

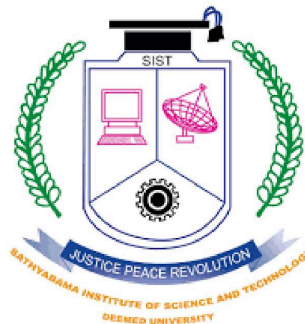
HUMAN ACTIVITY RECOGNITION WITH POSE ESTIMATION

Submitted in partial fulfillment of the requirements for the award of
Bachelor of Engineering Degree in Computer Science Engineering

By

TANMAYE NAINALA (REG. NO - 40111319)

SUCHI SMITHA KOLLURU (REG.NO- 40111273)



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SCHOOL OF COMPUTING**

SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Category-1 University By UGC

Accredited with Grade “A++” by NAAC | 12 B Status

by UGC | Approved by AICTE

JEPPIAR NAGAR, RAJIV GANDHISALAI,

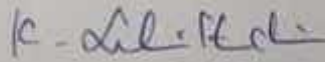
CHENNAI – 600119

APRIL 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

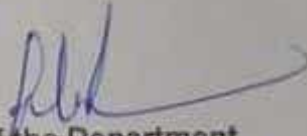
BONAFIDE CERTIFICATE

This is to certify that this Project Report is the bonafide work of Tanmaye Nainala (40111319) and Suchi Smitha Kolluru (40111273) who carried out the project entitled "Human Activity Recognition And Pose Estimation" under my supervision from November 2023 to April 2024.



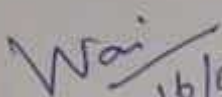
Internal Guide

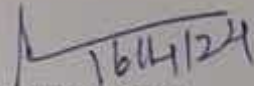
Dr.K. Lalitha Devi M.E.,Ph. D


Head of the Department

Dr. HEAD OF THE DEPARTMENT
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Jeppiaar Nagar, Rajiv Gandhi Salai,
Chennai - 600 119.

Submitted for Viva voce Examination held on 16 APRIL 2024


Internal Examiner 16/04/2024


External Examiner 16/4/24

DECLARATION

We Tanmaye Nainala (reg no: 40111319), Suchi Smitha Kolluru (reg no: 40111273) hereby declare that the Project Report entitled "HUMAN ACTIVITY RECOGNITION POSE ESTIMATION" done by us under the guidance of Dr. K. Lalitha Devi M.E, PhD is submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering degree in **Computer Science and Engineering**.

DATE: 16/04/24

PLACE: CHENNAI

Tanmaye.

SIGNATURE OF THE CANDIDATE



ACKNOWLEDGEMENT

I am pleased to acknowledge my sincere thanks to **Board of Management of SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY** for their kind encouragement in doing this project and for completing it successfully. I am grateful to them.

I convey my thanks to **Dr. T.Sasikala M.E., Ph. D, Dean**, School of Computing, **Dr. L. Lakshmanan M.E., Ph.D.**, Head of the Department of Computer Science and Engineering for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide **Dr. K. Lalitha devi M.E., Ph.D.**, for her valuable guidance, suggestions and constant encouragement paved way for the successful completion of my phase-1 project work.

I wish to express my thanks to all Teaching and Non-teaching staff members of the **Department of Computer Science and Engineering** who were helpful in many ways for the completion of the project.

ABSTRACT

This research paper presents a comprehensive comparison between Support Vector Machines (SVM) and Convolutional Neural Networks (CNN). In the domain of pattern recognition, both SVM and CNN have gained prominence in various applications including image organization, activity identification and speech identification. The study aims to provide insights into their respective strengths and weaknesses offering a nuanced understanding of their working in the provided datasets and scenarios the research methodology involves the implementation of SVM and CNN on benchmark datasets. The model represents diverse real-world challenge evaluation metrics such as accuracy, precision, recall and f1-score that help in assessing the model classification capabilities. The paper also investigates the impact of dataset characteristics size and complexity on the performance of SVM and CNN. Furthermore, the study explores the interpretability of both models shedding light on how SVM decision boundaries compare to CNNs hierarchical feature learning. Additionally, the accuracy and resource requirements of SVM and CNN are considered to address practical considerations for real-time applications. The findings of this research contribute valuable insights and help as a reference in choosing the appropriate model for human activity recognition.

TABLE OF CONTENTS

Chapter no.	Chapter name	Page no.
	Abstract	v
	List of figures	viii
	List of tables	ix
	List of Abbreviations	x
1	Introduction	1
	1.1 Background and Motivation	1
	1.2 Problem Statement	2
	1.3 Objective and Scope	2
	1.4 Applications of HAR	3
	1.5 Organization of Thesis	4
2	Literature survey	5
	2.1 Inferences from Literature Survey	6
	2.2 Open Drawbacks of Existing System	7
3	Requirement analysis	8
	3.1 Feasibility Study	8
	3.2 Requirements Specification	10
	3.3 System Use-Case	11
4	Description of proposed system	12
	4.1 Description of Research Approach	12
	Methods	
	4.2 Testing Proposed System Using CNN	18
	4.3 Pycharm	24
	4.4 Libraries Required	27
	4.5 System Architecture	27
5	Implementation details	30
	5.1 Development and Deployment of the	30
	Module	31

	5.2 Algorithms	33
	5.3 Testing	33
6	Result discussion performance analysis	35
	6.1 Result and Discussion	35
	6.2 Performance Analysis	35
	6.3 Output	37
		38
7	Summary and conclusion	40
	7.1 Summary	40
	7.2 Conclusion	41
8	References	43
9	Appendix	44
	9.1 Source code	44
	Research paper	

LIST OF FIGURES

Fig No.	Figure name	Page no.
1.1	Various applications of HAR	3
3.1	Use-case diagram	11
4.1	Image processing in CNN	12
4.2	System Architecture	29
6.1	Accuracy graph of Manual Net Architecture	35
6.2	Loss graph of Manual Net Architecture	36
6.3	Accuracy graph of Xception Net	36
6.4	Loss graph of batch Xception net	36
6.5	Accuracy graph of Lenet	37
6.6	Loss graph of Lenet	37
6.7	User interface	38
6.8	Upload the image	39
6.9	Activity detected	39

LIST OF TABLES

Table no.	Table name	Page no.
6.1	Validation accuracy and validation loss	35
6.2	Performance metrics	38

LIST OF ABBREVIATIONS

CNN – Convolutional Neural Network

LSTM – Long Short-Term Memory

HAR – Human Activity Recognition

SVM – Support Vector Machine

CHAPTER 1

INTRODUCTION

Human Activity Recognition (HAR) has emerged as a pivotal field within the realm of artificial intelligence and machine learning, aiming to decipher and comprehend human actions through the analysis of sensory data. In an era characterized by an influx of wearable devices, smart sensors, and pervasive computing, the need for robust and efficient methods to interpret human activities has become increasingly imperative. Convolutional Neural Networks (CNNs), originally designed for image recognition tasks, have proven to be exceptionally adept at extracting intricate patterns and features from sequential data, making them an ideal candidate for the challenges posed by HAR. The motivation behind the utilization of CNNs in HAR lies in their ability to automatically learn hierarchical representations of data, capturing both spatial and temporal dependencies inherent in human activities. Traditional methods for HAR often faced limitations in handling complex and dynamic movement patterns, as well as scalability issues. CNNs, by leveraging shared weights and local receptive fields, excel in recognizing spatial hierarchies within the input data, making them particularly well-suited for the diverse and intricate nature of human activity datasets. As HAR applications continue to diversify, ranging from healthcare monitoring and fitness tracking to surveillance and gesture recognition, the demand for sophisticated models capable of discerning subtle nuances in human motion has intensified. This paper explores the application of CNNs in the context of HAR, aiming to unravel the intricacies of their architecture, training methodologies and performance evaluations. By doing so, we aim to contribute to the growing body of knowledge that seeks to advance the accuracy and efficiency of human activity recognition systems, fostering applications that enhance our understanding of human behaviour in various domains.

1.1 BACKGROUND AND MOTIVATION

Human Activity Recognition (HAR) using SVM and CNN merges traditional SVM's efficiency in high-dimensional data handling with CNN's capability to extract intricate temporal and spatial features from raw sensor data. HAR is crucial in various domains like healthcare and sports analytics. SVMs excel in handling limited training samples

and overfitting, while CNNs automatically learn hierarchical representations. SVMs traditionally employ handcrafted features, whereas CNNs directly learn from raw data, bypassing the need for feature engineering. This hybrid approach aims to enhance accuracy, robustness, and generalization of HAR systems, promising significant advancements in healthcare monitoring, sports analytics, and smart environments, by capturing complex human activities effectively.

1.2 PROBLEM STATEMENT:

Despite advancements in Human Activity Recognition (HAR), existing methods face challenges in accurately identifying complex activities from sensor data. Traditional approaches relying on handcrafted features struggle to capture temporal dynamics, while deep learning techniques like Convolutional Neural Networks (CNNs) lack interpretability. Therefore, there's a need for a hybrid approach that combines the efficiency of Support Vector Machines (SVMs) with the feature learning capabilities of CNNs to improve accuracy, robustness, and generalization in HAR systems. This study aims to develop and evaluate such a hybrid SVM-CNN model to address these challenges and facilitate real-world applications in healthcare monitoring, sports analytics, and smart environments.

1.3 OBJECTIVE AND SCOPE:

1.3.1 Objective

Develop a robust Convolutional Neural Network (CNN) model for accurate and real-time recognition of diverse human activities from sensor data. Optimize the CNN architecture to efficiently extract spatial and temporal features, enhancing the model's ability to discern complex motion patterns. Investigate the generalization and scalability of the CNN-based approach across different environments and activity scenarios for reliable human activity recognition.

1.3.2 SCOPE

The scope of this study involves employing Convolutional Neural Networks (CNNs) to analyze sensor data for accurate human activity recognition. This research aims to enhance the efficiency and precision of activity recognition systems through the application of deep learning techniques, particularly focusing on CNN architectures.

1.4 APPLICATIONS OF HUMAN ACTIVITY RECOGNITION

Human Activity Recognition (HAR) has extensive applications across healthcare, sports analytics, smart environments, surveillance, as shown in fig. 1.1. In healthcare, HAR aids in remote patient monitoring, detecting falls, and ensuring timely interventions. It optimizes athletic performance by analyzing movement patterns and providing personalized feedback. Smart environments leverage HAR for ambient assisted living, enhancing independence for the elderly and individuals with disabilities. HAR revolutionizes HCI, enabling natural interaction in gaming and augmented reality. It enhances security through surveillance, identifying suspicious behaviors, and ensuring public safety. In industry, HAR improves productivity and safety by monitoring worker activities and optimizing operational efficiency. Overall, HAR significantly impacts various domains, enhancing wellness, performance, and safety.



Fig 1.1 Various applications of HAR

1.5 ORGANIZATION OF THESIS

The thesis aims to improve Human Activity Recognition (HAR) systems through a comprehensive approach. It starts with an overview of HAR, highlighting its importance and existing challenges. The literature review section explores various methods, from traditional techniques like Support Vector Machines to modern deep learning approaches like Convolutional Neural Networks. Data acquisition and preprocessing methods are discussed to ensure the quality of input data. Feature engineering techniques are then explored, including both manual and automated methods. The core of the thesis involves developing hybrid models that combine the strengths of different approaches, such as SVM, CNN, and LSTM. Evaluation metrics and experimental setups are defined to assess the performance of these models rigorously. The thesis concludes with a summary of findings and suggestions for future research directions in HAR.

CHAPTER 2

LITERATURE SURVEY

Human Activity Recognition (HAR) is a vast topic with plenty of articles and papers written on it. However, there is a lot more to discover and learn about it. From the plethora of resources available on this topic, here are some papers we took insights and techniques that helped us in our research regarding Human Activity Recognition and Pose Estimation.

Arjun S. Dileep et. al explored the theme of HAR using CNN [1]. Their model depicts the surveillance application of HAR i.e. CCTV footage in various public places like hospitals, railway stations. The model is trained to extract skeletal images from the security footage and recognize the activity. It then decides whether the activity is intrusive or non-intrusive based on the dataset it is trained on. If the action was found cautious, the user is notified immediately in the form of email or text message. We have learnt how to train and process data using a pretrained CNN model from this paper. We have adapted the feature extraction techniques from their research.

Saroja M; et. al compared various methods and approaches for Pose Estimation [2]. They compared camera-based and sensor-based input methods by using various algorithms of machine learning and deep learning. They have contrasted various combinations into 9 different possibilities and measured their efficiency rates. They have concluded that not every algorithm is suitable for camera-based estimation and same goes for sensor-based estimation as well. They have a comprehensive observation of different perceptions in a tabular format. We have concluded from their analysis what are the most feasible methods with high accuracy in real-time.

Zehua Sun, et al [3] demonstrated how the understanding of computer works in-terms of processing human activities. Modalities refer to the sources from which an action is generated. The paper provides information about different modalities used in each dataset along with their respective classes, subjects, samples and view-points. They have also considered types of neural networks used – RNN, CNN, GCN based methods. We have obtained that CNN has the most accurate schema for pose estimation, thanks to this paper. They have compared both vision-based and non-

vision-based modalities like- RGB, 3D Skeleton, Depth, IR sequence, acceleration, radar, Wi-Fi and so on for both 2D and 3D sequences.

Xiaojie Sun Hongji Xu, et. all [4] delves into HAR- a method vital for classifying human actions and movement patterns. Originating in the 1990s HAR has gained recent prominence especially with the integration of wearable sensor devices facilitated by semiconductor advances. These sensors being low-cost low-power and lightweight have led to har applications in sports medical diagnosis smart homes and computer-human reciprocation. The main insights we drew, emphasize the importance of efficient feature extraction noting the superiority of deep learning algorithms like cnn and long short-term memory lstm in comparison to original underdeveloped machine learning from early days. It introduces capsule network capsnet as a solution to cnns limitations and gated recurrent unit gru as a streamlined variant of lstm often coupled with the attention mechanism. The reseacrch concludes by highlighting the absence of public datasets for har in prison scenarios outlining the authors intention to contribute specific features for real-time recollection of aggressive activities in such settings.

2.1 INFERENCES FROM LITERATURE SURVEY

Dataset is prepared by passing pre-processed frames to Mediapipe and we generate the pose of humans for each frame when a human subject appears. The images after pre-processing. These generated pose images are further divided into two i.e, test and training set for each suspicious activity. The falling dataset is divided into two i.e, fall and not fall. Each contains test and training sets and the same for trespassing.

Compared the various approaches that have been incorporated in all the papers analyzed. The comparison of prediction activities that have been carried out in each paper. The table also shows the comparison of method by which the estimation is carried out i.e., camera based or sensor based human pose estimation, and how the estimation has been carried out and the performance.

Different modalities can have different characteristics with corresponding advantages and disadvantages. Hence numerous works exploited various modalities for HAR. Below we review the methods that use single data modalities including RGB, skeleton, depth, infrared, point cloud, event stream, audio, acceleration, radar, and WiFi, for

HAR • real-time approach for sign language detection and recognition in videos using the Holistic pose estimation method of MediaPipe.

2.2 OPEN DRAWBACKS OF EXISTING SYSTEM:

Pictorial structure estimation is a powerful tool in computer vision, but it has its limitations and drawbacks. Here are ten key drawbacks:

- **Sensitivity to Initialization:** Pictorial structure models can be sensitive to the initial configuration, making them prone to local optima during inference.
- **Computational Complexity:** The deformable spring-based model can be computationally intensive, especially for large numbers of parts, making real-time applications challenging.
- **Limited Handling of Occlusions:** Pictorial structure models may struggle with occluded objects or body parts, as they assume all parts are visible in the image.
- **Ambiguity in Symmetry:** When objects or body parts exhibit symmetry, the model may struggle to distinguish between different symmetric configurations.
- **Difficulty in Handling Variability:** Coping with substantial variability in object appearance, such as clothing changes or lighting conditions, can be challenging for the model.
- **Lack of Global Context:** Pictorial structures focus on local part relationships, often missing out on global contextual information in the scene.
- **Limited Generalization:** The model may not generalize well to objects or poses it has not seen during training.
- **Overfitting:** Pictorial structure models can be prone to overfitting if the training data is not representative or lacks diversity.
- **Manual Annotation:** Creating annotated datasets for training these models can be labor-intensive, particularly for tasks involving detailed object part labelling.
- **Limited Robustness:** Pictorial structure models may struggle when objects are viewed from unusual angles or when they are subject to significant perspective distortions.

CHAPTER 3

REQUIREMENT ANALYSIS

3.1 FEASIBILITY STUDY

Feasibility studies are crucial in determining whether a project or activity is viable, considering various aspects such as technical, economic, legal, operational, and scheduling factors. When considering human activity recognition using Convolutional Neural Networks (CNNs), several types of feasibility can be assessed:

3.1.1. Technical Feasibility:

Algorithm Suitability: Determine if CNNs are appropriate for the task of human activity recognition. CNNs are well-suited for tasks involving spatial hierarchies, making them a natural fit for image-based tasks such as activity recognition.

Data Availability: Assess if sufficient labeled data for training CNN models is available. Human activity recognition typically requires large datasets with accurately labeled activities.

Computational Resources: Evaluate whether the computational resources required for training and inference (such as GPUs or TPUs) are available and feasible for the intended application.

3.1.2. Economic Feasibility:

Cost of Development: Estimate the costs associated with developing CNN-based models for human activity recognition, including data collection, labelling, model development, and computational resources.

Cost of Deployment: Consider the costs of deploying the system in the intended environment, including hardware requirements, maintenance, and any ongoing operational expenses.

3.1.3 Operational Feasibility:

User Acceptance: Assess whether end-users will accept and adopt the CNN-based system for activity recognition. Factors such as ease of use, reliability, and performance play a significant role in user acceptance.

Integration: Determine if the CNN-based system can be seamlessly integrated into existing workflows or systems.

Scalability: Evaluate whether the system can scale to accommodate future growth in data or user base.

3.1.4. Legal and Ethical Feasibility:

Data Privacy: Ensure that the collection and use of data for training and deploying the CNN models comply with relevant privacy regulations and ethical standards.

Bias and Fairness: Mitigate potential biases in the data or models that could lead to unfair treatment of certain individuals or groups.

Regulatory Compliance: Ensure compliance with any legal or regulatory requirements governing the deployment of AI systems, particularly in sensitive domains such as healthcare or surveillance.

3.1.5. Schedule Feasibility:

Time to Deployment: Estimate the time required to develop, train, and deploy the CNN-based system for human activity recognition.

Project Milestones: Identify key milestones and deliverables to track progress and ensure timely completion of the project.

3.2 REQUIREMENTS SPECIFICATION:

Requirements are the basic constraints that are required to develop a system. Requirements are collected while designing the system. The following are the requirements that are to be discussed.

1. Functional requirements
2. Non-Functional requirements
3. Environmental requirements

3.2.1 Functional Requirements:

The software requirements specification is a technical specification of requirements for the software product. It is the first step in the requirements analysis process. It lists requirements of a particular software system. The following details to follow the special libraries like sk-learn, pandas, numpy, matplotlib and seaborn.

3.2.2 Non-Functional Requirements:

Process of functional steps,

1. Problem definition
2. Preparing data
3. Evaluating algorithms
4. Improving results
5. Prediction the result

3.2.3 Environmental Requirements:

1. Software Requirements:

Operating System : Windows

Tool : Anaconda with Jupyter Notebook

2. Hardware requirements:

Processor : Intel core i3

Hard disk : minimum 300 GB

RAM : minimum 4 GB

3.3 SYSTEM USE-CASE

Use case diagrams are considered for high level requirement analysis of a system. So, when the requirements of a system are analyzed the functionalities are captured in use cases. So, as shown in fig. 3.1., it can say that uses cases are nothing but the system functionalities written in an organized manner.

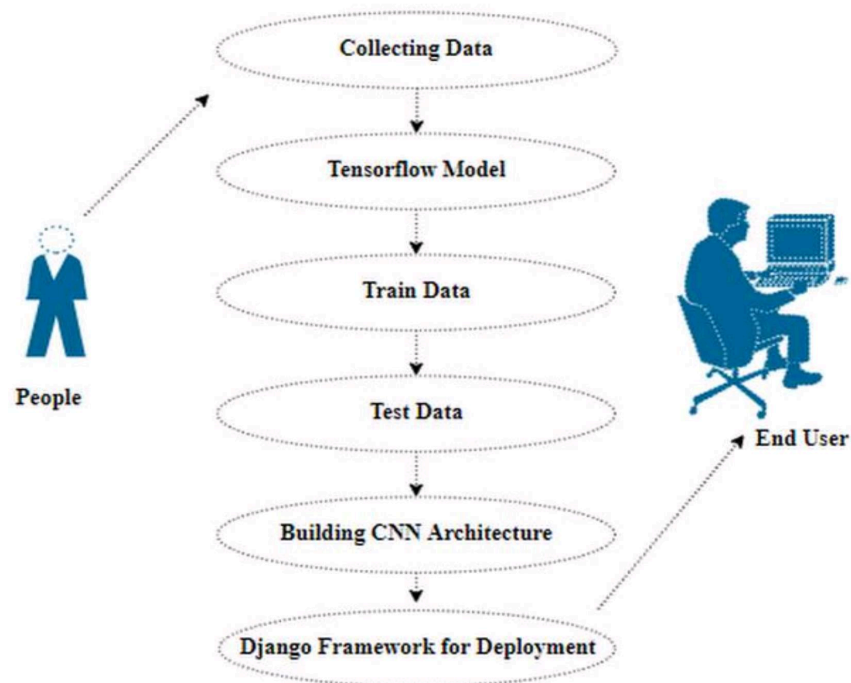


Fig. 3.1 Use-case diagram

CHAPTER 4

DESCRIPTION OF PROPOSED SYSTEM

4.1 DESCRIPTION OF RESEARCH APPROACH METHODS:

Pre-processing and training the model (CNN): The dataset is pre-processed such as Image reshaping, resizing and conversion to an array form. Similar processing is also done on the test image.

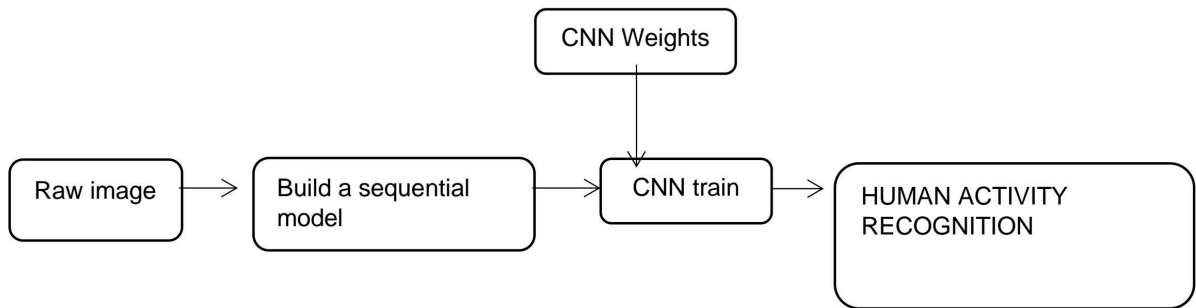


Fig 4.1 Image processing in CNN

Fig 4.1 represents the image processing involved in CNN. The train dataset is used to train the model (CNN) so that it can identify the test image and the severity it has CNN has different layers that are Dense, Dropout, Activation, Flatten, Convolution2D, and MaxPooling2D. After the model is trained successfully, the software can identify the Paddy Leaves Classification image contained in the dataset. After successful training and preprocessing, comparison of the test image and trained model takes place to predict the knee osteoarthritis

4.1.1 CNN Model steps:

- **Conv2d:** The most common type of convolution that is used is the 2D convolution layer and is usually abbreviated as conv2D.

The 2D convolution is a fairly simple operation at heart: you start with a kernel, which is simply a small matrix of weights. This kernel “slides” over the 2D input data,

performing an elementwise multiplication with the part of the input it is currently on, and then summing up the results into a single output pixel.

The kernel repeats this process for every location it slides over, converting a 2D matrix of features into yet another 2D matrix of features. The output features are essentially, the weighted sums (with the weights being the values of the kernel itself) of the input features located roughly in the same location of the output pixel on the input layer.

Whether or not an input feature falls within this “roughly same location”, gets determined directly by whether it’s in the area of the kernel that produced the output or not. This means the size of the kernel directly determines how many (or few) input features get combined in the production of a new output feature.

This is all in pretty stark contrast to a fully connected layer. In the above example, we have $5 \times 5 = 25$ input features, and $3 \times 3 = 9$ output features. If this were a standard fully connected layer, you’d have a weight matrix of $25 \times 9 = 225$ parameters, with every output feature being the weighted sum of every single input feature. Convolutions allow us to do this transformation with only 9 parameters, with each output feature, instead of “looking at” every input feature, only getting to “look” at input features coming from roughly the same location. Do take note of this, as it’ll be critical to our later discussion.

- **MaxPooling2D layer:** Down samples the input along its spatial dimensions (height and width) by taking the maximum value over an input window (of size defined by `pool_size`) for each channel of the input. The window is shifted by strides along each dimension.

The resulting output, when using the "valid" padding option, has a spatial shape (number of rows or columns) of: $\text{output_shape} = \text{math.floor}((\text{input_shape} - \text{pool_size}) / \text{strides}) + 1$ (when $\text{input_shape} \geq \text{pool_size}$)

The resulting output shape when using the "same" padding option is: $\text{output_shape} = \text{math.floor}((\text{input_shape} - 1) / \text{strides}) + 1$

The following are the arguments included:

- `pool_size`: integer or tuple of 2 integers, window size over which to take the maximum. (2, 2) will take the max value over a 2x2 pooling window. If only one integer is specified, the same window length will be used for both dimensions.
 - `strides`: Integer, tuple of 2 integers, or None. Strides values. Specifies how far the pooling window moves for each pooling step. If None, it will default to `pool_size`.
 - `padding`: One of "valid" or "same" (case-insensitive). "valid" means no padding. "same" results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.
 - `data_format`: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape (batch, height, width, channels) while `channels_first` corresponds to inputs with shape (batch, channels, height, width). It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
-
- Input shape:
 - If `data_format='channels_last'`: 4D tensor with shape (batch_size, rows, cols, channels).
 - If `data_format='channels_first'`: 4D tensor with shape (batch_size, channels, rows, cols)
 - Output shape
 - If `data_format='channels_last'`: 4D tensor with shape (batch_size, pooled_rows, pooled_cols, channels).
 - If `data_format='channels_first'`: 4D tensor with shape (batch_size, channels, pooled_rows, pooled_cols).

- **Flatten layer:** It is used to flatten the dimensions of the image obtained after convolving it. **Dense:** It is used to make this a fully connected model and is the hidden layer. **Dropout:** It is used to avoid over fitting on the dataset and dense is the output layer contains only one neuron which decide to which category image belongs.

Flatten is used to flatten the input. For example, if flatten is applied to layer having input shape as (batch_size, 2,2), then the output shape of the layer will be (batch_size, 4)

Flatten has one argument as follows

```
keras.layers.Flatten(data_format = None)
```

data_format is an optional argument and it is used to preserve weight ordering when switching from one data format to another data format. It accepts either channels_last or channels_first as value. channels_last is the default one and it identifies the input shape as (batch_size, ..., channels) whereas channels_first identifies the input shape as (batch_size, channels, ...)

- **Dense layer:** Dense implements the operation: $\text{output} = \text{activation}(\text{dot}(\text{input}, \text{kernel}) + \text{bias})$ where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer (only applicable if use_bias is True). These are all attributes of Dense.

Note: If the input to the layer has a rank greater than 2, then Dense computes the dot product between the inputs and the kernel along the last axis of the inputs and axis 0 of the kernel (using tf.tensordot). For example, if input has dimensions (batch_size, d0, d1), then we create a kernel with shape (d1, units), and the kernel operates along axis 2 of the input, on every sub-tensor of shape (1, 1, d1) (there are batch_size * d0 such sub-tensors). The output in this case will have shape (batch_size, d0, units).

Besides, layer attributes cannot be modified after the layer has been called once (except the trainable attribute). When a popular kwarg `input_shape` is passed, then keras will create an input layer to insert before the current layer. This can be treated equivalent to explicitly defining an `InputLayer`.

The following are the arguments included:

- `units`: Positive integer, dimensionality of the output space.
- `activation`: Activation function to use. If you don't specify anything, no activation is applied (ie. "linear" activation: $a(x) = x$).
- `use_bias`: Boolean, whether the layer uses a bias vector.
- `kernel_initializer`: Initializer for the kernel weights matrix.
- `bias_initializer`: Initializer for the bias vector.
- `kernel_regularizer`: Regularizer function applied to the kernel weights matrix.
- `bias_regularizer`: Regularizer function applied to the bias vector.
- `activity_regularizer`: Regularizer function applied to the output of the layer (its "activation").
- `kernel_constraint`: Constraint function applied to the kernel weights matrix.
- `bias_constraint`: Constraint function applied to the bias vector.

Input shape: N-D tensor with shape: `(batch_size, ..., input_dim)`. The most common situation would be a 2D input with shape `(batch_size, input_dim)`. Output shape: N-D tensor with shape: `(batch_size, ..., units)`. For instance, for a 2D input with shape `(batch_size, input_dim)`, the output would have shape `(batch_size, units)`.

- **Dropout layer:** The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by $1/(1 - \text{rate})$ such that the sum over all inputs is unchanged.

Note that the Dropout layer only applies when `training` is set to `True` such that no values are dropped during inference. When using `model.fit`, `training` will

be appropriately set to True automatically, and in other contexts, you can set the kwarg explicitly to True when calling the layer.

(This is in contrast to setting `trainable=False` for a Dropout layer. `trainable` does not affect the layer's behavior, as Dropout does not have any variables/weights that can be frozen during training.)

The following are the arguments included:

rate: Float between 0 and 1. Fraction of the input units to drop.

noise_shape: 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input. For instance, if your inputs have shape (batch_size, timesteps, features) and you want the dropout mask to be the same for all timesteps, you can use `noise_shape=(batch_size, 1, features)`.

seed: A Python integer to use as random seed.

- ***Image Data Generator:***

It is that rescales the image, applies shear in some range, zooms the image and does horizontal flipping with the image. This Image Data Generator includes all possible orientation of the image.

- ***Training Process:***

`train_datagen.flow_from_directory` is the function that is used to prepare data from the `train_dataset` directory. `Target_size` specifies the target size of the image. `Test_datagen.flow_from_directory` is used to prepare test data for the model and all is similar as above. `fit_generator` is used to fit the data into the model made above, other factors used are `steps_per_epochs` tells us about the number of times the model will execute for the training data.

- ***Epochs:***

It tells us the number of times model will be trained in forward and backward pass.

- ***Validation process:***

Validation_data is used to feed the validation/test data into the model.

Validation_steps denote the number of validation/test samples.

4.2 TESTING PROPOSED SYSTEM USING CONVOLUTIONAL NEURAL NETWORK:

A Convolutional neural network (CNN) is one type of Artificial Neural Network. A Convolutional neural network (CNN) is a neural network that has one or more convolutional layers and are used mainly for image processing, classification, segmentation and also for other auto correlated data.

4.2.1 Models API:

There are three ways to create Keras models:

- The Sequential model, which is very straightforward (a simple list of layers), but is limited to single-input, single-output stacks of layers (as the name gives away).
- The Functional API, which is an easy-to-use, fully-featured API that supports arbitrary model architectures. For most people and most use cases, this is what you should be using. This is the Keras "industry strength" model.
- Model subclassing, where you implement everything from scratch on your own. Use this if you have complex, out-of-the-box research use cases.

Types of Keras Models

Models in keras are available in two types:

- Keras Sequential Model
- Keras Functional API

4.2.2 Sequential Model in Keras

It allows us to create models layer by layer in sequential order. But it does not allow us to create models that have multiple inputs or outputs.

It is best for simple stack of layers which have 1 input tensor and 1 output tensor.

This model is not suited when any of the layer in the stack has multiple inputs or outputs. Even if we want non-linear topology, it is not suited.

4.2.3 Functional API in Keras

It provides more flexibility to define a model and add layers in keras. Functional API allows us to create models that have multiple input or output.

It also allows us to share these layers. In other words. we can make graphs of layers using Keras functional API.

As functional API is a data structure, it is easy to save it as a single file that helps in recreating the exact model without having the original code. Also, its easy to model the graph here and access its nodes as well.

Model Subclassing in Keras process involved the following. Sequential model does not allow you much flexibility to create your models. Functional API also only has a little of customization available for you. But you may create your own fully-customizable models in Keras. This is done by subclassing the Model class and implementing a call method. `Input()` is used to instantiate a Keras tensor. A Keras tensor is a symbolic tensor-like object, which we augment with certain attributes that allow us to build a Keras model just by knowing the inputs and outputs of the model. For instance, if `a`, `b` and `c` are Keras tensors, it becomes possible to do: `model = Model(input=[a, b], output=c)`

- **Kernels:** Each convolutional layer contains a series of filters known as convolutional kernels. The filter is a matrix of integers that are used on a subset of the input pixel values, the same size as the kernel. Each pixel is multiplied by the

corresponding value in the kernel, then the result is summed up for a single value for simplicity representing a grid cell, like a pixel, in the output channel/feature map. These are linear transformations; each convolution is a type of affine function.

In computer vision the input is often a 3 channel RGB image. For simplicity, if we take a greyscale image that has one channel (a two-dimensional matrix) and a 3x3 convolutional kernel (a two-dimensional matrix). The kernel strides over the input matrix of numbers moving horizontally column by column, sliding/scanning over the first rows in the matrix containing the images pixel values. Then the kernel strides down vertically to subsequent rows. Note, the filter may stride over one or several pixels at a time, this is detailed further below.

In other non-vision applications, a one-dimensional convolution may slide vertically over an input matrix.

To handle the edge pixels there are several approaches:

Losing the edge pixels, Padding with zero value pixels, Reflection padding

Reflection padding is by far the best approach, where the number of pixels needed for the convolutional kernel to process the edge pixels are added onto the outside copying the pixels from the edge of the image. For a 3x3 kernel, one pixel needs to be added around the outside, for a 7x7 kernel then three pixels would be reflected around the outside. The pixels added around each side is the dimension, halved and rounded down.

Traditionally in many research papers, the edge pixels are just ignored, which loses a small proportion of the data and this gets increasing worse if there are many deep convolutional layers. For this reason, I could not find existing diagrams to easily convey some of the points here without being misleading and confusing stride 1 convolutions with stride 2 convolutions.

With padding, the output from a input of width w and height h would be width w and height h (the same as the input with a single input channel), assuming the kernel takes a stride of one pixel at a time.

- **Strides:** It is common to use a stride two convolution rather than a stride one convolution, where the convolutional kernel strides over 2 pixels at a time, for example our 3x3 kernel would start at position (1,1), then stride to (1,3), then to (1,5) and so on, halving the size of the output channel/feature map, compared to the convolutional kernel taking strides of one.

With padding, the output from an input of width w , height h and depth 3 would be the ceiling of width $w/2$, height $h/2$ and depth 1, as the kernel outputs a single summed output from each stride.

- **Rectified Linear Unit (ReLU):** A Rectified Linear Unit is used as a non-linear activation function. A ReLU says if the value is less than zero, round it up to zero.

Normalisation is the process of subtracting the mean and dividing by the standard deviation. It transforms the range of the data to be between -1 and 1 making the data use the same scale, sometimes called Min-Max scaling.

It is common to normalize the input features, standardising the data by removing the mean and scaling to unit variance. It is often important the input features are centred around zero and have variance in the same order. With some data, such as images the data is scaled so that it's range is between 0 and 1, most simply dividing the pixel values by 255.

Batch normalisation has the benefits of helping to make a network output more stable predictions, reduce overfitting through regularisation and speeds up training by an order of magnitude.

Batch normalisation is the process of carrying normalisation within the scope activation layer of the current batch, subtracting the mean of the batch's activations and dividing by the standard deviation of the batches activations. This is necessary as even after normalizing the input as some activations can be higher, which can cause the subsequent layers to act abnormally and makes the network less stable. Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

Importantly, batch normalization works differently during training and during inference.

During training (i.e. when using `fit()` or when calling the layer/model with the argument `training=True`), the layer normalizes its output using the mean and standard deviation of the current batch of inputs. That is to say, for each channel being normalized, the layer returns $\gamma * (\text{batch} - \text{mean}(\text{batch})) / \sqrt{\text{var}(\text{batch}) + \text{epsilon}} + \text{beta}$, where:

- `epsilon` is small constant (configurable as part of the constructor arguments)
- `gamma` is a learned scaling factor (initialized as 1), which can be disabled by passing `scale=False` to the constructor.
- `beta` is a learned offset factor (initialized as 0), which can be disabled by passing `center=False` to the constructor.

During inference (i.e. when using `evaluate()` or `predict()` or when calling the layer/model with the argument `training=False` (which is the default), the layer normalizes its output using a moving average of the mean and standard deviation of the batches it has seen during training. That is to say, it returns $\gamma * (\text{batch} - \text{self.moving_mean}) / \sqrt{\text{self.moving_var} + \text{epsilon}} + \text{beta}$.

`self.moving_mean` and `self.moving_var` are non-trainable variables that are updated each time the layer is called in training mode, as such:

- $\text{moving_mean} = \text{moving_mean} * \text{momentum} + \text{mean}(\text{batch}) * (1 - \text{momentum})$
- $\text{moving_var} = \text{moving_var} * \text{momentum} + \text{var}(\text{batch}) * (1 - \text{momentum})$

As such, the layer will only normalize its inputs during inference *after having been trained on data that has similar statistics as the inference data*.

The following are the arguments:

- **axis**: Integer, the axis that should be normalized (typically the features axis).
For `BatchNormalization` instance, `axis` is the axis after a `Conv2D` layer with `data_format="channels_first"`, set `axis=1` in `BatchNormalization`.
- **momentum**: Momentum for the moving average.
- **epsilon**: Small float added to variance to avoid dividing by zero.
- **center**: If True, add offset of `beta` to normalized tensor. If False, `beta` is ignored.
- **scale**: If True, multiply by `gamma`. If False, `gamma` is not used. When the next layer is linear (also e.g. `nn.relu`), this can be disabled since the scaling will be done by the next layer.
- **beta_initializer**: Initializer for the beta weight.
- **gamma_initializer**: Initializer for the gamma weight.
- **moving_mean_initializer**: Initializer for the moving mean.
- **moving_variance_initializer**: Initializer for the moving variance.
- **beta_regularizer**: Optional regularizer for the beta weight.
- **gamma_regularizer**: Optional regularizer for the gamma weight.
- **beta_constraint**: Optional constraint for the beta weight.
- **gamma_constraint**: Optional constraint for the gamma weight.

The following are the call arguments:

- **inputs**: Input tensor (of any rank).
- **training**: Python boolean indicating whether the layer should behave in training mode or in inference mode.

4.3 PYCHARM:

PyCharm is a dedicated Python Integrated Development Environment (IDE) providing a wide range of essential tools for Python developers, tightly integrated to create a convenient environment for productive Python, web, and data science development. Code faster and with more easily in a smart and configurable editor with code completion, snippets, code folding and split windows support.

4.3.1 PyCharm Features

Intelligent Coding Assistance – PyCharm provides smart code completion, code inspections, on-the-fly error highlighting and quick-fixes, along with automated code refactorings and rich navigation capabilities.

Intelligent Code Editor – PyCharm's smart code editor provides first-class support for Python, JavaScript, CoffeeScript, TypeScript, CSS, popular template languages and more. Take advantage of language-aware code completion, error detection, and on-the-fly code fixes!

- **Smart Code Navigation** – Use smart search to jump to any class, file or symbol, or even any IDE action or tool window. It only takes one click to switch to the declaration, super method, test, usages, implementation, and more.
- **Fast and Safe Refactorings** – Refactor your code the intelligent way, with safe Rename and Delete, Extract Method, Introduce Variable, Inline Variable or Method, and other refactorings. Language and framework-specific refactorings help you perform project-wide changes.
- **Built-in Developer Tools** – PyCharm's huge collection of tools out of the box includes an integrated debugger and test runner; Python profiler; a built-in terminal; integration with major VCS and built-in database tools; remote development capabilities with remote interpreters; an integrated ssh terminal; and integration with Docker and Vagrant.
- **Debugging, Testing and Profiling** – Use the powerful debugger with a graphical UI for Python and JavaScript. Create and run your tests with coding assistance and a GUI-based test runner. Take full control of your code with Python Profiler integration.

- **VCS, Deployment and Remote Development** – Save time with a unified UI for working with Git, SVN, Mercurial or other version control systems. Run and debug your application on remote machines. Easily configure automatic deployment to a remote host or VM and manage your infrastructure with Vagrant and Docker.
- **Database tools** – Access Oracle, SQL Server, PostgreSQL, MySQL and other databases right from the IDE. Rely on PyCharm's help when editing SQL code, running queries, browsing data, and altering schemas.
- **Web Development** – In addition to Python, PyCharm provides first-class support for various Python web development frameworks, specific template languages, JavaScript, CoffeeScript, TypeScript, HTML/CSS, AngularJS, Node.js, and more.
- **Python Web frameworks** – PyCharm offers great framework-specific support for modern web development frameworks such as Django, Flask, Google App Engine, Pyramid, and web2py, including Django templates debugger, manage.py and appcfg.py tools, special auto completion and navigation, just to name a few.
- **JavaScript & HTML** – PyCharm provides first-class support for JavaScript, Coffee Script, Type Script, HTML and CSS, as well as their modern successors. The JavaScript debugger is included in PyCharm and is integrated with the Django server run configuration.
- **Live Edit** – Live Editing Preview lets you open a page in the editor and the browser and see the changes being made in code instantly in the browser. PyCharm auto-saves your changes, and the browser smartly updates the page on the fly, showing your edits.
- **Scientific Tools** – PyCharm integrates with Ipython Notebook, has an interactive Python console, and supports Anaconda as well as multiple scientific packages including Matplotlib and NumPy.
- **Interactive Python console** – You can run a REPL Python console in PyCharm which offers many advantages over the standard one: on-the-fly syntax check with inspections, braces and quotes matching, and of course code completion.

- **Scientific Stack Support** – PyCharm has built-in support for scientific libraries. It supports Pandas, Numpy, Matplotlib, and other scientific libraries, offering you best-in-class code intelligence, graphs, array viewers and much more.
- **Conda Integration** – Keep your dependencies isolated by having separate Conda environments per project, PyCharm makes it easy for you to create and select the right environment.
- **Customizable and Cross-platform IDE** – Use PyCharm on Windows, Mac OS and Linux with a single license key. Enjoy a fine-tuned workspace with customizable color schemes and key-bindings, with VIM emulation available.
- **Customizable UI** – Enjoy a fine-tuned workspace with customizable color schemes and key-bindings.
- **Plugins** – More than 10 years of IntelliJ platform development gives PyCharm 50+ IDE plugins of different nature, including support for additional VCS, integrations with different tools and frameworks, and editor enhancements such as Vim emulation.
- **Cross-platform IDE** – PyCharm works on Windows, Mac OS or Linux. You can install and run PyCharm on as many machines as you have, and use the same environment and functionality across all your machines.

4.4 LIBRARIES REQUIRED

- **tensorflow**: Just to use the tensor board to compare the loss and adam curve our result data or obtained log.
- **keras**: To pre-process the image dataset.
- **matplotlib**: To display the result of our predictive outcome.
- **os**: To access the file system to read the image from the train and test directory from our machines.

4.5 SYSTEM ARCHITECTURE

The system architecture of Convolutional Neural Networks (CNNs), represented in fig. 4.2., is a crucial aspect of their design, responsible for their remarkable success in various computer vision tasks. CNNs are a class of deep neural networks specifically tailored for processing structured grid data, such as images. Here's a concise summary of their architecture:

4.5.1. Input Layer:

The input layer of a CNN receives the raw input data, typically images represented as pixel values. The dimensions of the input layer correspond to the dimensions of the input image.

4.5.2. Convolutional Layers:

Convolutional layers are the core building blocks of CNNs. They consist of a set of learnable filters or kernels. Each filter is small spatially but extends through the full depth of the input volume. During the forward pass, these filters convolve across the input image, computing dot products between the entries of the filter and the input at every position. This operation generates feature maps that represent the presence of specific features or patterns in the input. Multiple filters are used in each convolutional layer, enabling the network to learn various features at different spatial locations.

4.5.3. Activation Function:

Typically, an activation function like ReLU (Rectified Linear Unit) follows each convolutional operation. ReLU introduces non-linearity, enabling the network to learn complex patterns and relationships within the data.

4.5.4. Pooling Layers:

Pooling layers are interspersed between convolutional layers to reduce the spatial dimensions of the feature maps while retaining the most important information.

Common pooling operations include max-pooling and average-pooling, which down-sample the feature maps by taking the maximum or average value within a specific window.

4.5.5. Fully Connected Layers:

Following several convolutional and pooling layers, the feature maps are flattened into a vector and fed into one or more fully connected layers. These layers perform high-level reasoning on the extracted features and eventually produce the network's output, often through a softmax activation function for classification tasks.

4.5.6. Output Layer:

The output layer produces the final prediction or classification based on the information processed through the preceding layers. The number of neurons in this layer corresponds to the number of classes in the classification task. Overall, the system architecture of CNNs enables them to automatically learn hierarchical representations of data, making them powerful tools for image recognition, object detection, and various other visual tasks.

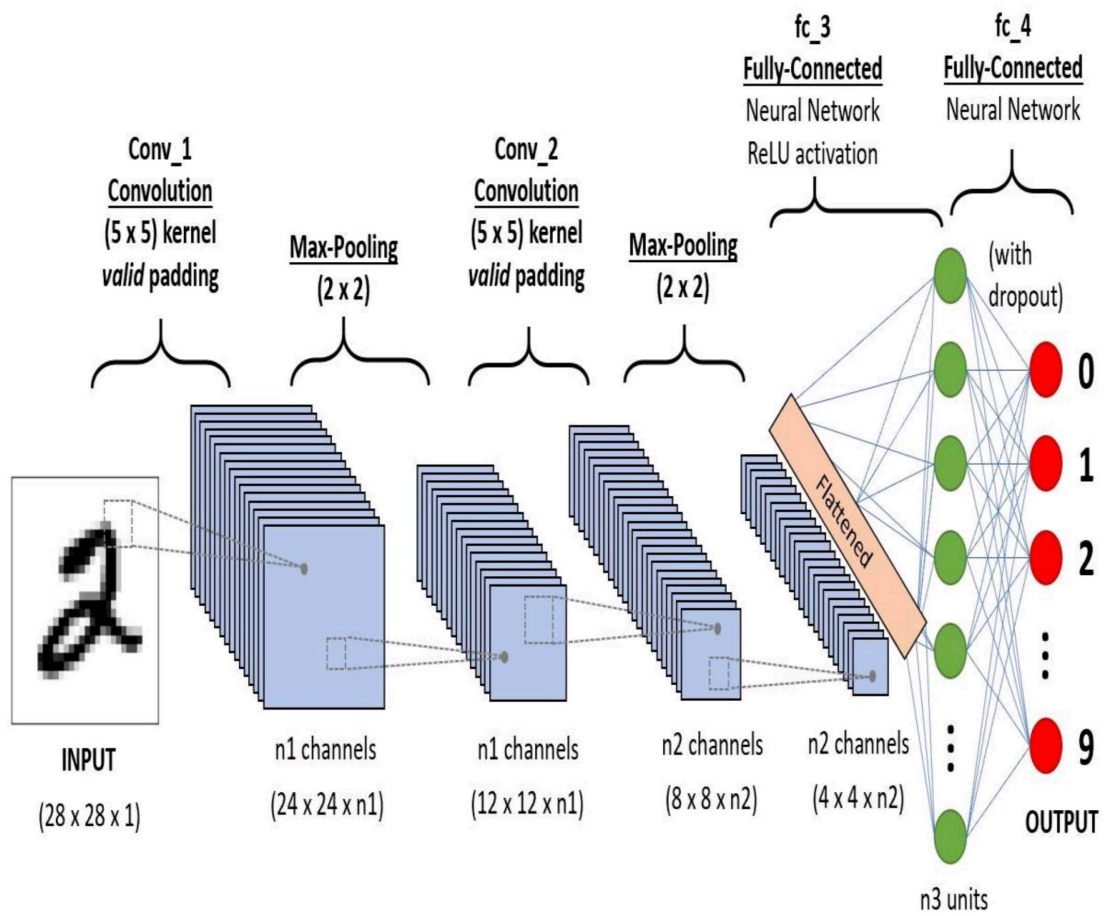


Fig. 4.2 System Architecture

CHAPTER 5

IMPLEMENTATION DETAILS

5.1 DEVELOPMENT AND DEPLOYMENT OF THE MODULE:

To implement the KPEDCNN (Key Point Extraction with Deep Convolutional Neural Networks) model for human action recognition, you would need to follow several steps. Below is a highlevel overview of the implementation process:

5.1.1. Data Preparation:

Obtain a dataset containing videos of human activities along with their corresponding labels (i.e., the activities being performed in each video). Split the dataset into training, validation, and testing sets.

5.1.2. Key Point Extraction:

Use an OpenPose model to extract accurate keypoints from each frame of the input videos. OpenPose is a popular library for multiperson keypoint detection in images and videos. The keypoint extraction process identifies the locations of specific body joints (such as shoulders, elbows, wrists, etc.) in each frame, capturing the human pose.

5.1.3. Hyperparameter Tuning:

Define the hyperparameters for training the Deep Convolutional Neural Network (DCNN) model. This includes parameters such as learning rate, batch size, and number of epochs Use techniques like grid search or random search to find the optimal combination of hyperparameters. RMSProp optimizer can be employed to adjust the hyperparameters effectively during training.

5.1.4. Pose Estimation:

Design and implement a DCNN model architecture optimized for human action recognition based on the extracted keypoints. Train the DCNN model on the extracted keypoints and corresponding activity labels using the hyperparameters determined in the tuning stage. The DCNN model learns to classify human activities based on the spatial relationships and patterns in the extracted keypoints.

5.1.5. Evaluation:

Evaluate the trained KPEDCNN model on the testing set to assess its performance. Measure performance metrics such as accuracy, precision, recall, F1score, and confusion matrix to quantify the model's effectiveness in recognizing human actions.

5.1.6. Comparison with Benchmark Algorithms:

Compare the performance of the KPEDCNN model with benchmark algorithms such as CNN, DBN, SVM, STAL, TCNN, etc., using the same dataset and evaluation metrics. Analyze the experimental results to determine whether the proposed KPEDCNN technique outperforms existing algorithms in terms of accuracy and robustness.

5.1.7. Fine-Tuning and Optimization:

Finetune the KPEDCNN model based on insights gained from the experimental results to further improve its performance. Optimize the model architecture and hyperparameters as needed to achieve better results.

5.1.8. Deployment:

Once satisfied with the performance of the KPEDCNN model, deploy it for realworld applications such as intelligent surveillance, humanrobot interaction, or robot vision. Integrate the model into the target environment, ensuring compatibility with hardware and software requirements.

By following these steps and iteratively refining the implementation process, you can develop an effective and accurate KPEDCNN model for human action recognition, as described in the study.

5.2 ALGORITHMS:

5.2.1 Convolutional Neural Network

In the context of human activity recognition, CNNs typically operate on sequential data, such as video frames or sensor readings over time. The following steps outline how CNNs can be used for human activity recognition:

- **Data Preprocessing:** Input data, such as video sequences or sensor readings, are preprocessed to extract relevant features and prepare them for input to the

CNN model. This may involve resizing images, normalizing sensor data, or applying transformations to enhance feature representation.

- **Model Architecture:** A CNN architecture suitable for sequential data, such as 3D CNNs or temporal convolutional networks (TCNs), is chosen or designed. These architectures are capable of capturing temporal dependencies in the input data across multiple time steps.
- **Training:** The CNN model is trained on a labeled dataset of human activities. During training, the model learns to extract spatial and temporal features from the input data and map them to corresponding activity labels. This is typically done using supervised learning techniques, where the model is optimized to minimize a suitable loss function, such as categorical crossentropy.
- **Evaluation:** The trained model is evaluated on a separate test dataset to assess its performance in recognizing human activities. Performance metrics such as accuracy, precision, recall, and F1score are computed to quantify the model's effectiveness.
- **Deployment:** Once the CNN model demonstrates satisfactory performance, it can be deployed for realtime human activity recognition in the target environment. This may involve integrating the model into existing systems or deploying it on dedicated hardware platforms, depending on the application requirements.

Overall, CNNs offer a powerful and flexible approach to human activity recognition, capable of capturing complex spatial and temporal patterns in input data. By carefully considering various feasibility factors and employing appropriate CNN architectures and training techniques, effective systems for human activity recognition can be developed and deployed in a wide range of applications, including healthcare, security, and humancomputer interaction.

5.3 TESTING:

A testing dataset is a separate subset of data used to evaluate the performance of a machine learning model after it has been trained on a training dataset. The testing dataset should ideally represent the same distribution as the realworld data that the model will encounter during deployment. Here's how you can create or obtain a testing dataset:

5.3.1. Splitting from the Original Dataset:

If you have a single dataset containing both training and testing samples, you can split it into two parts: one for training and one for testing. The usual split ratios are 7030, 8020, or 9010, with the larger portion used for training. Ensure that the split maintains the distribution of classes or labels across both subsets.

5.3.2. Using Public Datasets:

Many machine learning and deep learning tasks have publicly available benchmark datasets with predefined training and testing splits. Examples include MNIST, CIFAR10, CIFAR100, ImageNet, and UCI Machine Learning Repository datasets. You can download these datasets and use the provided testing subset for evaluation.

5.3.3. Data Collection:

If you're working on a specific problem domain, you may need to collect your own testing data. Ensure that the testing data is collected under similar conditions as the training data to maintain consistency. Label the testing data appropriately for evaluation.

5.3.4. Cross-Validation:

In cases where you have limited data, or you want to ensure robustness of the model, you can perform crossvalidation instead of having a separate testing dataset. Crossvalidation involves splitting the dataset into multiple subsets (folds), training the model on some folds, and testing it on others. This allows you to evaluate the model's performance on multiple subsets of the data and obtain more reliable performance estimates.

5.3.5. Data Preprocessing:

Preprocess the testing dataset similarly to how you preprocessed the training dataset. This may include tasks such as resizing images, normalizing pixel values, handling missing values, and encoding categorical variables.

5.3.6. Validation vs. Testing:

It's important to distinguish between validation and testing datasets. The validation dataset is used during model training to tune hyperparameters and monitor performance, while the testing dataset is used only after the model has been trained to assess its generalization ability.

5.3.7. Maintaining Data Integrity:

Ensure that the testing dataset remains untouched during the model development process. Avoid any form of data leakage where information from the testing dataset inadvertently influences the training process.

By following these steps, you can obtain or create a testing dataset suitable for evaluating the performance of your machine learning or deep learning model effectively. This ensures that your model's performance estimates are reliable and indicative of its realworld performance.

CHAPTER 6

RESULT, DISCUSSION, PERFORMANCE ANALYSIS

6.1 RESULT AND DISCUSSION:

Module name	Dataset size	No. of epochs	Val_accuracy	Val_loss
Manual Net	512	10	0.0729	2.7055
Xception	512	100	0.2083	1.7225
Lenet	32	100	0.9899	0.0357

Table 6.1 Validation accuracy and validation loss

6.1.1 Graphs of Batches:

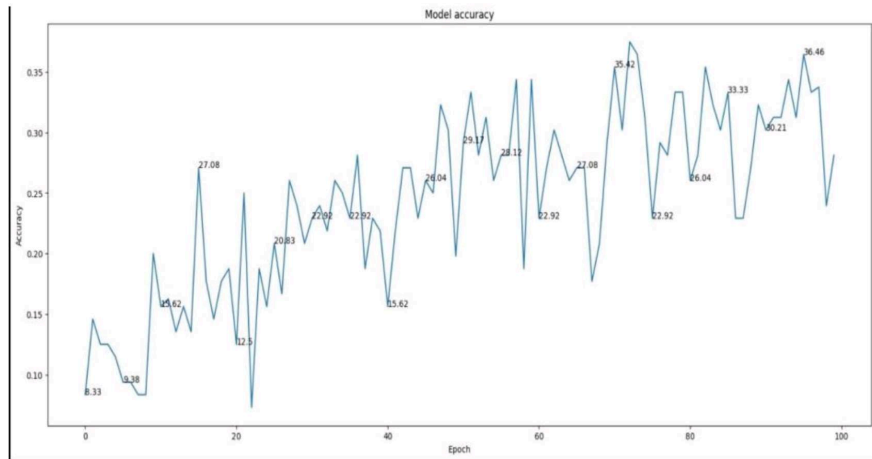


Fig 6.1 Accuracy graph of manual net

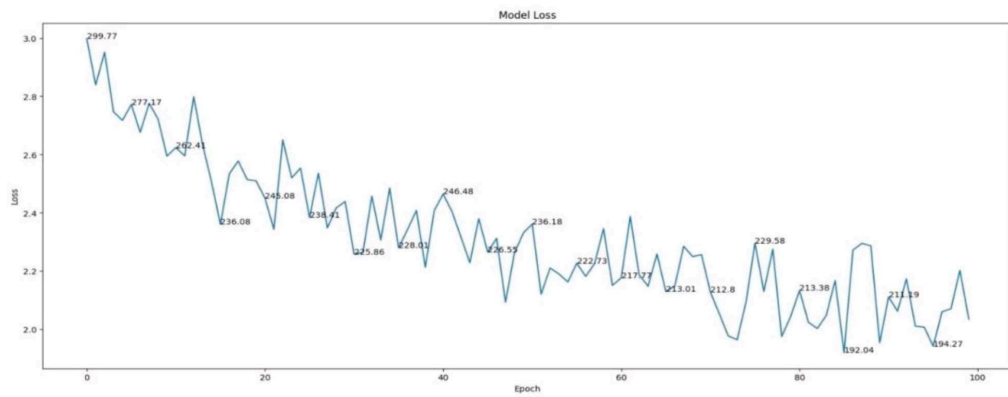


Fig 6.2 Loss Graph of Manual Net

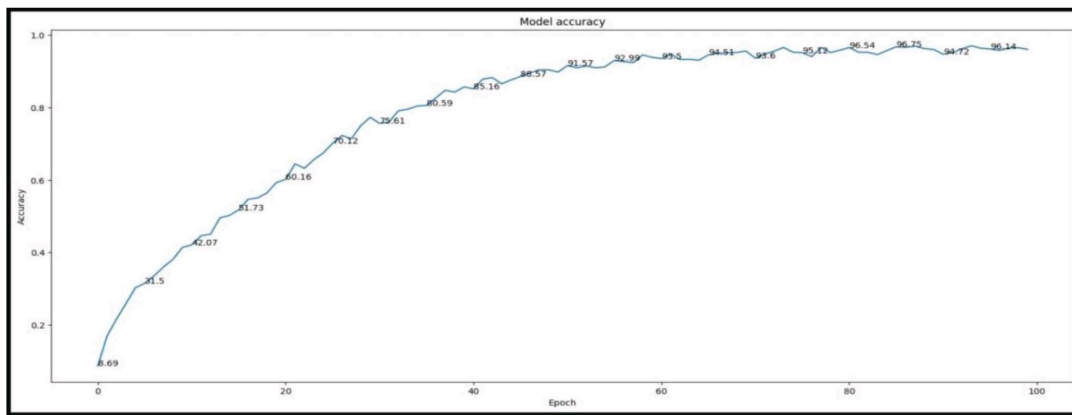


Fig 6.3 Accuracy Graph of Xception Net

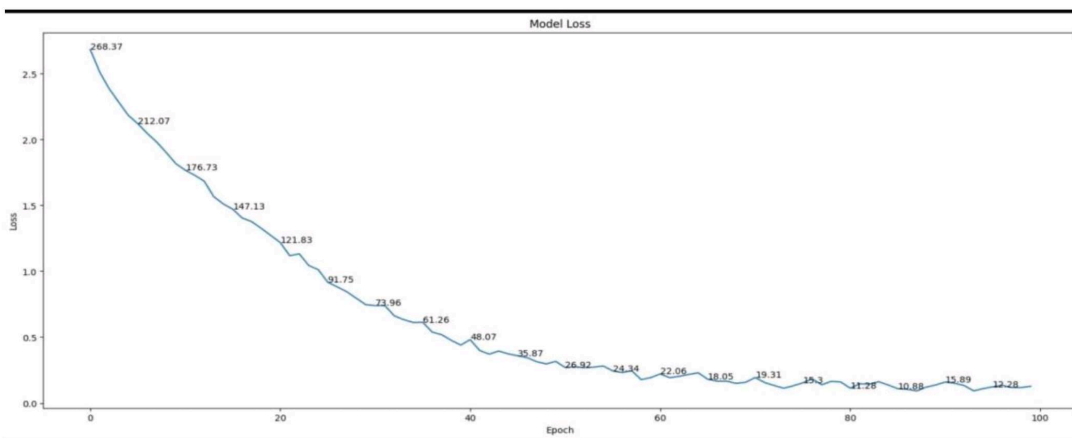


Fig 6.4 Loss Graph of Xception Net

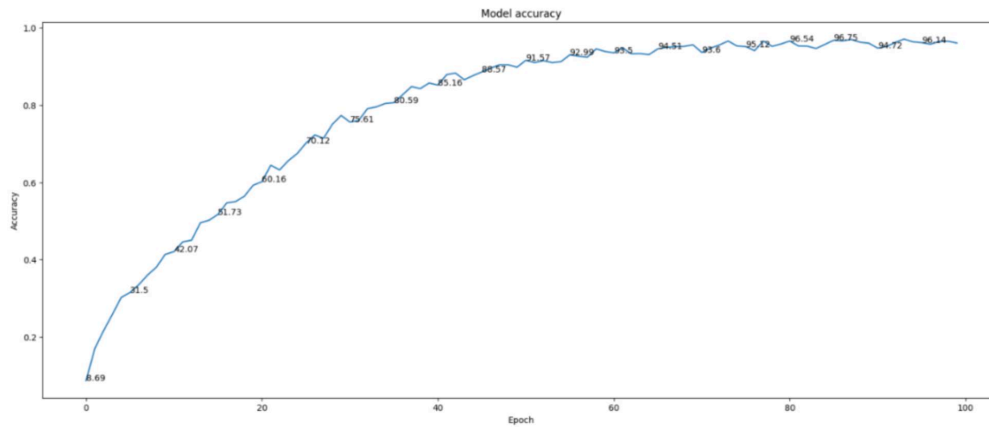


Fig 6.5 Accuracy Graph of Lenet

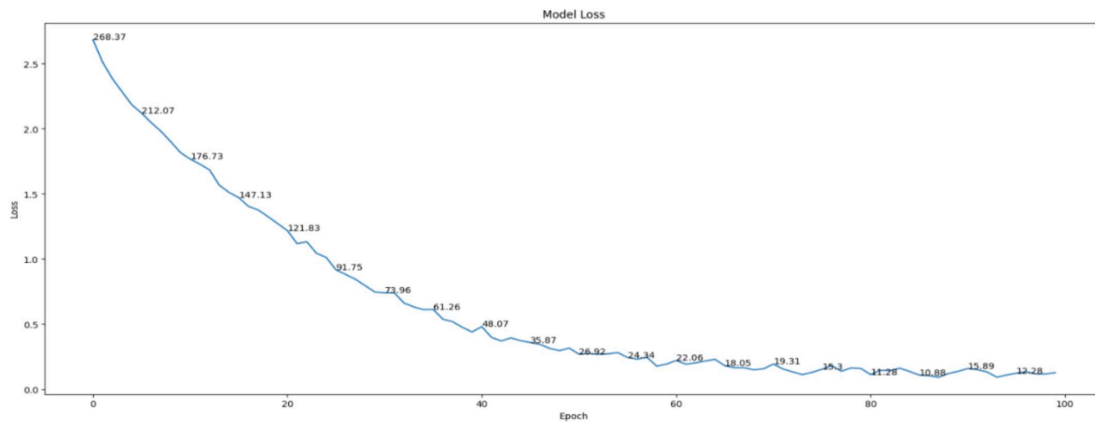


Fig 6.6 Loss Graph of Lenet

Fig 6.1 and 6.2 represent the validation accuracy and validation loss of Manual Net Architecture. In this module a total of 10 epochs are carried out using a batch size of 512.

Fig 6.3 and 6.4 represent the validation accuracy and validation loss of Xception Net Architecture. In this module a total of 100 epochs are carried out using a batch size of 512.

6.2 PERFORMANCE ANALYSIS:

Performance evaluation for Convolutional Neural Networks (CNNs) involves assessing how well the model is able to accomplish its intended tasks. Here are some common metrics and methods used for evaluating the performance of CNNs:

1. Accuracy: The ratio of correctly predicted instances to the total instances.
2. Precision: The ratio of true positive predictions to the total positive predictions.
3. Recall: The ratio of true positive predictions to the total positive predictions.
4. F1 Score: The harmonic mean of precision and recall.

Performance Metrics	Formula
Precision	$\frac{TP}{TP + FP}$
Recall	$\frac{TP}{TP + FN}$
F1-Score	$2 * \frac{precision * Recall}{precision + Recall}$
Average Accuracy (AA)	$\frac{TP + TN}{TP + TN + FP + FN}$
Overall Accuracy (OA)	$\frac{TP + TN}{(P + N)}$
Kappa (K)	$\frac{p_o - p_e}{1 - p_e}$

Table 6.2 Performance metrics

6.3 OUTPUT:

HUMAN

Human Activities classification using Deep Learning

All the activities performed by humans to meet the end goals are known as human activities. In general life, some human activities are executed to gain profit and some are done to achieve personal satisfaction. There are two types of human activities. Those are mentioned below: Economic Activities.

[Explore](#)



Fig 6.5 User Interface

The image fig. 6.5 is the opening page of the website where you click explore and that leads you to the login page. The user can enter their details to proceed further after login in the user need to upload the image as shown in the fig 6.6. After uploading the image the model runs and the result is displayed as shown in fig 6.7

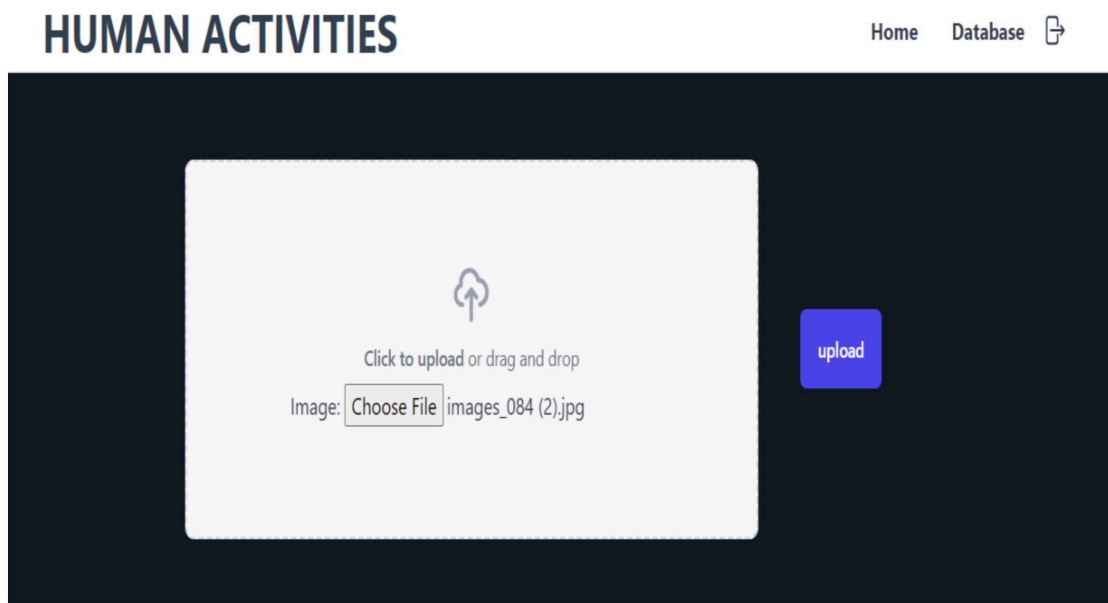


fig 6.6 Upload the Image

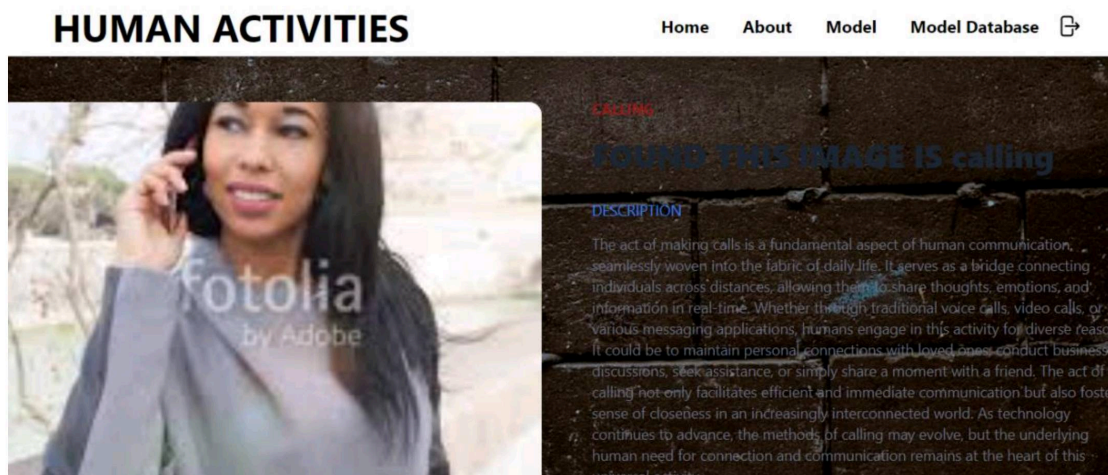


fig 6.7 Activity Detected

CHAPTER 7

SUMMARY AND CONCLUSION

7.1 SUMMARY:

The provided texts explore the realm of Human Activity Recognition (HAR) through Convolutional Neural Networks (CNNs) in various contexts. Firstly, a methodology employing CNNs is developed to effectively capture local dependencies and scale invariance within signals, particularly focusing on accelerometer data. Additionally, a novel technique called partial weight sharing is introduced to enhance CNN-based HAR models, showcasing improved accuracy compared to existing methods across three public datasets.

A literature review underscores the significance of HAR research, highlighting the utilization of different sensor technologies like Inertial Measurement Units (IMUs) and smart home sensors. CNNs emerge as a promising solution for HAR, leveraging raw sensor data for accurate activity classification.

The outlined project aims to leverage CNNs for real-time activity recognition, with objectives encompassing the development of robust models, optimization of architecture for efficient feature extraction, and exploration of model generalization across diverse scenarios.

The system architecture revolves around a deep learning approach, utilizing artificial neural networks with multiple hidden layers to automatically learn patterns from data. This architecture is adaptable to various applications, including image and speech recognition.

Detailed project requirements include functional aspects such as the use of specific libraries for software development and non-functional aspects involving a systematic workflow from problem definition to prediction. Feasibility studies entail dataset splitting, model construction, and design architecture, particularly focusing on data flow diagrams (DFDs) to illustrate information flow within the system.

The working process involves steps like installing Anaconda, loading datasets, evaluating algorithms, and making predictions using Python. Finally, environmental requirements specify software dependencies like Anaconda with Jupyter Notebook and hardware specifications like minimum processor and RAM.

In summary, the texts provide comprehensive insights into the development and application of CNN-based models for HAR, emphasizing methodology, literature review, project requirements, feasibility study, system architecture, and practical implementation steps.

7.2 CONCLUSION:

The provided texts explore the realm of Human Activity Recognition (HAR) through Convolutional Neural Networks (CNNs) in various contexts. Firstly, a methodology employing CNNs is developed to effectively capture local dependencies and scale invariance within signals, particularly focusing on accelerometer data. Additionally, a novel technique called partial weight sharing is introduced to enhance CNN-based HAR models, showcasing improved accuracy compared to existing methods across three public datasets.

A literature review underscores the significance of HAR research, highlighting the utilization of different sensor technologies like Inertial Measurement Units (IMUs) and smart home sensors. CNNs emerge as a promising solution for HAR, leveraging raw sensor data for accurate activity classification.

The outlined project aims to leverage CNNs for real-time activity recognition, with objectives encompassing the development of robust models, optimization of architecture for efficient feature extraction, and exploration of model generalization across diverse scenarios.

The system architecture revolves around a deep learning approach, utilizing artificial neural networks with multiple hidden layers to automatically learn patterns from data. This architecture is adaptable to various applications, including image and speech recognition.

Detailed project requirements include functional aspects such as the use of specific libraries for software development and non-functional aspects involving a systematic workflow from problem definition to prediction. Feasibility studies entail dataset splitting, model construction, and design architecture, particularly focusing on data flow diagrams (DFDs) to illustrate information flow within the system.

The working process involves steps like installing Anaconda, loading datasets, evaluating algorithms, and making predictions using Python. Finally, environmental requirements specify software dependencies like Anaconda with Jupyter Notebook and hardware specifications like minimum processor and RAM.

In summary, the texts provide comprehensive insights into the development and application of CNN-based models for HAR, emphasizing methodology, literature review, project requirements, feasibility study, system architecture, and practical implementation steps.

7.2.1 FUTURE WORK:

Future work could explore enhanced feature extraction methods and incorporate attention mechanisms to improve the accuracy and robustness of Convolutional Neural Networks for Human Activity Recognition.

REFERENCES

- [1] Gupta, A., Gupta, K., Gupta, K. and Gupta, K., 2020, July. A survey on human activity recognition and classification. In *2020 international conference on communication and signal processing (ICCSP)* (pp. 0915-0919). IEEE.
- [2] Gupta, A., Kembhavi, A. and Davis, L.S., 2009. Observing human-object interactions: Using spatial and functional compatibility for recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31(10), pp.1775-1789.
- [3] Sharma, G., Jurie, F. and Schmid, C., 2013. Expanded parts model for human attribute and action recognition in still images. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 652-659).
- [4] Sun, X., Xu, H., Dong, Z., Shi, L., Liu, Q., Li, J., Li, T., Fan, S. and Wang, Y., 2022. Capsganet: Deep neural network based on capsule and gru for human activity recognition. *IEEE Systems Journal*, 16(4), pp.5845-5855.

APPENDIX

A SOURCE CODE

MANUAL NET ARCHITECTURE

```
import warnings
```

```
warnings.filterwarnings('&#39;ignore&#39;)
```

```
import os
```

```
import glob
```

```
import numpy as np
```

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

```
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense
```

```
from PIL import Image
```

```
from tensorflow.keras.layers import Convolution2D
```

```
from tensorflow.keras.layers import MaxPooling2D
```

```
from tensorflow.keras.layers import Flatten
```

```
from tensorflow.keras.layers import Activation
```

```
from keras.callbacks import ModelCheckpoint
```

```
import matplotlib.pyplot as plt
```

```
call = '&#39;datasets/train/calling&#39;
```

```
cycle = '&#39;datasets/train/cycling&#39;
```

```
fight = '&#39;datasets/train/fighting&#39;
```

```

run = '#39;datasets/train/running#39;
sleep = '#39;datasets/train/sleeping#39;
def plot_images(item_dir, n=6):
    all_item_dir = os.listdir(item_dir)

    item_files = [os.path.join(item_dir, file) for file in
    all_item_dir][:n]

    plt.figure(figsize=(80, 40))
    for idx, img_path in enumerate(item_files):
        plt.subplot(3, n, idx+1)
        img = plt.imread(img_path)
        plt.imshow(img, cmap='#39;gray#39;')
        plt.axis('#39;off#39;')
    plt.tight_layout()

    def image_details_print(data,path):
        print('#39;===== Images in: #39;, path)
        for key,values in data.items():
            print(key,'#39;:\t#39;, values)

    def images_details(path):
        files=[f for f in glob.glob(path + '"**/*.*"', recursive=True)]
        data={}
        data['Images_count']=len(files)
        data['Min_width']=10**100

```

```

data[&#39;Max_width&#39;]=0

data[&#39;Min_height&#39;]=10**100

data[&#39;Max_height&#39;]=0

for f in files:

img=Image.open(f)

width,height=img.size

data[&#39;Min_width&#39;]=min(width,data[&#39;Min_width&#39;])

data[&#39;Max_width&#39;]=max(width, data[&#39;Max_width&#39;])

data[&#39;Min_height&#39;]=min(height, data[&#39;Min_height&#39;])

data[&#39;Max_height&#39;]=max(height, data[&#39;Max_height&#39;])

image_details_print(data,path)

print("&quot;&quot;")

print("&quot;TRAINING DATA FOR call:&quot;")

print("&quot;&quot;")

images_details(call)

print("&quot;&quot;")

plot_images(call, 10)

print("&quot;&quot;")

print("&quot;TRAINING DATA FOR cycle:&quot;")

print("&quot;&quot;")

images_details(cycle)

print("&quot;&quot;")

plot_images(cycle, 10)

print("&quot;&quot;")

```

```

print("&quot;TRAINING DATA FOR fight:&quot;")

print("&quot;&quot;")

images_details(fight)

print("&quot;&quot;")

plot_images(fight, 10)

print("&quot;&quot;")

print("&quot;TRAINING DATA FOR run:&quot;")

print("&quot;&quot;")


images_details(run)

print("&quot;&quot;")

plot_images(run, 10)

print("&quot;&quot;")

print("&quot;TRAINING DATA FOR sleep:&quot;")

print("&quot;&quot;")

images_details(sleep)

print("&quot;&quot;")

plot_images(sleep, 10)

train_datagen=ImageDataGenerator(rescale=1./255,shear_range=0.2,zoom_range=
0.2,horizontal_flip=True)

training_set=train_datagen.flow_from_directory('&#39;datasets/train&#39;;target_size
=(224,224),batch_size=32,class_mode=&#39;categorical&#39;')

test_datagen=ImageDataGenerator(rescale=1./255)

```

```

test_set=test_datagen.flow_from_directory('datasets/test',target_size=(22
4,
224),batch_size=32,class_mode='categorical')

Classifier=Sequential()

Classifier.add(Convolution2D(32,(3,3),input_shape=(224,224,3),activation='r
elu'))

Classifier.add(MaxPooling2D(pool_size=(2,2)))

Classifier.add(Flatten())

Classifier.add(Dense(38, activation='relu'))

Classifier.add(Dense(15, activation='softmax'))

Classifier.compile(optimizer='rmsprop',loss='categorical_crossentropy',metr
ics=['accuracy'])

model_path = 'MANUAL.h5'

callbacks = [

ModelCheckpoint(model_path, monitor='accuracy', verbose=1,
save_best_only=True)

]

epochs = 10

batch_size = 512

#### Fitting the model

history = Classifier.fit(
training_set, steps_per_epoch=training_set.samples //
batch_size,
epochs=epochs,

```

```

validation_data=test_set,validation_steps=test_set.samples //

batch_size,

callbacks=callbacks)

import matplotlib.pyplot as plt

def graph():

    #Plot training & validation accuracy values

    plt.plot(history.history['accuracy'])

    plt.plot(history.history['val_accuracy'])

    plt.title('Model accuracy')

    plt.ylabel('Accuracy')

    plt.xlabel('Epoch')

    plt.legend(['Train', 'Test'], loc='upper left')

    plt.show()

graph()

import matplotlib.pyplot as plt

def graph():

    plt.plot(history.history['loss'])

    plt.plot(history.history['val_loss'])

    plt.title('Model loss')

    plt.ylabel('Loss')

    plt.xlabel('Epoch')

    plt.legend(['Train', 'Test'], loc='upper left')

    plt.show()

```



```
## XCEPTION NET ARCHITECTURE
```

```
import warnings
```

```
warnings.filterwarnings('#39;ignore#39;)
```

```
import tensorflow
```

```
import tensorflow as tf
```

```
print(tf.__version__)
```

```
import keras
```

```
import keras.backend as K
```

```
from keras.models import Model
```

```
from keras.layers import Input, Dense, Conv2D, Conv3D, DepthwiseConv2D,
```

```
SeparableConv2D, Conv3DTranspose
```

```
from keras.layers import Flatten, MaxPool2D, AvgPool2D, GlobalAvgPool2D,
```

```
UpSampling2D, BatchNormalization
```

```
from keras.layers import Concatenate, Add, Dropout, ReLU, Lambda,
```

```
Activation, LeakyReLU, PReLU
```

```
from IPython.display import SVG
```

```
from keras.utils import model_to_dot
```

```
from time import time
```

```
import numpy as np
```

```

from keras.callbacks import ModelCheckpoint

from tensorflow.keras.callbacks import EarlyStopping


import warnings

warnings.filterwarnings('&#39;ignore&#39;;)

from tensorflow.keras.preprocessing.image import ImageDataGenerator


train=ImageDataGenerator(rescale=1./255,shear_range=0.2,zoom_range=0.2,ho
rizontal_flip=True,validation_split = 0.2)


train_data=train.flow_from_directory(directory =
&#39;datasets/train&#39;,target_size=(224,224),
batch_size=32,class_mode=&#39;categorical&#39;,)
test=ImageDataGenerator(rescale=1./255)
test_data=test.flow_from_directory(directory =
&#39;datasets/test&#39;,target_size=(224,224),
batch_size=32,class_mode=&#39;categorical&#39;,)
def xception(input_shape, n_classes):

def conv_bn(x, f, k, s=1, p=&#39;same&#39;):
x = Conv2D(f, k, strides=s, padding=p, use_bias=False)(x)
x = BatchNormalization()(x)

return x

```

```

def sep_bn(x, f, k, s=1, p='same'):
    x = SeparableConv2D(f, k, strides=s, padding=p, use_bias=False)(x)
    x = BatchNormalization()(x)
    return x

```

```

def entry_flow(x):
    x = conv_bn(x, 32, 3, 2)
    x = ReLU()(x)
    x = conv_bn(x, 64, 3)
    tensor = ReLU()(x)

```

```

x = sep_bn(tensor, 128, 3)
x = ReLU()(x)
x = sep_bn(x, 128, 3)
x = MaxPool2D(3, strides=2, padding='same')(x)

```

```

tensor = conv_bn(tensor, 128, 1, 2)

```

```

x = Add()([tensor, x])
x = ReLU()(x)
x = sep_bn(x, 256, 3)

```

```

x = ReLU()(x)
x = sep_bn(x, 256, 3)

```

```
x = MaxPool2D(3, strides=2, padding='same')(x)
```

```
tensor = conv_bn(tensor, 256, 1, 2)
```

```
x = Add()(tensor, x)
```

```
x = ReLU()(x)
```

```
x = sep_bn(x, 728, 3)
```

```
x = ReLU()(x)
```

```
x = sep_bn(x, 728, 3)
```

```
x = MaxPool2D(3, strides=2, padding='same')(x)
```

```
tensor = conv_bn(tensor, 728, 1, 2)
```

```
x = Add()(tensor, x)
```

```
return x
```

```
def middle_flow(tensor):
```

```
    for _ in range(8):
```

```
        x = ReLU()(tensor)
```

```
        x = sep_bn(x, 728, 3)
```

```
        x = ReLU()(x)
```

```
        x = sep_bn(x, 728, 3)
```

```
        x = ReLU()(x)
```

```
        x = sep_bn(x, 728, 3)
```

```
tensor = Add()([tensor, x])
```

```
return tensor
```

```
def exit_flow(tensor):
```

```
    x = ReLU()(tensor)
```

```
    x = sep_bn(x, 728, 3)
```

```
    x = ReLU()(x)
```

```
    x = sep_bn(x, 1024, 3)
```

```
    x = MaxPool2D(3, strides=2, padding='same')(x)
```

```
    tensor = conv_bn(tensor, 1024, 1, 2)
```

```
    x = Add()([tensor, x])
```

```
    x = sep_bn(x, 1536, 3)
```

```
    x = ReLU()(x)
```

```
    x = sep_bn(x, 2048, 3)
```

```
    x = ReLU()(x)
```

```
    x = GlobalAvgPool2D()(x)
```

```
    x = Dense(n_classes, activation='softmax')(x)
```

```
return x
```

```
input = Input(input_shape)
```

```
x = entry_flow(input)
```

```
x = middle_flow(x)
```

```
output = exit_flow(x)
```

```
model = Model(input, output)
```

```
model.compile(optimizer='Adam',loss='categorical_crossentropy',  
metrics=['acc
```

```
uracy',tensorflow.keras.metrics.Precision()])
```

```
return model
```

```
input_shape = 224, 224, 3
```

```
n_classes = 15
```

```
K.clear_session()
```

```
model = xception(input_shape, n_classes)
```

```
model.summary()
```

```
model_path = 'XCEPTION.h5'
```

```
from keras.callbacks import ModelCheckpoint
```

```
M = ModelCheckpoint(model_path, monitor='accuracy', verbose=1,
```

```

save_best_only=True)

epochs = 100

batch_size = 512

#### Fitting the model

history = model.fit(
train_data, steps_per_epoch=train_data.samples // batch_size,
epochs=epochs,
validation_data=test_data, validation_steps=test_data.samples //
batch_size,
callbacks=[M])

history.history.keys()

import matplotlib.pyplot as plt

import numpy as np

plt.figure(figsize=(20, 8))

plt.plot(history.history['accuracy'])

for i in range(epochs):
    if i%5 == 0:
        plt.annotate(np.round(history.history['accuracy'][i]*100,2),xy=(i,history.history[
'accuracy'][i]))

plt.title('Model accuracy')

```

```

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.show()

plt.figure(figsize=(20, 8))

plt.plot(history.history['loss'])

for i in range(epochs):

    if i%5 == 0:

        plt.annotate(np.round(history.history['loss'][i]*100,2),xy=(i,history.history[
            'loss']

            [i]))

plt.title('Model Loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.show()

# LENET ARCHITECTURE

import warnings

warnings.filterwarnings('ignore')

# import os

# os.environ['KMP_DUPLICATE_LIB_OK']='TRUE'

import tensorflow

```



```

import tensorflow as tf

print(tf.__version__)


from tensorflow.keras.models import Sequential


from tensorflow.keras.layers import Conv2D

from tensorflow.keras.layers import Convolution2D

from tensorflow.keras.layers import MaxPool2D

from tensorflow.keras.layers import MaxPooling2D

from tensorflow.keras.layers import Flatten

from tensorflow.keras.layers import Dense

from tensorflow.keras.layers import Dropout

from tensorflow.keras.layers import Activation

from tensorflow.keras.preprocessing.image import ImageDataGenerator


train=ImageDataGenerator(rescale=1./255,shear_range=0.2,zoom_range=0.2,ho
rizontal_flip=True,validation_split = 0.2)

train_data=train.flow_from_directory(directory =
'&#39;datasets/train&#39;,target_size=(224,224),
batch_size=32,class_mode='&#39;categorical&#39;.)

test=ImageDataGenerator(rescale=1./255)

test_data=test.flow_from_directory(directory =
'&#39;datasets/test&#39;,target_size=(224,224),
batch_size=32,class_mode='&#39;categorical&#39;.)

```

```

MODEL=Sequential()

MODEL.add(Convolution2D(filters=32, kernel_size=(3,3), strides=(3,3),
input_shape=(224,224,3), padding=('#39;valid#39;'), activation=('#39;relu#39;'))

MODEL.add(MaxPooling2D(pool_size=(2,2), strides=(2,2),
padding=('#39;valid#39;'))

MODEL.add(Convolution2D(filters=128, kernel_size=(3,3), strides=(3,3),
padding=('#39;valid#39;'), activation=('#39;relu#39;'))

MODEL.add(MaxPooling2D(pool_size=(2,2), strides=(2,2),
padding=('#39;valid#39;'))

MODEL.add(Flatten())

MODEL.add(Dense(256, activation=('#39;relu#39;'))

MODEL.add(Dense(15, activation=('#39;softmax#39;'))


OPT = tensorflow.keras.optimizers.Adam(0.001)


MODEL.compile(optimizer=OPT,loss=('#39;categorical_crossentropy#39;',metrics=
['&quot;acc
uracy&quot;', tensorflow.keras.metrics.Precision(), tensorflow.keras.metrics.Recall()])

MODEL.summary()

model_path = '&quot;LENET.h5&quot;'


from tensorflow.keras.callbacks import ModelCheckpoint

M = ModelCheckpoint(model_path, monitor=('#39;accuracy#39;', verbose=1,

```

```

save_best_only=True, mode='max')

epochs = 100

batch_size = 32

WORKING = MODEL.fit_generator(
    train_data, steps_per_epoch=train_data.samples // batch_size,
    epochs=epochs,
    validation_data=test_data, validation_steps=test_data.samples //
    batch_size,
    callbacks=[M])

WORKING.history.keys()

import matplotlib.pyplot as plt

import numpy as np

plt.figure(figsize=(20, 8))

plt.plot(WORKING.history['accuracy'])

for i in range(epochs):
    if i%5 == 0:
        plt.annotate(np.round(WORKING.history['accuracy'][i]*100,2),xy=(i,WORK
IN
G.history['accuracy'][i]))

plt.title('Model accuracy')

```

```

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.show()

import matplotlib.pyplot as plt

plt.figure(figsize=(20, 8))

plt.plot(WORKING.history['loss'])

for i in range(epochs):

    if i%5 == 0:

        plt.annotate(np.round(WORKING.history['loss'][i]*100,2),xy=(i,WOR
hi
king.history['loss'][i]))

plt.title('Model Loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.show()

```