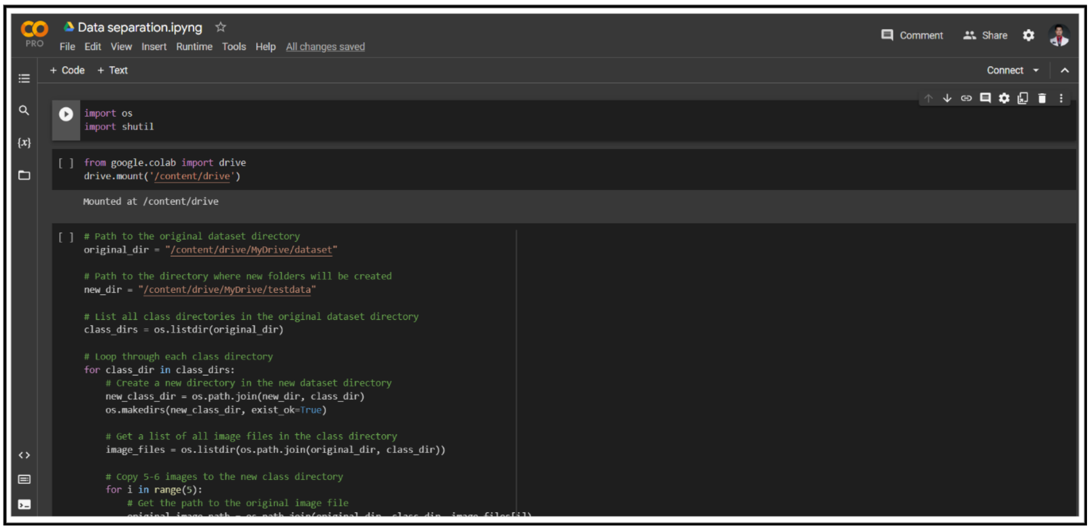


Figure 2: Google Colab

The dataset [1] was taken from Kaggle and then downloaded to local disk and then extracted from zip to normal file shown in Figure 3 and then uploaded to google drive. After it Successful upload with the help of google colab mount we can mount the data to google colab as shown in Figure 4.

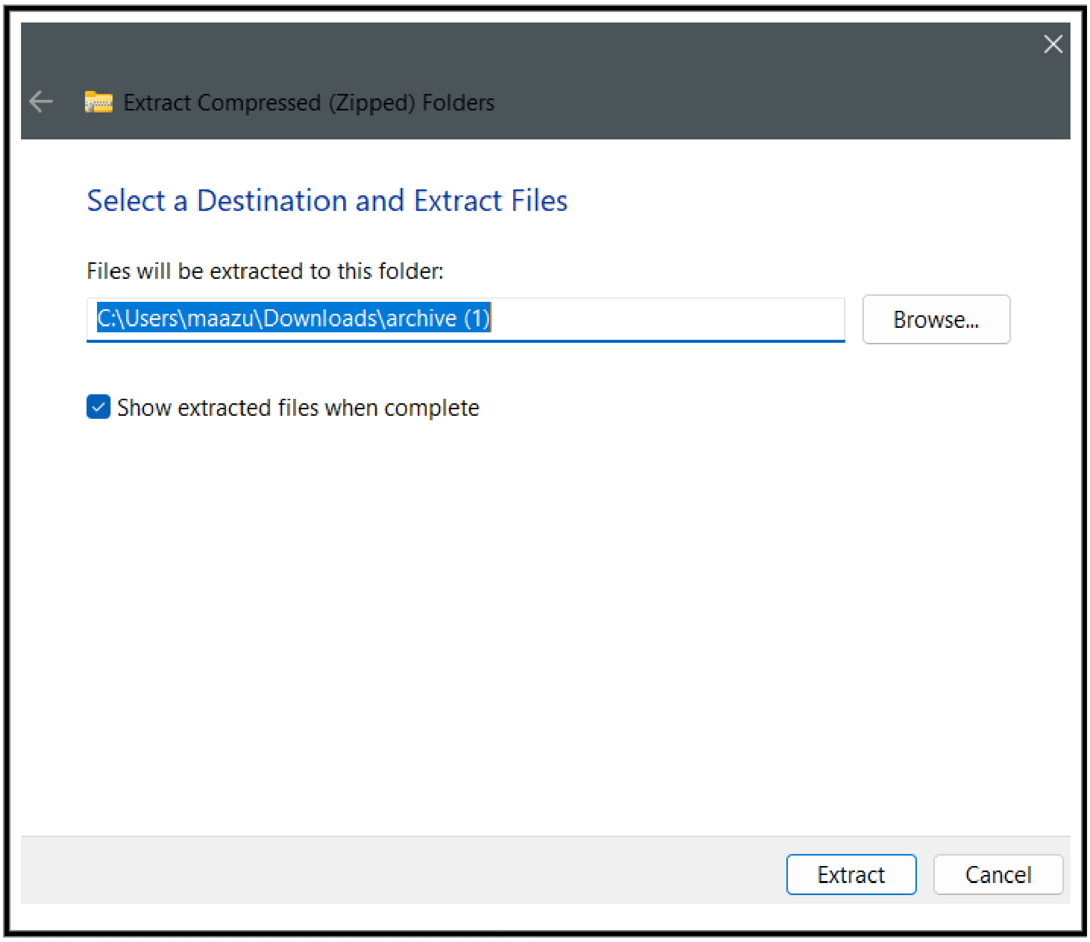


Figure 3: Data Extraction

[1] https://www.kaggle.com/datasets/shrutisaxena/yoga-pose-image-classification-dataset

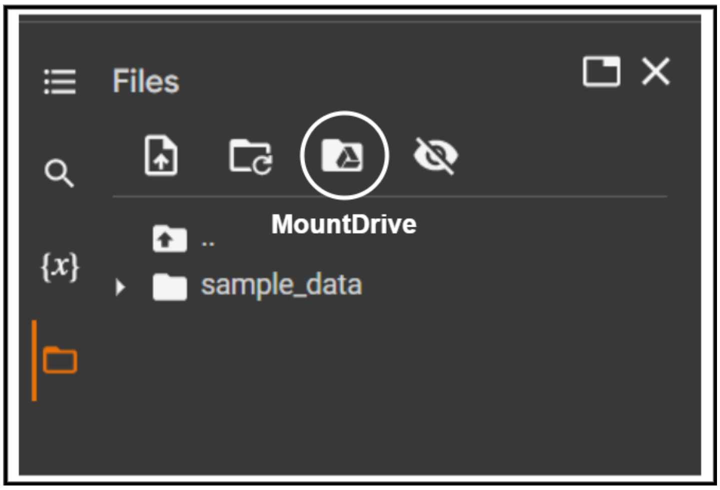


Figure 4: Mount google drive to Colab

As shown in Figure 5, after mounting the drive, a runtime must be assigned, and TPU must be chosen for quicker processing. The candidate used Google Colab Pro, which came with a premium TPU, but standard GPUs can also be used and do just fine.

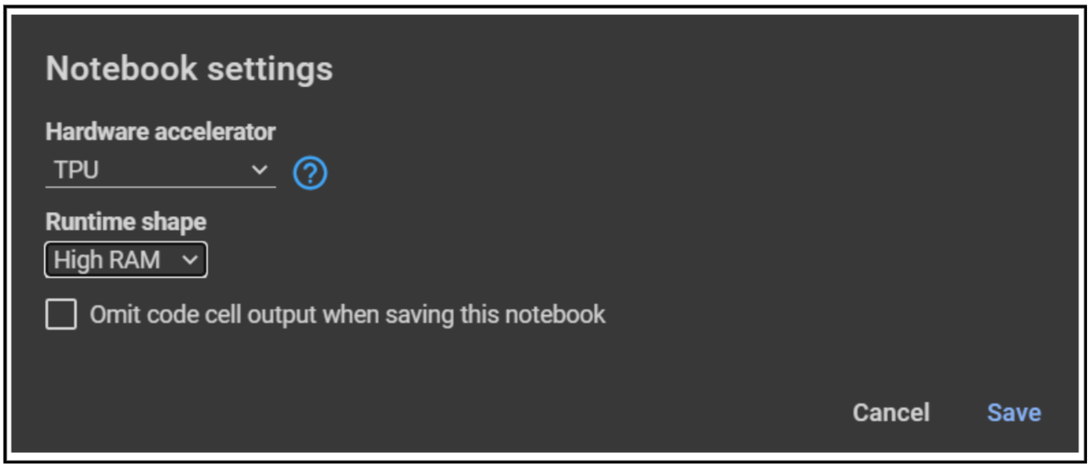


Figure 5: Assign Runtime

3 Implementation

The research project's data acquisition, model construction, training, results, and visualizations are all covered in detail in this section, along with step-by-step instructions for reproducing the study using the provided code.

3.1 Reading the Data

After mounting the data to the google colab. The directories for training and testing data are assigned to fetch data seen in Figure 6.

```python
original_dir = "/content/drive/MyDrive/dataset"
new_dir = "/content/drive/MyDrive/testdata"
```

Figure 6: Assign the directories to fetch data

3.2 Data Preparation

The process of Data Preparation starts from the first notebook name "Data Separation" in which the data is created by taking 5 or 6 images from each class and stored into a new file called test data after the data is copied the same image will be removed from old data-set which can be seen in Figure 7.

```python
for class_dir in class_dirs:

    new_class_dir = os.path.join(new_dir, class_dir)
    os.makedirs(new_class_dir, exist_ok=True)

    image_files = os.listdir(os.path.join(original_dir, class_dir))

    for i in range(5):

        original_image_path = os.path.join(original_dir, class_dir, image_files[i])

        new_image_path = os.path.join(new_class_dir, image_files[i])

        shutil.copy(original_image_path, new_image_path)

        os.remove(original_image_path)
```

Figure 7: Splitting the data into to parts

3.3 Model Training of ResNet-50

For ResNet-50 and DenseNet-121 training open a new colab notebook and start by importing the libraries as show in Figure8. After importing all the libraries initialising the ResNet-50 model as shown in Figure9

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Figure 8: Importing the libraries

.

```
[ ] resnet = keras.applications.resnet.ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    for layer in resnet.layers:
      layer.trainable = False

    x = resnet.output
    x = keras.layers.GlobalAveragePooling2D()(x)
    x = keras.layers.Dense(512, activation='relu')(x)
    x = keras.layers.Dropout(0.5)(x)
    predictions = keras.layers.Dense(107, activation='softmax')(x)
    resnet_model = keras.models.Model(inputs=resnet.input, outputs=predictions)
```

Figure 9: Initialising the ResNet-50 model

The model must then be fitted to the training set of data after the model architecture has been established. The model in this project was trained over 75 epochs.(Figure 10)

```
resnet_history = resnet_model.fit(train_data, validation_data=validation_data, epochs=100, steps_per_epoch=len(train_data), validation_steps=len(validation_data))
print("ResNet Best Accuracy:", max(resnet_history.history['val_accuracy']))

Epoch 1/100
138/138 [==============================] - 137s 964ms/step - loss: 4.7238 - accuracy: 0.0147 - val_loss: 4.6508 - val_accuracy: 0.0200
Epoch 2/100
138/138 [==============================] - 132s 952ms/step - loss: 4.6514 - accuracy: 0.0138 - val_loss: 4.6295 - val_accuracy: 0.0267
Epoch 3/100
138/138 [==============================] - 131s 946ms/step - loss: 4.6259 - accuracy: 0.0202 - val_loss: 4.6043 - val_accuracy: 0.0219
Epoch 4/100
138/138 [==============================] - 130s 940ms/step - loss: 4.6101 - accuracy: 0.0195 - val_loss: 4.5806 - val_accuracy: 0.0381
Epoch 5/100
138/138 [==============================] - 131s 945ms/step - loss: 4.5892 - accuracy: 0.0200 - val_loss: 4.5493 - val_accuracy: 0.0334
Epoch 6/100
138/138 [==============================] - 130s 941ms/step - loss: 4.5615 - accuracy: 0.0286 - val_loss: 4.5373 - val_accuracy: 0.0429
Epoch 7/100
138/138 [==============================] - 130s 943ms/step - loss: 4.5349 - accuracy: 0.0272 - val_loss: 4.5030 - val_accuracy: 0.0458
Epoch 8/100
138/138 [==============================] - 130s 943ms/step - loss: 4.5210 - accuracy: 0.0245 - val_loss: 4.4826 - val_accuracy: 0.0410
Epoch 9/100
138/138 [==============================] - 130s 942ms/step - loss: 4.4867 - accuracy: 0.0320 - val_loss: 4.4444 - val_accuracy: 0.0524
Epoch 10/100
138/138 [==============================] - 132s 954ms/step - loss: 4.4786 - accuracy: 0.0309 - val_loss: 4.4141 - val_accuracy: 0.0591
```

Figure 10: Running Training for ResNet-50

To print the accuracy of ResNet-50 model graph first we import Matplot libraries and use the accuracy and validation accuracy to print the graph as show in Figure11. After the code is executed a graph will pop-up as shown in Figure 12.

```
import matplotlib.pyplot as plt

plt.plot(resnet_history.history['accuracy'])
plt.plot(resnet_history.history['val_accuracy'])
plt.title('ResNet Accuracy')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

Figure 11: Code to plot graph for ResNet-50

Figure 12: Accuracy Graph for ResNet-50

3.4 Model Training of DenesNet-121

The pre-trained model for DenesNet-121 is then initialised(Figure 13).

```python
densenet = keras.applications.densenet.DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
for layer in densenet.layers:
  layer.trainable = False

x = densenet.output
x = keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dense(512, activation='relu')(x)
x = keras.layers.Dropout(0.5)(x)
predictions = keras.layers.Dense(107, activation='softmax')(x)
densenet_model = keras.models.Model(inputs=densenet.input, outputs=predictions)
```

Figure 13: Initialising the DenesNet-121 model

The data is then put into the model to train with DenesNet-121 for 75 epochs to give an output like shown in Figure 14.

```
[ ] densenet_history = densenet_model.fit(train_data, validation_data=validation_data, epochs=100, steps_per_epoch=len(train_data), validation_steps=len(validation_data))
    print("DenseNet Best Accuracy:", max(densenet_history.history['val_accuracy']))

    Epoch 1/100
    138/138 [==============================] - 1059s 8s/step - loss: 4.4100 - accuracy: 0.0563 - val_loss: 3.8158 - val_accuracy: 0.1649
    Epoch 2/100
    138/138 [==============================] - 114s 828ms/step - loss: 3.5817 - accuracy: 0.1548 - val_loss: 3.0546 - val_accuracy: 0.2850
    Epoch 3/100
    138/138 [==============================] - 114s 827ms/step - loss: 2.9815 - accuracy: 0.2448 - val_loss: 2.6563 - val_accuracy: 0.3298
    Epoch 4/100
    138/138 [==============================] - 113s 821ms/step - loss: 2.6009 - accuracy: 0.3286 - val_loss: 2.3602 - val_accuracy: 0.4004
    Epoch 5/100
    138/138 [==============================] - 113s 819ms/step - loss: 2.3718 - accuracy: 0.3690 - val_loss: 2.2355 - val_accuracy: 0.4395
    Epoch 6/100
    138/138 [==============================] - 114s 823ms/step - loss: 2.1926 - accuracy: 0.4025 - val_loss: 2.1179 - val_accuracy: 0.4347
    Epoch 7/100
    138/138 [==============================] - 114s 824ms/step - loss: 2.0336 - accuracy: 0.4407 - val_loss: 2.0017 - val_accuracy: 0.4681
    Epoch 8/100
    138/138 [==============================] - 113s 819ms/step - loss: 1.9469 - accuracy: 0.4634 - val_loss: 1.9421 - val_accuracy: 0.4719
    Epoch 9/100
    138/138 [==============================] - 113s 821ms/step - loss: 1.8370 - accuracy: 0.4774 - val_loss: 1.8789 - val_accuracy: 0.4805
    Epoch 10/100
    138/138 [==============================] - 114s 826ms/step - loss: 1.7325 - accuracy: 0.5155 - val_loss: 1.8431 - val_accuracy: 0.4623
```

Figure 14: Running Training for DenesNet-121

```
plt.plot(densenet_history.history['accuracy'])
plt.plot(densenet_history.history['val_accuracy'])
plt.title('DenseNet Accuracy')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt
```

Figure 15: Plotting graph for DenesNet-121

Then the accuracy and validation accuracy is used to plot an accuracy graph for DenesNet-121 the code can be seen in Figure 15. Whereas, the graph can be seen in Figure 16.

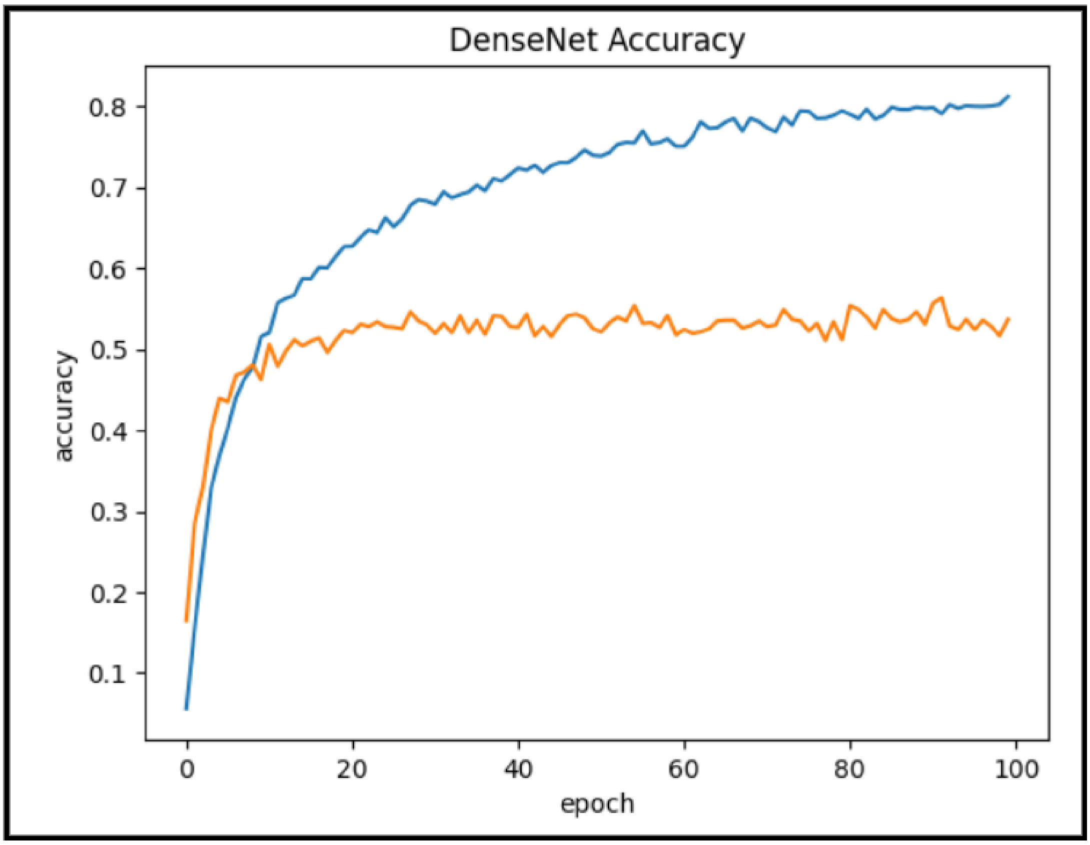


Figure 16: Accuracy Graph for DenesNet-121

At the end of the training make an folder in your drive as trainedcnn then save the weight of both the model ResNet-50 and DenesNet-121 is save in to the drive in a trainedcnn folder by using the model.save command as shown in Figure 17.

```
[ ] resnet_model.save("/content/drive/MyDrive/trainedcnn/resnetmodel.h5")

[ ] densenet_model.save("/content/drive/MyDrive/trainedcnn/densenetmodel.h5")
```

Figure 17: Saving trained ResNet-50 and DenesNet-121

3.5 Model Training of Custom CNN without Augmentation

For the Custom CNN model I used a new Colab notebook. Where you can start by installing Keras Tuner pip file refer Figure 18 for this. And then you can import all the important libraries as shown in Figure19.

```
pip install keras-tuner

Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting keras-tuner
  Downloading keras_tuner-1.3.5-py3-none-any.whl (176 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 176.1/176.1 kB 4.6 MB/s eta 0:00:00
Requirement already satisfied: packaging in /usr/local/lib/python3.9/dist-packages (from keras-tuner) (23.1)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (from keras-tuner) (2.27.1)
Collecting kt-legacy
  Downloading kt_legacy-1.0.5-py3-none-any.whl (9.6 kB)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner) (3.4)
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner) (2.0.12)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner) (2022.12.7)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner) (1.26.15)
Installing collected packages: kt-legacy, keras-tuner
Successfully installed keras-tuner-1.3.5 kt-legacy-1.0.5
```

Figure 18: Installing Keras Tuner

```
[ ] import os
    import tensorflow as tf
    from tensorflow import keras
    from tensorflow.keras import layers
    from tensorflow.keras.preprocessing.image import ImageDataGenerator
    from kerastuner.tuners import RandomSearch
    from kerastuner.engine.hyperparameters import HyperParameters
```

Figure 19: Importing libraries for Custom CNN model

Then assign the dataset path for Kear Tuner(Figure 20).

```python
dataset_path = '/content/drive/MyDrive/dataset'
train_generator = datagen.flow_from_directory(
    dataset_path,
    target_size=input_shape[:2],
    batch_size=32,
    subset='training')
val_generator = datagen.flow_from_directory(
    dataset_path,
    target_size=input_shape[:2],
    batch_size=32,
    subset='validation')
```

Figure 20: Assign the directories to fetch data for Keras Tuner

And then you can build the model with different hyperparameter in my case it 3 and 7 and low and high for that you can refer the Figure 21.

```python
def build_model(hp):
    model = keras.Sequential()

    for i in range(hp.Int('conv_layers', 3, 7)):
        filters = hp.Choice('filters_{}'.format(i), [32, 64, 128, 256])
        kernel_size = hp.Choice('kernel_size_{}'.format(i), [3, 5])
        activation = hp.Choice('activation_{}'.format(i), ['relu', 'tanh'])
        model.add(layers.Conv2D(filters=filters, kernel_size=kernel_size,
                                activation=activation, input_shape=input_shape))
        model.add(layers.MaxPooling2D())

    model.add(layers.Flatten())

    for i in range(hp.Int('dense_layers', 1, 3)):
        units = hp.Choice('units_{}'.format(i), [128, 256, 512, 1024])
        activation = hp.Choice('dense_activation_{}'.format(i), ['relu', 'tanh'])
        model.add(layers.Dense(units=units, activation=activation))

    model.add(layers.Dense(107, activation='softmax'))

    model.compile(optimizer=keras.optimizers.Adam(hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4])),
                  loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

Figure 21: Initialising the Custom CNN model with Keras Tuner

```
[ ] tuner = RandomSearch(
        build_model,
        objective='val_accuracy',
        max_trials=10,
        directory='tuner_dir',
        project_name='yoga_pose_recognition')
```

```
tuner.search(train_generator, validation_data=val_generator, epochs=10)

Trial 10 Complete [00h 15m 42s]
val_accuracy: 0.016205910593271255

Best val_accuracy So Far: 0.4385128617286682
Total elapsed time: 04h 57m 04s
```

Figure 22: Searching for best Custom CNN model with Keras Tuner

Then with the help of Keras Tuner run the model for 10 epochs for 10 trial after completion of to the trial you will find the best fit model for the data set cab be seen in Figure 22. After finding the best fit model put the same data to get trained for 75 epochs to find the accuracy and validation accuracy like Figure23.

```
best_model_old_history = best_model.fit(train_generator, epochs=75, validation_data=val_generator)

Epoch 2/75
138/138 [==============================] - 440s 3s/step - loss: 0.0610 - accuracy: 0.9725 - val_loss: 2.8884 - val_accuracy: 0.4290
Epoch 3/75
138/138 [==============================] - 440s 3s/step - loss: 0.0638 - accuracy: 0.9737 - val_loss: 2.8793 - val_accuracy: 0.4299
Epoch 4/75
138/138 [==============================] - 440s 3s/step - loss: 0.0552 - accuracy: 0.9732 - val_loss: 2.9018 - val_accuracy: 0.4433
Epoch 5/75
138/138 [==============================] - 440s 3s/step - loss: 0.0534 - accuracy: 0.9712 - val_loss: 2.9524 - val_accuracy: 0.4471
Epoch 6/75
138/138 [==============================] - 440s 3s/step - loss: 0.0492 - accuracy: 0.9755 - val_loss: 2.9052 - val_accuracy: 0.4500
Epoch 7/75
138/138 [==============================] - 440s 3s/step - loss: 0.0553 - accuracy: 0.9714 - val_loss: 2.9209 - val_accuracy: 0.4528
Epoch 8/75
138/138 [==============================] - 443s 3s/step - loss: 0.0488 - accuracy: 0.9741 - val_loss: 2.9686 - val_accuracy: 0.4404
Epoch 9/75
138/138 [==============================] - 453s 3s/step - loss: 0.0485 - accuracy: 0.9739 - val_loss: 2.9989 - val_accuracy: 0.4557
Epoch 10/75
138/138 [==============================] - 450s 3s/step - loss: 0.0471 - accuracy: 0.9739 - val_loss: 2.9787 - val_accuracy: 0.4461
```

Figure 23: Training the Custom CNN without Augmentation

Save the model in trainedcnn file using model.save commandand then print the test accuracy result as shown in Figure 24. You can also plot a accuray graph by using accuracy and validation accuracy refer Figure 25&26.

```
[ ]  best_model.save("/content/drive/MyDrive/trainedcnn/model.h5")
```

```
[ ]  test_loss, test_acc = test.evaluate(val_generator, verbose=2)
     print('Test accuracy:', test_acc)

     33/33 - 16s - loss: 3.3229 - accuracy: 0.4366 - 16s/epoch - 475ms/step
     Test accuracy: 0.4366062879562378
```

Figure 24: Saving trained Custom CNN without Augmentation

```
plt.plot(best_model_old_history.history['accuracy'])
plt.plot(best_model_old_history.history['val_accuracy'])
plt.title('CNN Accuracy With Augmentation')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

Figure 25: Code to plot accuracy graph for Custom CNN without Augmentation

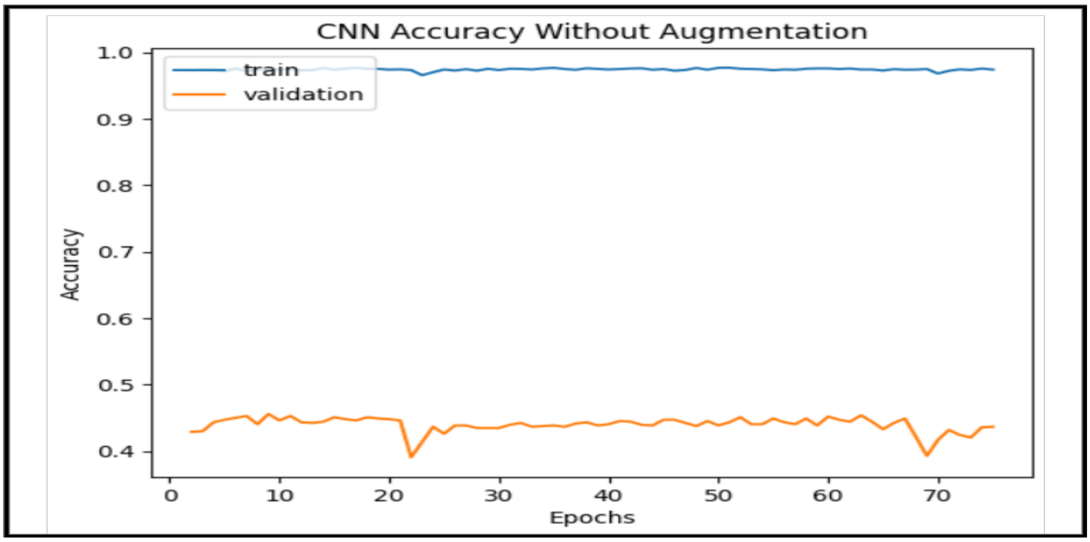


Figure 26: Accuracy graph for Custom CNN without augmentation

Because of multiple classes classification matrix was performed for the Custom CNN model without augmentation. In which we can see Accuracy, Macro avg and weighted avg in the Figure27.

```
from sklearn.metrics import classification_report
y_pred_labels = np.argmax(y_pred, axis=1)
y_true = val_generator.classes
class_names = list(val_generator.class_indices.keys())

print(classification_report(y_true, y_pred_labels, target_names=class_names, digits=4))
```

accuracy			0.0114	1049
macro avg	0.0099	0.0092	0.0094	1049
weighted avg	0.0124	0.0114	0.0118	1049

Figure 27: Classification matrix for Custom CNN without Augmentation

3.6 Model Training of Custom CNN with Augmentation

For the data augmentation part, the data go through flipping, resizing and scaling all of this is done and can be seen in Figure 28.

```
[ ] datagen_augmentation = ImageDataGenerator(rescale=1./255,
                                              width_shift_range = 0.1,
                                              height_shift_range = 0.1,
                                              shear_range = 0.1,
                                              zoom_range= 0.1,
                                              horizontal_flip=True,
                                              fill_mode='nearest',
                                              validation_split=0.2)
```

Figure 28: Initialising data for Custom CNN with Augmentation

```
dataset_path = '/content/drive/MyDrive/dataset'
train_generator_augmentation = datagen.flow_from_directory(
    dataset_path,
    target_size=input_shape[:2],
    batch_size=32,
    subset='training')
val_generator_augmentation = datagen.flow_from_directory(
    dataset_path,
    target_size=input_shape[:2],
    batch_size=32,
    subset='validation')
```

Figure 29: Assign the directories to fetch data for Keras Tuner

Figure 29Then the dataset path is put in for training and validation. Whereas in Figure 30 the data set is loaded from the keras model.load command and then put for training using new augmented data. Where it runs for 75 epochs to get accuracy and validation accuracy.

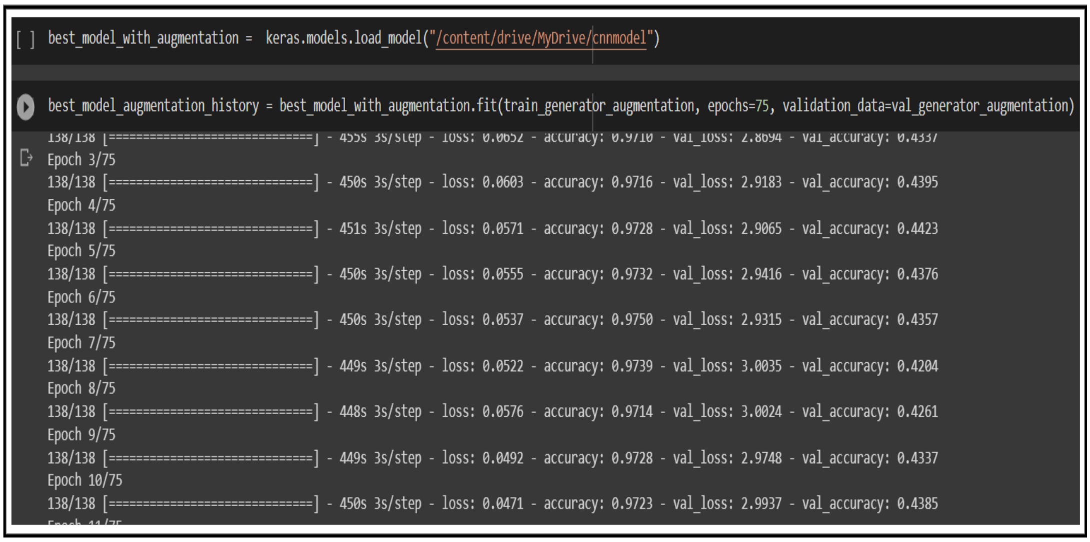


Figure 30: Training model for Custom CNN with augmentation

Figure 31: Saving the trained model of Custom CNN with augmentation

After the model is trained with augmented data we use model.save command to save the weight of model. Figure31 also print the test accuracy for custom CNN model with data augmentation. Which can also be seen in a graphical presentation refer Figure 32 & 33.

```python
plt.plot(best_model_augmentation_history.history['accuracy'])
plt.plot(best_model_augmentation_history.history['val_accuracy'])
plt.title('CNN Accuracy With Augmentation')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

Figure 32: Code to plot accuracy graph of Custom CNN with augmentation

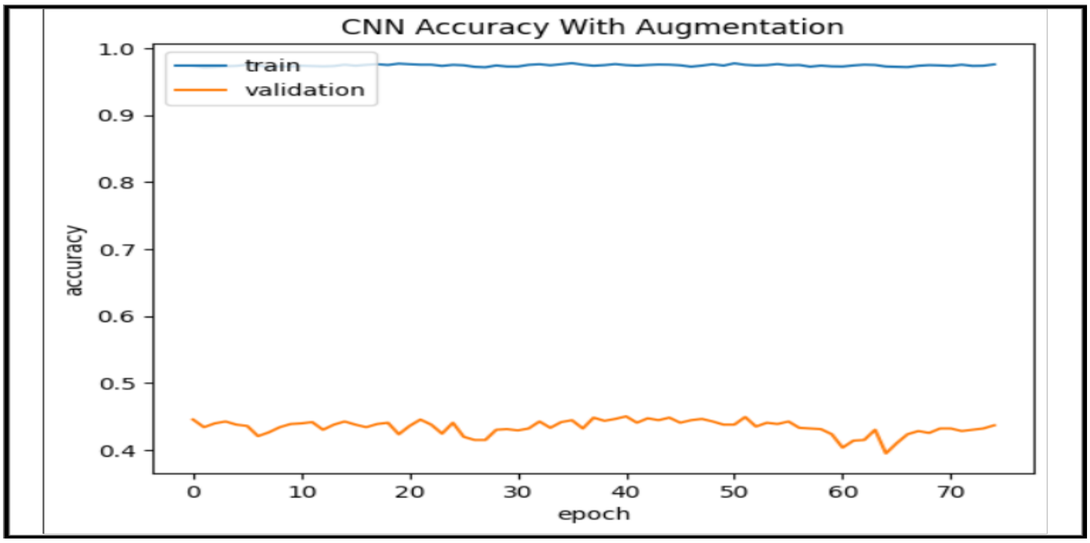
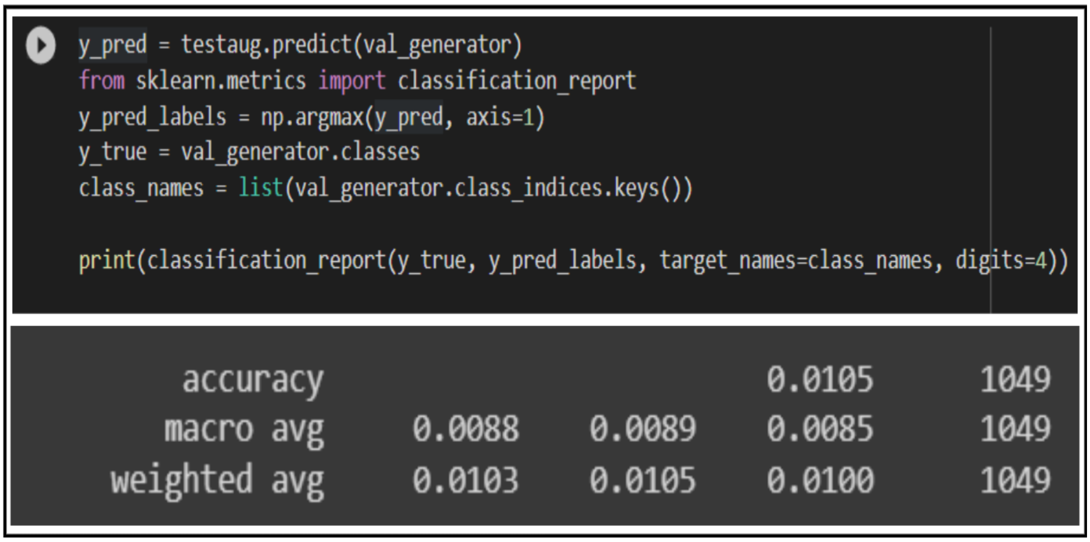


Figure 33: Accuracy graph of Custom CNN with augmentation

The classification matrix is performed in Figure 34 for the Custom CNN model with data augmentation.



```
y_pred = testaug.predict(val_generator)
from sklearn.metrics import classification_report
y_pred_labels = np.argmax(y_pred, axis=1)
y_true = val_generator.classes
class_names = list(val_generator.class_indices.keys())

print(classification_report(y_true, y_pred_labels, target_names=class_names, digits=4))
```

```
    accuracy                          0.0105      1049
   macro avg     0.0088    0.0089    0.0085      1049
weighted avg     0.0103    0.0105    0.0100      1049
```

Figure 34: Classification matrix for Custom CNN with Augmentation

3.7 Testing of ResNet-50, DenesNet-121, Custom CNN without Augmentation and Custom CNN with Augmentation

In this Section, we test all the trained model using the data set we generate for testing mentioned in Section 3.2. Figure 35, 36,37 and 38 hows the accuracy for ResNet-50, DenesNet-121, Custom CNN without augmentation and Custom CNN with augmentation.

```
resnet = keras.models.load_model("/content/drive/MyDrive/trainedcnn/resnetmodel.h5")
```

```
test_loss, test_acc = resnet.evaluate(test_data, verbose=1)
print('Test accuracy on Resnet-50', test_acc)
```

```
17/17 [==============================] - 51s 3s/step - loss: 4.3059 - accuracy: 0.0467
Test accuracy on Resnet-50 0.04672897234559059
```

Figure 35: Test of save model and Result of ResNet-50

```
[ ] densenet = keras.models.load_model("/content/drive/MyDrive/trainedcnn/densenetmodel.h5")

[ ] test_loss, test_acc = densenet.evaluate(test_data, verbose=1)
    print('Test accuracy on DenseNet', test_acc)

    17/17 [==============================] - 22s 1s/step - loss: 1.6681 - accuracy: 0.5439
    Test accuracy on DenseNet 0.5439252257347107
```

Figure 36: Test of save model and Result of DenesNet-121

```
[ ] cnn = keras.models.load_model("/content/drive/MyDrive/trainedcnn/model.h5")

[ ] test_loss, test_acc = cnn.evaluate(test_data, verbose=1)
    print('Test accuracy on CNN without Augmentation', test_acc)

    17/17 [==============================] - 9s 479ms/step - loss: 3.5389 - accuracy: 0.4168
    Test accuracy on CNN without Augmentation 0.4168224334716797
```

Figure 37: Test of save model and Result of Custom CNN without augmentation

```
[ ] cnnaug = keras.models.load_model("/content/drive/MyDrive/trainedcnn/modelaugmenation.h5")

[ ] test_loss, test_acc = cnnaug.evaluate(test_data, verbose=1)
    print('Test accuracy on CNN with Augmentation', test_acc)

    17/17 [==============================] - 8s 473ms/step - loss: 3.7837 - accuracy: 0.4168
    Test accuracy on CNN with Augmentation 0.4168224334716797
```

Figure 38: Test of save model and Result of Custom CNN with augmentation

3.8 Visualisation of Data

To visualise some data first we imported some libraries and the load the data set which you want to visualise as done in Figure 39.

```
[ ] import os
    import matplotlib.pyplot as plt
    from PIL import ImageFile
    import pandas as pd
    ImageFile.LOAD_TRUNCATED_IMAGES = True

    base_dir = '/content/drive/MyDrive/dataset'
    print('Number of post to be predicted: ',len(os.listdir(base_dir)))

    Number of post to be predicted:  107
```

Figure 39: Importing Libraries for Visualization of data

```
[ ] class_dir = '/content/drive/MyDrive/dataset/janu sirsasana'

[ ] fig, axes = plt.subplots(nrows=2, ncols=3,figsize=(10,5), subplot_kw={'xticks':[], 'yticks':[]})
    for i, ax in enumerate(axes.flat):
        ax.imshow(plt.imread(os.path.join(class_dir,os.listdir(class_dir)[i])))
        ax.set_title(os.listdir(base_dir)[1])
    plt.tight_layout()
    plt.show()
```

Figure 40: Initialising data visualisation Class Janu Sirsasana

Then in that dataset you can just identify a class to show some sample image like in Figure 40 I did for janu sirsasana and after you run the code you will se a figure 41 below. Similar with Figure42 & 43.

Figure 41: Image sample of Janu Sirsasana

```
[ ] class_dird = '/content/drive/MyDrive/dataset/bakasana'

[ ] fig, axes = plt.subplots(nrows=2, ncols=3,figsize=(10,5), subplot_kw={'xticks':[], 'yticks':[]})
    for i, ax in enumerate(axes.flat):
        ax.imshow(plt.imread(os.path.join(class_dird,os.listdir(class_dird)[i])))
        ax.set_title(os.listdir(base_dir)[1])
    plt.tight_layout()
    plt.show()
```

Figure 42: Initialising data visualisation Class Bakasana

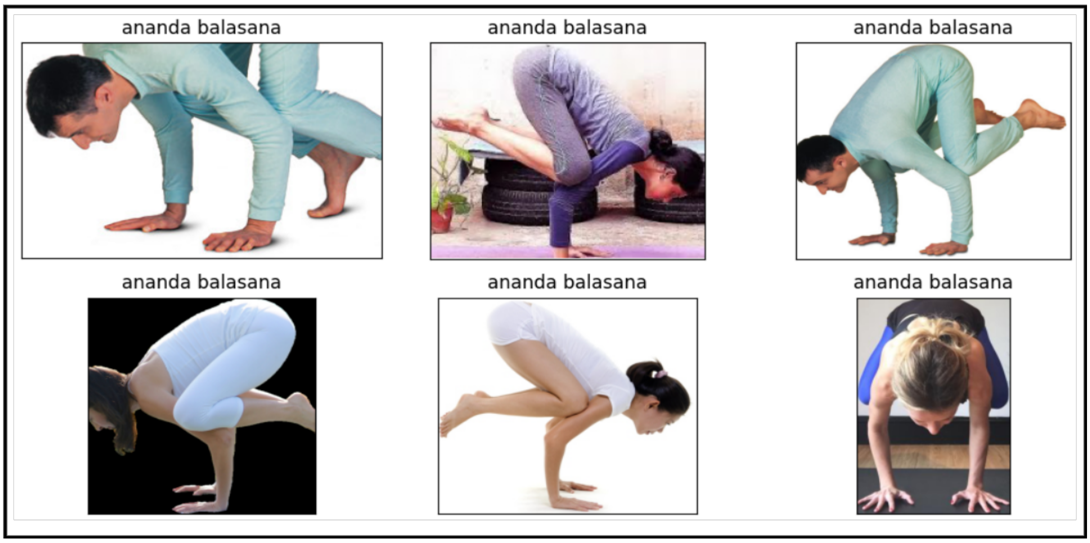


Figure 43: Image sample of Bakasana

At last I wanted to show the number of classes I am using in this model ii visualise the seaborn graph the code and command can be seen in Figure44 and result can be seen in Figure 45.

```
[ ] DF = pd.DataFrame(columns=['class','count'])
    DF['class']=pd.Series([os.listdir(base_dir)[x] for x in range(0,107)])
    DF['count']=pd.Series([len(os.listdir(os.path.join(base_dir,os.listdir(base_dir)[x]))) for x in range(0,107)])

[ ] import seaborn as sns
    plt.figure(figsize=(14,10))
    g=sns.barplot(x='class', y='count',data=DF)
    g.set_xticklabels(g.get_xticklabels(), rotation=90)
    plt.tight_layout()
```

Figure 44: Initialising data visualisation for 107 class

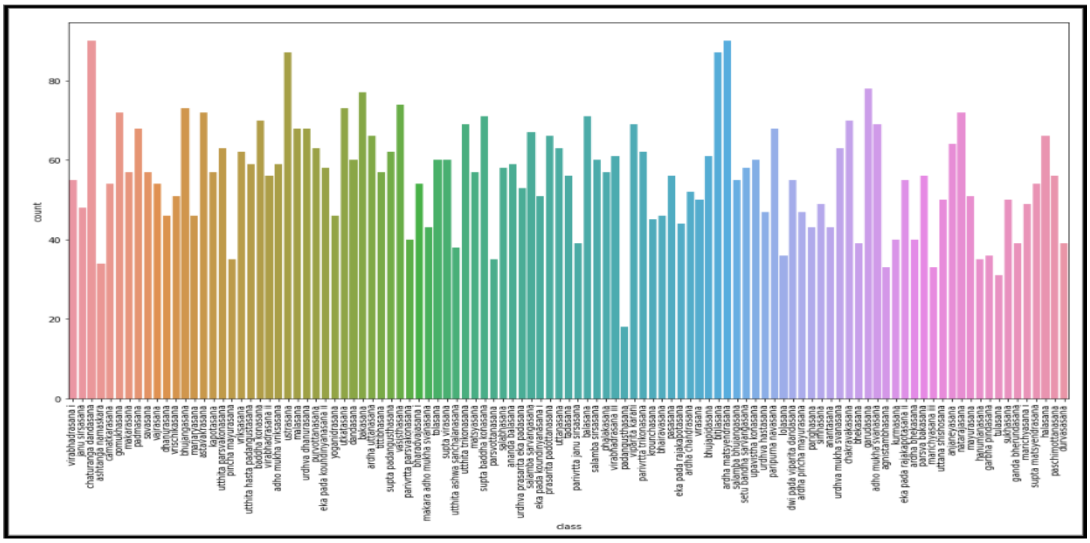


Figure 45: Visualisation of 107 Classes in histogram

4 Conclusion

Users who follow the instructions in the preceding sections can successfully replicate and use the codebase for this research project. This will make it possible to comprehend the project's inner workings better and to make contributions to its future development. Similar results will be obtained by following the detailed instructions provided for data collection, model building, training, and results visualization. This guide aims to make it easier to replicate and build on existing research, which is crucial for the advancement of machine learning and deep learning.

References

Team, K. (n.d.a). Keras documentation: Densenet.
URL: *https://keras.io/api/applications/densenet/*

Team, K. (n.d.b). Keras documentation: Kerastuner.
URL: *https://keras.io/keras$_t$uner/*

Team, K. (n.d.c). Keras documentation: Resnet and resnetv2.
URL: *https://keras.io/api/applications/resnet/*