National College of Ireland

Configuration Manual

MSc Research Project

Data Analytics

Sai Chethan Singu

Student ID: x18181937

School of Computing

National College of Ireland

Supervisor: Dr. Catherine Mulwa

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Sai Chethan Singu

Student ID: x18181937

Programme: MSc. Data Analytics **Year:** 2020

Module: Research Project

Supervisor: Dr. Catherine Mulwa

Submission Due Date: 17th Dec 2020

Project Title: CONFIGURATION MANUAL

Word Count: 1724 **Page Count** 17

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:

Date:

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):	

A Deep Neural Network Approach for The Detection of Driver Distraction: USA

Sai Chethan Singu
x18181937

1 Introduction

The objective of the configuration manual is to present the details of the software and hardware used a part of the environment in which the code is developed and deployed. A step-by-step explanation for the research project "A Deep Neural Network Approach for The Detection of Driver Distraction: USA". Each section of the documents presents the in-depth details along with the steps that need to be followed to implement and reproduce the solution provided.

2 Environment Configuration

The environment that is used to develop the solution is the Python programming language. The artifact is executed using two different environments which involve the local environments and the one that is hosted in Google Colaboratory.

2.1 Software Configuration

Coming to the software that is used as a part of the artifact development and deployment are as follows:

Google Collab
Google Collaboratory is the software that is used to execute few models in the artifact as the Collaboratory provides the resources through the cloud and they can be used without any intervention from the local environment that the machine has. Figure 1 presents the collab environment.

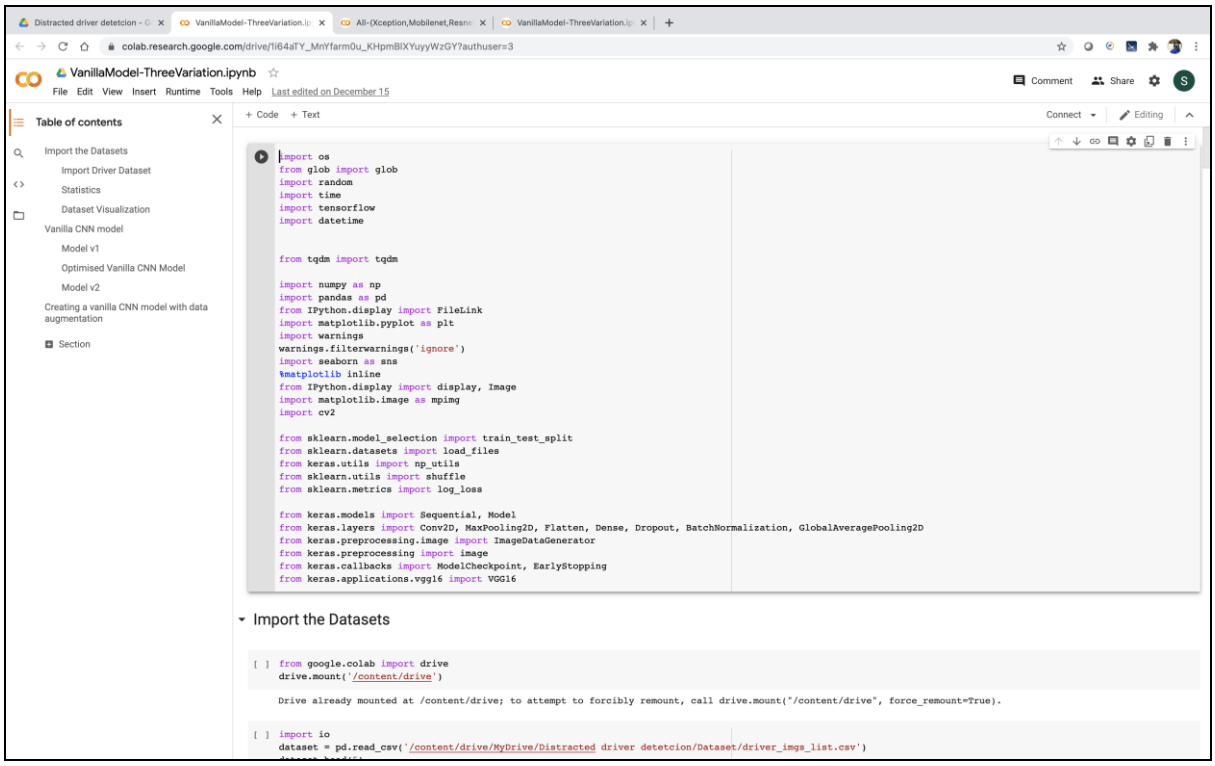

Figure 1: Colab Environment

Anaconda Jupyter Notebook

Anaconda's Jupyter notebook is also used to implement and execute a few models. This software makes use of the local configuration of the machine to run the models as part of the artifact. Figure 2 depicts the environment of the anaconda and Figure 3 presents the Jupyter notebook running on localhost.

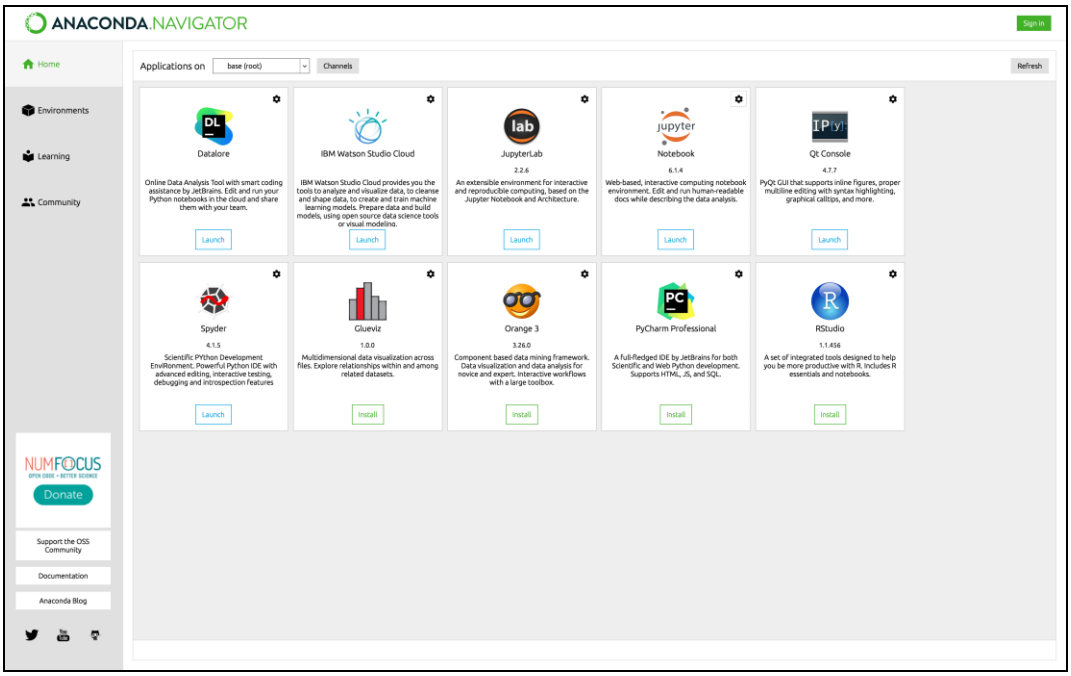

Figure 2: Anaconda Navigator

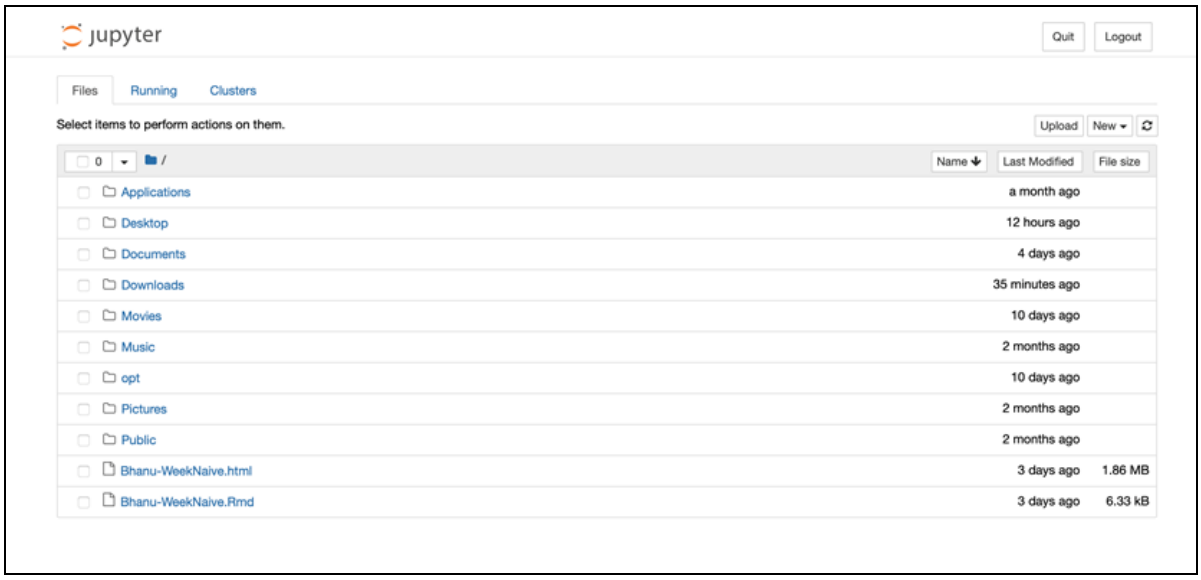

Figure 3: Jupyter Notebook

2.2 Hardware Configuration

The hardware that is used and the operating systems and the configuration of the system are as presented in figure 4.

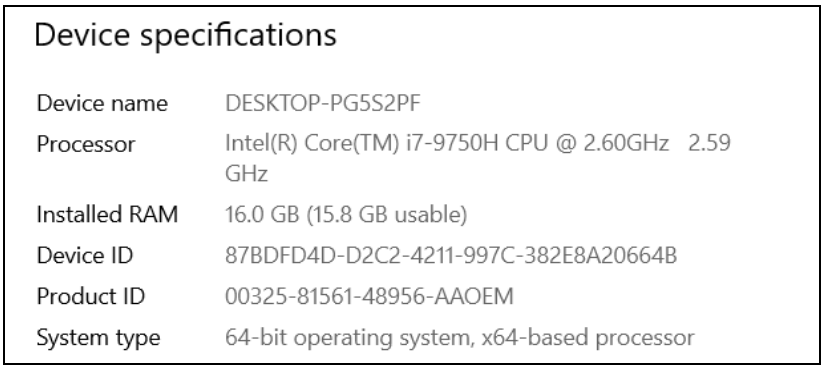

Figure 4: Hardware Configuration

3 Steps to Reproduce Artefact

3.1 Data Collection

The data is collected from State Farm Dataset. The dataset is in the form of images and each of them has different activities being performed by the driver. Figure 5 depicts the source of the dataset.

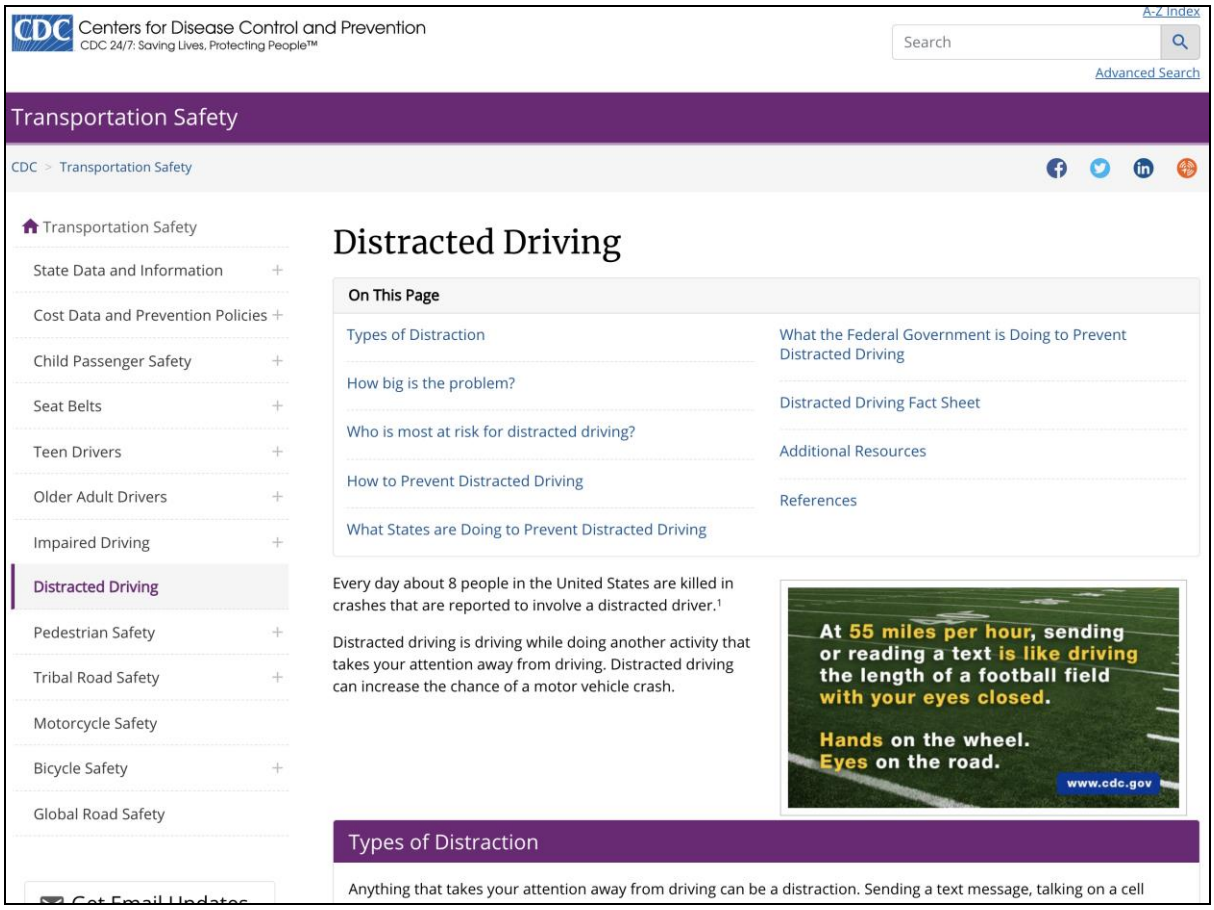

Figure 5: Dataset source

3.2 Data Loading

To import the dataset that is collected from the source, google drive is used to deploy the collected data and even the local machine is also made available with the data collected. The first step is to mount the drive in which the dataset is made available. Figure 6 provides the code block used to mount the drive.

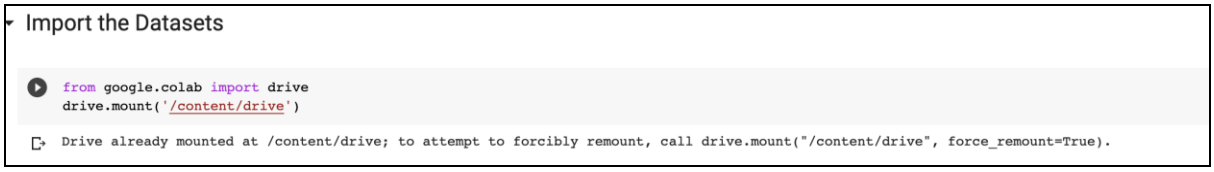

Figure 6: Drive Mounting

3.3 Data Exploration

The data that is collected is loaded and explored to understand the distribution. Figure 7-8 presents the code blocks involved in the exploration and reading of the data.

```
[ ] import io
    dataset = pd.read_csv('/content/drive/MyDrive/Distracted driver detetcion/Dataset/driver_imgs_list.csv')
    dataset.head(5)
```

	subject	classname	img
0	p002	c0	img_44733.jpg
1	p002	c0	img_72999.jpg
2	p002	c0	img_25094.jpg
3	p002	c0	img_69092.jpg
4	p002	c0	img_92629.jpg

Figure 7: Data Exploration

```
NUMBER_CLASSES = 10
# Color type: 1 - grey, 3 - rgb

def get_cv2_image(path, img_rows, img_cols, color_type=3):
    if color_type == 1:
        img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    elif color_type == 3:
        img = cv2.imread(path, cv2.IMREAD_COLOR)
    # Reduce size
    img = cv2.resize(img, (img_rows, img_cols))
    return img

# Training
def load_train(img_rows, img_cols, color_type=3):
    start_time = time.time()
    train_images = []
    train_labels = []
    # Loop over the training folder
    for classed in tqdm(range(NUMBER_CLASSES)):
        print('Loading directory c{}'.format(classed))
        files = glob(os.path.join('/content/drive/MyDrive/Distracted driver detetcion', 'Dataset', 'images','train', 'c' + str(classed), '*.jpg'))
        for file in files:
            img = get_cv2_image(file, img_rows, img_cols, color_type)
            train_images.append(img)
            train_labels.append(classed)
    print("Data Loaded in {} second".format(time.time() - start_time))
    return train_images, train_labels
```

```
x_train, x_test, y_train, y_test = read_and_normalize_train_data(img_rows, img_cols, color_type)
print('Train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
```

```
  0%|          | 0/10 [00:00<?, ?it/s]Loading directory c0
 10%|▉         | 1/10 [00:17<02:35, 17.31s/it]Loading directory c1
 20%|█▊        | 2/10 [00:33<02:14, 16.85s/it]Loading directory c2
 30%|██▋       | 3/10 [00:48<01:54, 16.39s/it]Loading directory c3
 40%|███▌      | 4/10 [01:04<01:38, 16.36s/it]Loading directory c4
 50%|████▌     | 5/10 [01:20<01:21, 16.26s/it]Loading directory c5
 60%|█████▍    | 6/10 [01:39<01:07, 16.99s/it]Loading directory c6
 70%|██████▎   | 7/10 [01:55<00:50, 16.78s/it]Loading directory c7
 80%|███████▏  | 8/10 [02:08<00:31, 15.66s/it]Loading directory c8
 90%|████████▏ | 9/10 [02:21<00:14, 14.71s/it]Loading directory c9
100%|█████████| 10/10 [19:13<00:00, 115.35s/it]Data Loaded in 1153.5110874176025 second
Train shape: (17939, 64, 64, 1)
17939 train samples
```

Figure 8: Loading the training dataset

```
# Validation
def load_test(size=200000, img_rows=64, img_cols=64, color_type=3):
    path = os.path.join('/content/drive/MyDrive/Distracted driver detetcion/Dataset/images/test/*')
    files = sorted(glob(path))
    X_test, X_test_id = [], []
    total = 0
    files_size = len(files)
    for file in tqdm(files):
        if total >= size
            break
        file_base = os.path.basename(file)
        img = get_cv2_image(file, img_rows, img_cols, color_type)
        X_test.append(img)
        X_test_id.append(file_base)
        total += 1
    return X_test, X_test_id
```

/content/drive/MyDrive/Distracted (cmd + click)

Figure 9: Loading Test Dataset

3.4 Data Pre-Processing

As part of the pre-processing, the loaded dataset is further processing using the code in figure 10.

```
def read_and_normalize_train_data(img_rows, img_cols, color_type):
    X, labels = load_train(img_rows, img_cols, color_type)
    y = np_utils.to_categorical(labels, 10)
    x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    x_train = np.array(x_train, dtype=np.uint8).reshape(-1,img_rows,img_cols,color_type)
    x_test = np.array(x_test, dtype=np.uint8).reshape(-1,img_rows,img_cols,color_type)

    return x_train, x_test, y_train, y_test
```

Figure 10: Normalizing Train Data

```
def read_and_normalize_sampled_test_data(size, img_rows, img_cols, color_type=3):
    test_data, test_ids = load_test(size, img_rows, img_cols, color_type)

    test_data = np.array(test_data, dtype=np.uint8)
    test_data = test_data.reshape(-1,img_rows,img_cols,color_type)

    return test_data, test_ids
```

Figure 11: Normalizing Test Data

3.5 Data Transformation

Figure 12 gives the details of the code used to understand the statistics and the distribution of classes in the dataset considered.

```
[ ] # Load the list of names
    names = [item[17:19] for item in sorted(glob("/content/drive/MyDrive/Distracted driver detetcion/Dataset/images/train/*"))]
    test_files_size = len(np.array(glob(os.path.join('/content/drive/MyDrive/Distracted driver detetcion/Dataset/images/test/*'))))
    x_train_size = len(x_train)
    categories_size = len(names)
    x_test_size = len(x_test)
    print('There are %s total images.\n' % (test_files_size + x_train_size + x_test_size))
    print('There are %d training images.' % x_train_size)
    print('There are %d total training categories.' % categories_size)
    print('There are %d validation images.' % x_test_size)
    print('There are %d test images.'% test_files_size)

    There are 102150 total images.

    There are 17939 training images.
    There are 10 total training categories.
    There are 4485 validation images.
    There are 79726 test images.
```

Figure 12: Statistics of Data

```
# Plot figure size
plt.figure(figsize = (10,10))
# Count the number of images per category
sns.countplot(x = 'classname', data = dataset)
# Change the Axis names
plt.ylabel('Count')
plt.title('Categories Distribution')
# Show plot
plt.show()
```

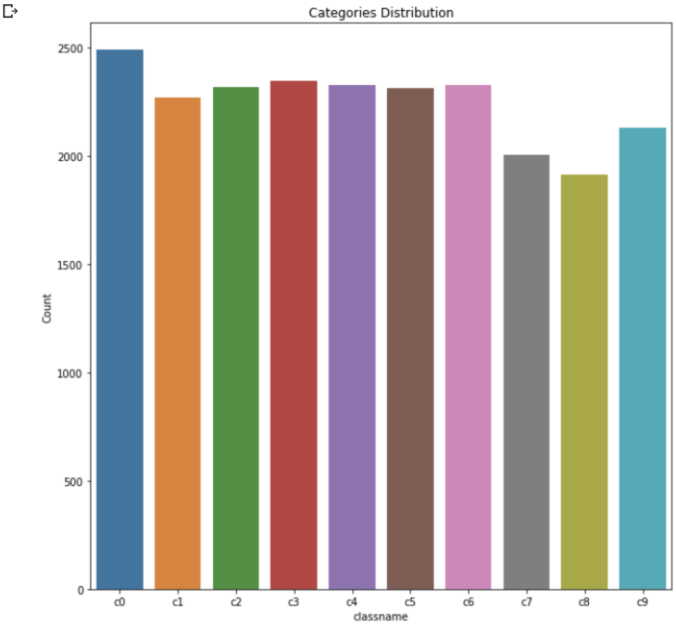

Figure 13: Data Visualization

As different activities are being performed each of them is given a category name so that it's easy to understand. Figure 14 presents the code involved in mapping the activities.

```
activity_map = {'c0': 'Safe driving',
                'c1': 'Texting - right',
                'c2': 'Talking on the phone - right',
                'c3': 'Texting - left',
                'c4': 'Talking on the phone - left',
                'c5': 'Operating the radio',
                'c6': 'Drinking',
                'c7': 'Reaching behind',
                'c8': 'Hair and makeup',
                'c9': 'Talking to passenger'}
```

Figure 14: Activity Mapping

4 Implementation

4.1 Vanilla Model

Model Construction

Figure 15 presents the code block that is used to construct the vanilla model. In this, the Keras library is used to implement the layers in the model. Input is given as the argument to construct the model.

```python
def create_model_v1():
    # Vanilla CNN model
    model = Sequential()

    model.add(Conv2D(filters = 64, kernel_size = 3, padding='same', activation = 'relu', input_shape=(img_rows, img_cols, color_type)))
    model.add(MaxPooling2D(pool_size = 2))

    model.add(Conv2D(filters = 128, padding='same', kernel_size = 3, activation = 'relu'))
    model.add(MaxPooling2D(pool_size = 2))

    model.add(Conv2D(filters = 256, padding='same', kernel_size = 3, activation = 'relu'))
    model.add(MaxPooling2D(pool_size = 2))

    model.add(Conv2D(filters = 512, padding='same', kernel_size = 3, activation = 'relu'))
    model.add(MaxPooling2D(pool_size = 2))

    model.add(Dropout(0.5))

    model.add(Flatten())

    model.add(Dense(500, activation = 'relu'))
    model.add(Dropout(0.5))
    model.add(Dense(10, activation = 'softmax'))

    return model
```

Figure 15: Model Construction

Model Compilation

The model that is constructed is compiled with the required parameters which include the loss, metrics and the optimizer. Figure 16 provides the details of the model compilation.

```python
model_v1 = create_model_v1()

# More details about the layers
model_v1.summary()

# Compiling the model
model_v1.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 64, 64, 64)        640
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 128)       73856
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 256)       295168
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 256)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 512)         1180160
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 512)         0
_________________________________________________________________
dropout (Dropout)            (None, 4, 4, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense (Dense)                (None, 500)               4096500
_________________________________________________________________
dropout_1 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                5010
=================================================================
Total params: 5,651,334
Trainable params: 5,651,334
Non-trainable params: 0
_________________________________________________________________
```

Figure 16: Model Compilation

Model Training and Evaluation

The model that is compiled will be used further to get trained using the train data formulated and the validation data is given to evaluate the model that is constructed and compiled.

```
# Training the Vanilla Model version 1
history_v1 = model_v1.fit(x_train, y_train,
          validation_data=(x_test, y_test),
          callbacks=callbacks,
          epochs=nb_epoch, batch_size=batch_size, verbose=1)
```

Figure 17: Model Training

```
Epoch 1/10
449/449 [==============================] - ETA: 0s - loss: 2.5199 - accuracy: 0.4953
Epoch 00001: val_loss improved from inf to 0.29325, saving model to saved_models/weights_best_vanilla.hdf5
449/449 [==============================] - 137s 305ms/step - loss: 2.5199 - accuracy: 0.4953 - val_loss: 0.2932 - val_accuracy: 0.9142
Epoch 2/10
449/449 [==============================] - ETA: 0s - loss: 0.3564 - accuracy: 0.8948
Epoch 00002: val_loss improved from 0.29325 to 0.14773, saving model to saved_models/weights_best_vanilla.hdf5
449/449 [==============================] - 138s 307ms/step - loss: 0.3564 - accuracy: 0.8948 - val_loss: 0.1477 - val_accuracy: 0.9648
Epoch 3/10
449/449 [==============================] - ETA: 0s - loss: 0.2363 - accuracy: 0.9376
Epoch 00003: val_loss did not improve from 0.14773
449/449 [==============================] - 137s 304ms/step - loss: 0.2363 - accuracy: 0.9376 - val_loss: 0.1634 - val_accuracy: 0.9639
Epoch 4/10
449/449 [==============================] - ETA: 0s - loss: 0.2000 - accuracy: 0.9538
Epoch 00004: val_loss improved from 0.14773 to 0.08214, saving model to saved_models/weights_best_vanilla.hdf5
449/449 [==============================] - 143s 318ms/step - loss: 0.2000 - accuracy: 0.9538 - val_loss: 0.0821 - val_accuracy: 0.9844
Epoch 5/10
449/449 [==============================] - ETA: 0s - loss: 0.2022 - accuracy: 0.9565
Epoch 00005: val_loss improved from 0.08214 to 0.05210, saving model to saved_models/weights_best_vanilla.hdf5
449/449 [==============================] - 142s 316ms/step - loss: 0.2022 - accuracy: 0.9565 - val_loss: 0.0521 - val_accuracy: 0.9889
Epoch 6/10
449/449 [==============================] - ETA: 0s - loss: 0.1882 - accuracy: 0.9605
Epoch 00006: val_loss did not improve from 0.05210
449/449 [==============================] - 141s 314ms/step - loss: 0.1882 - accuracy: 0.9605 - val_loss: 0.1440 - val_accuracy: 0.9799
Epoch 7/10
449/449 [==============================] - ETA: 0s - loss: 0.1985 - accuracy: 0.9629
Epoch 00007: val_loss did not improve from 0.05210
449/449 [==============================] - 134s 298ms/step - loss: 0.1985 - accuracy: 0.9629 - val_loss: 0.0953 - val_accuracy: 0.9889
Epoch 00007: early stopping
```

Figure 18: Model Execution

4.2 Optimized Vanilla CNN

Model Construction

Below Figure 19 presents the code block that is used to construct the vanilla model. Additional normalization and dropout layers are added to the base model to optimize it. In this, the Keras library is used to implement the layers in the model. Input is given as the argument to construct the model.

```
def create_model_v2():
    # Optimised Vanilla CNN model
    model = Sequential()
    model.add(Conv2D(32,(3,3),activation='relu',input_shape=(img_rows, img_cols, color_type)))
    model.add(BatchNormalization())
    model.add(Conv2D(32,(3,3),activation='relu',padding='same'))
    model.add(BatchNormalization(axis = 3))
    model.add(MaxPooling2D(pool_size=(2,2),padding='same'))
    model.add(Dropout(0.3))
    model.add(Conv2D(64,(3,3),activation='relu',padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(64,(3,3),activation='relu',padding='same'))
    model.add(BatchNormalization(axis = 3))
    model.add(MaxPooling2D(pool_size=(2,2),padding='same'))
    model.add(Dropout(0.3))
    model.add(Conv2D(128,(3,3),activation='relu',padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(128,(3,3),activation='relu',padding='same'))
    model.add(BatchNormalization(axis = 3))
    model.add(MaxPooling2D(pool_size=(2,2),padding='same'))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(512,activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
    model.add(Dense(128,activation='relu'))
    model.add(Dropout(0.25))
    model.add(Dense(10,activation='softmax'))

    return model
```

Figure 19: Model Construction

Model Compilation

The model that is constructed is compiled with the required parameters which include the loss, metrics and the optimizer. Figure 20 provides the details of the model compilation.

```
model_v2 = create_model_v2()

# More details about the layers
model_v2.summary()

# Compiling the model
model_v2.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 62, 62, 32)        320
_________________________________________________________________
batch_normalization (BatchNo (None, 62, 62, 32)        128
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 62, 62, 32)        9248
_________________________________________________________________
batch_normalization_1 (Batch (None, 62, 62, 32)        128
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 31, 31, 32)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 31, 31, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 31, 31, 64)        18496
_________________________________________________________________
batch_normalization_2 (Batch (None, 31, 31, 64)        256
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 31, 31, 64)        36928
_________________________________________________________________
batch_normalization_3 (Batch (None, 31, 31, 64)        256
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 16, 16, 64)        0
_________________________________________________________________
dropout_3 (Dropout)          (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 16, 16, 128)       73856
_________________________________________________________________
batch_normalization_4 (Batch (None, 16, 16, 128)       512
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 16, 16, 128)       147584
_________________________________________________________________
batch_normalization_5 (Batch (None, 16, 16, 128)       512
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 8, 8, 128)         0
_________________________________________________________________
dropout_4 (Dropout)          (None, 8, 8, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               4194816
_________________________________________________________________
batch_normalization_6 (Batch (None, 512)               2048
```

Figure 20: Model Compilation

Model Training and evaluation

The model that is compiled will be used further to get trained using the train data formulated and the validation data is given to evaluate the model that is constructed and compiled.

```
# Training the Vanilla Model
history_v2 = model_v2.fit(x_train, y_train,
          validation_data=(x_test, y_test),
          callbacks=callbacks,
          epochs=nb_epoch, batch_size=batch_size, verbose=1)
```

Figure 21: Model Training

```
Epoch 1/10
449/449 [==============================] - ETA: 0s - loss: 1.2292 - accuracy: 0.5928
Epoch 00001: val_loss did not improve from 0.05210
449/449 [==============================] - 185s 413ms/step - loss: 1.2292 - accuracy: 0.5928 - val_loss: 0.4827 - val_accuracy: 0.8544
Epoch 2/10
449/449 [==============================] - ETA: 0s - loss: 0.3530 - accuracy: 0.8906
Epoch 00002: val_loss did not improve from 0.05210
449/449 [==============================] - 183s 408ms/step - loss: 0.3530 - accuracy: 0.8906 - val_loss: 0.1491 - val_accuracy: 0.9605
Epoch 3/10
449/449 [==============================] - ETA: 0s - loss: 0.2120 - accuracy: 0.9351
Epoch 00003: val_loss did not improve from 0.05210
449/449 [==============================] - 184s 409ms/step - loss: 0.2120 - accuracy: 0.9351 - val_loss: 0.2427 - val_accuracy: 0.9318
Epoch 4/10
449/449 [==============================] - ETA: 0s - loss: 0.1478 - accuracy: 0.9553
Epoch 00004: val_loss did not improve from 0.05210
449/449 [==============================] - 184s 409ms/step - loss: 0.1478 - accuracy: 0.9553 - val_loss: 0.0991 - val_accuracy: 0.9766
Epoch 5/10
449/449 [==============================] - ETA: 0s - loss: 0.1220 - accuracy: 0.9645
Epoch 00005: val_loss did not improve from 0.05210
449/449 [==============================] - 184s 409ms/step - loss: 0.1220 - accuracy: 0.9645 - val_loss: 0.2256 - val_accuracy: 0.9425
Epoch 6/10
449/449 [==============================] - ETA: 0s - loss: 0.1120 - accuracy: 0.9663
Epoch 00006: val_loss did not improve from 0.05210
449/449 [==============================] - 184s 409ms/step - loss: 0.1120 - accuracy: 0.9663 - val_loss: 0.0868 - val_accuracy: 0.9797
Epoch 7/10
449/449 [==============================] - ETA: 0s - loss: 0.1002 - accuracy: 0.9718
Epoch 00007: val_loss did not improve from 0.05210
449/449 [==============================] - 184s 410ms/step - loss: 0.1002 - accuracy: 0.9718 - val_loss: 0.0916 - val_accuracy: 0.9866
Epoch 8/10
449/449 [==============================] - ETA: 0s - loss: 0.0943 - accuracy: 0.9743
Epoch 00008: val_loss did not improve from 0.05210
449/449 [==============================] - 186s 415ms/step - loss: 0.0943 - accuracy: 0.9743 - val_loss: 0.0582 - val_accuracy: 0.9882
Epoch 9/10
449/449 [==============================] - ETA: 0s - loss: 0.0798 - accuracy: 0.9768
```

Figure 22: Model Execution

4.3 Vanilla CNN with Data Augmentation

Creating Augmentation of Data

The data that is available for the training and testing is augmented using the image generator. And these generated images are further used as a part of the optimized vanilla CNN model to complete the training and evaluation of the model. Figures 23-25 present the code block for the generation of data, training and evaluation.

Create a vanilla CNN model with data augmentation

```
[ ] # Prepare data augmentation configuration
    train_datagen = ImageDataGenerator(rescale = 1.0/255,
                                       shear_range = 0.2,
                                       zoom_range = 0.2,
                                       horizontal_flip = True,
                                       validation_split = 0.2)

    test_datagen = ImageDataGenerator(rescale=1.0/ 255, validation_split = 0.2)
```

```
[ ] nb_train_samples = x_train.shape[0]
    nb_validation_samples = x_test.shape[0]
    print(nb_train_samples)
    print(nb_validation_samples)
    training_generator = train_datagen.flow(x_train, y_train, batch_size=batch_size)
    validation_generator = test_datagen.flow(x_test, y_test, batch_size=batch_size)

    17939
    4485
```

Figure 23: Data Augmentation

```
Train the model with Data Augmentation

Using fit_generator, I'll train the model.

[ ] checkpoint = ModelCheckpoint('saved_models/weights_best_vanilla.hdf5', monitor='val_acc', verbose=1, save_best_only=True, mode='max')
    history_v3 = model_v2.fit_generator(training_generator,
                                steps_per_epoch = nb_train_samples // batch_size,
                                epochs = 5,
                                callbacks=[es, checkpoint],
                                verbose = 1,
                                validation_data = validation_generator,
                                validation_steps = nb_validation_samples // batch_size)
```

Figure 24: Model Training

```
Please use Model.fit, which supports generators.
Epoch 1/5
448/448 [==============================] - ETA: 0s - loss: 1.3772 - accuracy: 0.6108WARNING:tensorflow:Can save best model only with val_acc available, skipping.
448/448 [==============================] - 175s 390ms/step - loss: 1.3772 - accuracy: 0.6108 - val_loss: 17.8554 - val_accuracy: 0.0879
Epoch 2/5
448/448 [==============================] - ETA: 0s - loss: 0.7181 - accuracy: 0.7647WARNING:tensorflow:Can save best model only with val_acc available, skipping.
448/448 [==============================] - 173s 387ms/step - loss: 0.7181 - accuracy: 0.7647 - val_loss: 0.4548 - val_accuracy: 0.8574
Epoch 3/5
448/448 [==============================] - ETA: 0s - loss: 0.5386 - accuracy: 0.8283WARNING:tensorflow:Can save best model only with val_acc available, skipping.
448/448 [==============================] - 175s 391ms/step - loss: 0.5386 - accuracy: 0.8283 - val_loss: 0.1409 - val_accuracy: 0.9759
Epoch 4/5
448/448 [==============================] - ETA: 0s - loss: 0.4516 - accuracy: 0.8612WARNING:tensorflow:Can save best model only with val_acc available, skipping.
```

Figure 25: Model Evaluation

4.4 Xception Model

Model Construction

Xception Model is constructed using the Keras application where the pre-trained model is available. Figure 26 presents the code used to load the base Xception model. Later the model is enhanced by adding extra layers and is presented in Figure 27.

```
## Defining the input

from keras.layers import Input
xception_input = Input(shape = (224, 224, 3), name = 'Image_input')

from keras.applications.xception import preprocess_input, decode_predictions
from keras.applications.xception import Xception

model_xception_conv = Xception(weights= 'imagenet', include_top=False, input_shape= (224,224,3))
model_xception_conv.summary()
```

Figure 26: Xception Base Model

Model Compilation

Once the model is constructed as in Figure 26, the model is compiled with the required parameters as shown in figure 28.

```
#Use the generated model
from keras.models import Model


output_xception_conv = model_xception_conv(xception_input)

#Add the fully-connected layers

x=GlobalAveragePooling2D()(output_xception_conv)
x=Dense(1024,activation='relu')(x) #we add dense layers so that the model can learn more complex functions and classify for better results.
x = Dropout(0.1)(x) # **reduce dropout
x=Dense(1024,activation='relu')(x) #dense layer 2
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(512,activation='relu')(x) #dense layer 3
x = Dense(10, activation='softmax', name='predictions')(x)


xception_pretrained = Model(inputs = xception_input, outputs = x)
xception_pretrained.summary()
```

Figure 27: Xception Model with extra Layers

```
# Compile CNN model
adam = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-8, decay=0.0)
sgd = optimizers.SGD(lr = 0.005)
xception_pretrained.compile(loss='categorical_crossentropy',optimizer = sgd,metrics=['accuracy'])
```

Figure 28: Model Compilation

Model Training and evaluation

The model is trained with the image generator and the data that is augmented using real-world augmentation. Training and evaluation of the model using the fit generator function are compiled using the code in figure 29.

```
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ModelCheckpoint,EarlyStopping

checkpointer = ModelCheckpoint('xception_weights_aug_extralayer_alltrained_sgd2_V2.hdf5', verbose=1, save_best_only=True)
earlystopper = EarlyStopping(monitor='val_loss', patience=10, verbose=1)


datagen = ImageDataGenerator(
    height_shift_range=0.5,
    width_shift_range = 0.5,
    zoom_range = 0.5,
    rotation_range=30
        )
#datagen.fit(X_train)
data_generator = datagen.flow(X_train, y_train, batch_size = 64)

# Fits the model on batches with real-time data augmentation:
xception_model = xception_pretrained.fit_generator(data_generator,steps_per_epoch = len(X_train) / 64, callbacks=[checkpointer, earlystopper],
                                                   epochs = 10, verbose = 1, validation_data = (X_test, y_test))
```

Figure 29: Model Training and Evaluation

Model Results

The results that are obtained using the Xception model constructed are presented in Figure.30

```
Epoch 1/10
293/292 [==============================] - ETA: -4s - loss: 2.3438 - accuracy: 0.1453
Epoch 00001: val_loss improved from inf to 2.26873, saving model to xception_weights_aug_extralayer_alltrained_sgd2_V2.hdf5
293/292 [==============================] - 4337s 15s/step - loss: 2.3438 - accuracy: 0.1453 - val_loss: 2.2687 - val_accuracy: 0.2793
Epoch 2/10
293/292 [==============================] - ETA: -4s - loss: 2.0359 - accuracy: 0.2619
Epoch 00002: val_loss improved from 2.26873 to 1.72574, saving model to xception_weights_aug_extralayer_alltrained_sgd2_V2.hdf5
293/292 [==============================] - 4746s 16s/step - loss: 2.0359 - accuracy: 0.2619 - val_loss: 1.7257 - val_accuracy: 0.4456
Epoch 3/10
293/292 [==============================] - ETA: -13s - loss: 1.6085 - accuracy: 0.4242
Epoch 00003: val_loss improved from 1.72574 to 1.14961, saving model to xception_weights_aug_extralayer_alltrained_sgd2_V2.hdf5
293/292 [==============================] - 12576s 43s/step - loss: 1.6085 - accuracy: 0.4242 - val_loss: 1.1496 - val_accuracy: 0.6330
Epoch 4/10
293/292 [==============================] - ETA: -3s - loss: 1.2508 - accuracy: 0.5637
Epoch 00004: val_loss improved from 1.14961 to 0.82085, saving model to xception_weights_aug_extralayer_alltrained_sgd2_V2.hdf5
293/292 [==============================] - 3782s 13s/step - loss: 1.2508 - accuracy: 0.5637 - val_loss: 0.8209 - val_accuracy: 0.7275
Epoch 5/10
293/292 [==============================] - ETA: -4s - loss: 1.0051 - accuracy: 0.6507
Epoch 00005: val_loss improved from 0.82085 to 0.69705, saving model to xception_weights_aug_extralayer_alltrained_sgd2_V2.hdf5
293/292 [==============================] - 3943s 13s/step - loss: 1.0051 - accuracy: 0.6507 - val_loss: 0.6971 - val_accuracy: 0.7771
Epoch 6/10
293/292 [==============================] - ETA: -3s - loss: 0.8850 - accuracy: 0.6925
Epoch 00006: val_loss improved from 0.69705 to 0.62197, saving model to xception_weights_aug_extralayer_alltrained_sgd2_V2.hdf5
293/292 [==============================] - 3798s 13s/step - loss: 0.8850 - accuracy: 0.6925 - val_loss: 0.6220 - val_accuracy: 0.7868
Epoch 7/10
293/292 [==============================] - ETA: -4s - loss: 0.8037 - accuracy: 0.7207
Epoch 00007: val_loss improved from 0.62197 to 0.60074, saving model to xception_weights_aug_extralayer_alltrained_sgd2_V2.hdf5
293/292 [==============================] - 4693s 16s/step - loss: 0.8037 - accuracy: 0.7207 - val_loss: 0.6007 - val_accuracy: 0.8082
Epoch 8/10
293/292 [==============================] - ETA: -4s - loss: 0.7176 - accuracy: 0.7509
Epoch 00008: val_loss improved from 0.60074 to 0.56567, saving model to xception_weights_aug_extralayer_alltrained_sgd2_V2.hdf5
293/292 [==============================] - 4077s 14s/step - loss: 0.7176 - accuracy: 0.7509 - val_loss: 0.5657 - val_accuracy: 0.8069
```

Figure 30: Model Results

4.5 MobileNet Model

Model Construction

Mobile Net Model is constructed using the Keras application where the pre-trained model is available. Figure 31 presents the code used to load the base MobileNet model. Later the model is enhanced by adding extra layers and is presented in Figure 32.

```
[ ]  x=base_model.output
     x=GlobalAveragePooling2D()(x)

     preds=Dense(10,activation='softmax')(x) #final layer with softmax activation

     model = Model(inputs=base_model.input, outputs=preds)

     model.summary()
```

Figure 31: MobileNet Base Model

Model Compilation

Once the model is constructed as in Figure 31, the model is compiled with the required parameters as shown in figure 33.

```
x=base_model.output
x=GlobalAveragePooling2D()(x)

preds=Dense(10,activation='softmax')(x) #final layer with softmax activation

model = Model(inputs=base_model.input, outputs=preds)

model.summary()
```

Figure 32: Mobile Net Model with extra Layers

```
conv_dw_4_relu (ReLU)        (None, None, None, 128)   0
```

```
from keras import optimizers
sgd = optimizers.SGD(lr = 0.005)

model.compile(optimizer=sgd,loss='categorical_crossentropy',metrics=['accuracy'])
```

Figure 33: Model Compilation

Model Training and evaluation

The model is trained with the image generator and the data that is augmented using real-world augmentation. Training and evaluation of the model using the fit generator function are compiled using the code in figure 34.

```
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ModelCheckpoint,EarlyStopping

checkpointer = ModelCheckpoint('mobilenet_sgd_nolayers.hdf5', verbose=1, save_best_only=True)
earlystopper = EarlyStopping(monitor='val_loss', patience=10, verbose=1)

datagen = ImageDataGenerator(
    height_shift_range=0.5,
    width_shift_range = 0.5,
    zoom_range = 0.5,
    rotation_range=30
        )
#datagen.fit(X_train)
data_generator = datagen.flow(X_train, y_train, batch_size = 64)

# Fits the model on batches with real-time data augmentation:
mobilenet_model = model.fit_generator(data_generator,steps_per_epoch = len(X_train) / 64, callbacks=[checkpointer, earlystopper],
                                      epochs = 5, verbose = 1, validation_data = (X_test, y_test))
```

Figure 34: Model Training and Evaluation

Model Results

Below Figure 35 presents the results that are obtained using the model constructed.

```
Epoch 1/5
293/292 [==============================] - ETA: -1s - loss: 1.8243 - accuracy: 0.3775
Epoch 00001: val_loss improved from inf to 1.08047, saving model to mobilenet_sgd_nolayers.hdf5
293/292 [==============================] - 1221s 4s/step - loss: 1.8243 - accuracy: 0.3775 - val_loss: 1.0805 - val_accuracy: 0.6487
Epoch 2/5
293/292 [==============================] - ETA: -1s - loss: 1.0267 - accuracy: 0.6693
Epoch 00002: val_loss improved from 1.08047 to 0.60916, saving model to mobilenet_sgd_nolayers.hdf5
293/292 [==============================] - 1213s 4s/step - loss: 1.0267 - accuracy: 0.6693 - val_loss: 0.6092 - val_accuracy: 0.7988
Epoch 3/5
293/292 [==============================] - ETA: -1s - loss: 0.7726 - accuracy: 0.7504
Epoch 00003: val_loss improved from 0.60916 to 0.51395, saving model to mobilenet_sgd_nolayers.hdf5
293/292 [==============================] - 1219s 4s/step - loss: 0.7726 - accuracy: 0.7504 - val_loss: 0.5140 - val_accuracy: 0.8229
Epoch 4/5
293/292 [==============================] - ETA: -1s - loss: 0.6713 - accuracy: 0.7818
Epoch 00004: val_loss improved from 0.51395 to 0.48115, saving model to mobilenet_sgd_nolayers.hdf5
293/292 [==============================] - 1266s 4s/step - loss: 0.6713 - accuracy: 0.7818 - val_loss: 0.4811 - val_accuracy: 0.8402
Epoch 5/5
```

Figure 35: Model Results

4.6 ResNet50 Model

Model Construction

ResNet50 model is constructed using the Keras application where the pre-trained model is available. Figure 36 presents the code used to load the base ResNet50 model.

```
## Defining the input

from keras.layers import Input
resnet50_input = Input(shape = (224, 224, 3), name = 'Image_input')

## The RESNET model

from keras.applications.resnet50 import preprocess_input, decode_predictions
from keras.applications.resnet50 import ResNet50

model_resnet50_conv = ResNet50(weights= 'imagenet', include_top=False, input_shape= (224,224,3))
model_resnet50_conv.summary()
```

Figure 36: ResNet50 Base Model

Model Compilation

Once the model is constructed as in Figure 36, the model is compiled with the required parameters as shown in figure 37.

```
from keras.callbacks import LearningRateScheduler
output_resnet50_conv = model_resnet50_conv(resnet50_input)

#Add the fully-connected layers

x = Flatten(name='flatten')(output_resnet50_conv)
x = Dense(10, activation='softmax', name='predictions')(x)


resnet50_pretrained = Model(inputs = resnet50_input, outputs = x)
resnet50_pretrained.summary()

# Compile CNN model
adam = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-8, decay=0.0)



def step_decay(epoch):
    initial_lrate = 0.001
    drop = 0.5
    epochs_drop = 10.0
    lrate = initial_lrate * math.pow(drop,
        math.floor((1+epoch)/epochs_drop))
    return lrate
lrate = LearningRateScheduler(step_decay)

sgd = optimizers.SGD(lr = 0.001)


resnet50_pretrained.compile(loss='categorical_crossentropy',optimizer = sgd,metrics=['accuracy'])
```

Figure 37: Model Compilation

Model Training and evaluation

The model is trained with the image generator and the data that is augmented using real-world augmentation. Training and evaluation of the model using the fit generator function are compiled using the code in figure 38.

```
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ModelCheckpoint,EarlyStopping
import math
checkpointer = ModelCheckpoint('resnet_weights_aug_alltrained_sgd2_setval.hdf5', verbose=1, save_best_only=True)
earlystopper = EarlyStopping(monitor='accuracy', patience=7, verbose=1)


datagen = ImageDataGenerator(
    height_shift_range=0.5,
    width_shift_range = 0.5,
    zoom_range = 0.5,
    rotation_range=30
      )
#datagen.fit(X_train)
data_generator = datagen.flow(X_train, y_train, batch_size = 64)

# Fits the model on batches with real-time data augmentation:
resnet50_model = resnet50_pretrained.fit_generator(data_generator,steps_per_epoch = len(X_train) / 64, callbacks=[checkpointer, earlystopper,lrate],
                                                   epochs = 10, verbose = 1, validation_data = (X_test, y_test))
```

Figure 38: Model Training and Evaluation

Model Results

Figure 39 presents the results that are obtained using the model constructed.

```
Epoch 1/5
293/292 [==============================] - ETA: -1s - loss: 1.8243 - accuracy: 0.3775
Epoch 00001: val_loss improved from inf to 1.08047, saving model to mobilenet_sgd_nolayers.hdf5
293/292 [==============================] - 1221s 4s/step - loss: 1.8243 - accuracy: 0.3775 - val_loss: 1.0805 - val_accuracy: 0.6487
Epoch 2/5
293/292 [==============================] - ETA: -1s - loss: 1.0267 - accuracy: 0.6693
Epoch 00002: val_loss improved from 1.08047 to 0.60916, saving model to mobilenet_sgd_nolayers.hdf5
293/292 [==============================] - 1213s 4s/step - loss: 1.0267 - accuracy: 0.6693 - val_loss: 0.6092 - val_accuracy: 0.7988
Epoch 3/5
293/292 [==============================] - ETA: -1s - loss: 0.7726 - accuracy: 0.7504
Epoch 00003: val_loss improved from 0.60916 to 0.51395, saving model to mobilenet_sgd_nolayers.hdf5
293/292 [==============================] - 1219s 4s/step - loss: 0.7726 - accuracy: 0.7504 - val_loss: 0.5140 - val_accuracy: 0.8229
Epoch 4/5
293/292 [==============================] - ETA: -1s - loss: 0.6713 - accuracy: 0.7818
Epoch 00004: val_loss improved from 0.51395 to 0.48115, saving model to mobilenet_sgd_nolayers.hdf5
293/292 [==============================] - 1266s 4s/step - loss: 0.6713 - accuracy: 0.7818 - val_loss: 0.4811 - val_accuracy: 0.8402
Epoch 5/5
```

Figure 39: Model Results

4.7 VGG16 Model

Model Construction

The VGG16 model is constructed using the Keras application where the pre-trained model is available. Figure 40 presents the code used to load the base VGG16 model.

```
## Defining the input

from keras.layers import Input
vgg16_input = Input(shape = (224, 224, 3), name = 'Image_input')


## The VGG model

from keras.applications.vgg16 import VGG16, preprocess_input

model_vgg16_conv = VGG16(weights='imagenet', include_top=False, input_tensor = vgg16_input)
model_vgg16_conv.summary()
```

Figure 40: VGG16 Base Model

Model Compilation

Once the model is constructed as in Figure 40, the model is compiled with the required parameters as shown in figure 41.

```
#Use the generated model
from keras.models import Model


output_vgg16_conv = model_vgg16_conv(vgg16_input)

x = Flatten(name='flatten')(output_vgg16_conv)

x = Dense(10, activation='softmax', name='predictions')(x)

vgg16_pretrained = Model(inputs = vgg16_input, outputs = x)
vgg16_pretrained.summary()

# Compile CNN model
sgd = optimizers.SGD(lr = 0.001)
vgg16_pretrained.compile(loss='categorical_crossentropy',optimizer = sgd,metrics=['accuracy'])
```

Figure 41: Model Compilation

Model Training and evaluation

The model is trained with the image generator and the data that is augmented using real-world augmentation. Training and evaluation of the model using the fit generator function are compiled using the code in figure 42.

```
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ModelCheckpoint,EarlyStopping

checkpointer = ModelCheckpoint('vgg_weights_aug_setval_sgd.hdf5', verbose=1, save_best_only=True)
earlystopper = EarlyStopping(monitor='val_loss', patience=10, verbose=1)


datagen = ImageDataGenerator(
    height_shift_range=0.5,
    width_shift_range = 0.5,
    zoom_range = 0.5,
    rotation_range=30
        )
#datagen.fit(X_train)
data_generator = datagen.flow(X_train, y_train, batch_size = 64)

# Fits the model on batches with real-time data augmentation:
vgg16_model = vgg16_pretrained.fit_generator(data_generator,steps_per_epoch = len(X_train) / 64, callbacks=[checkpointer, earlystopper],
                                             epochs = 10, verbose = 1, validation_data = (X_test, y_test))
```

Figure 42: Model Training and Evaluation

Model Results

Figure 43 presents the results that are obtained using the model constructed.

```
                                             epochs = 10, verbose = 1, validation_data = (X_test, y_test))

Epoch 1/10
293/292 [==============================] - ETA: -5s - loss: 2.3742 - accuracy: 0.1088
Epoch 00001: val_loss improved from inf to 2.27588, saving model to vgg_weights_aug_setval_sgd.hdf5
293/292 [==============================] - 5264s 18s/step - loss: 2.3742 - accuracy: 0.1088 - val_loss: 2.2759 - val_accuracy: 0.1587
Epoch 2/10
293/292 [==============================] - ETA: -4s - loss: 2.2545 - accuracy: 0.1507
Epoch 00002: val_loss improved from 2.27588 to 2.17650, saving model to vgg_weights_aug_setval_sgd.hdf5
293/292 [==============================] - 4883s 17s/step - loss: 2.2545 - accuracy: 0.1507 - val_loss: 2.1765 - val_accuracy: 0.1918
Epoch 3/10
293/292 [==============================] - ETA: -5s - loss: 2.1727 - accuracy: 0.1847
Epoch 00003: val_loss did not improve from 2.17650
293/292 [==============================] - 4989s 17s/step - loss: 2.1727 - accuracy: 0.1847 - val_loss: 2.1802 - val_accuracy: 0.1999
Epoch 4/10
293/292 [==============================] - ETA: -5s - loss: 2.0648 - accuracy: 0.2228
Epoch 00004: val_loss improved from 2.17650 to 1.91398, saving model to vgg_weights_aug_setval_sgd.hdf5
293/292 [==============================] - 5107s 17s/step - loss: 2.0648 - accuracy: 0.2228 - val_loss: 1.9140 - val_accuracy: 0.2692
Epoch 5/10
293/292 [==============================] - ETA: -5s - loss: 1.9001 - accuracy: 0.2774
Epoch 00005: val_loss improved from 1.91398 to 1.86395, saving model to vgg_weights_aug_setval_sgd.hdf5
```

Figure 43: Model Results