SRH University Heidelberg

Human Activity Classification using Inertial Measurement Data Device

Author:	Nikhil Kumar
Matriculation Number:	11037894
Email Address:	nikhil.kumar@stud.hochschule-heidelberg.de
Date of Submission:	26 Sep 2024

Under the Guidance of Professor

Dr. Ing Milan Gnjatovic

Aim of the Task:

This project aims to develop a machine-learning model capable of classifying five different cooking activities (brownies, pizza, sandwiches, salad, scrambled eggs) based on sensor data from an Inertial Measurement Unit (IMU) in the CMU-MMAC dataset. This dataset captures motion and orientation data from 38 subjects using a specific IMU sensor (ID 2796) across all subjects. The goal is to accurately predict the prepared recipe by analysing time-series data collected during the cooking process using deep learning techniques such as Recurrent Neural Networks (RNN). The performance of the model will be evaluated using metrics like accuracy, precision, recall, and F1score. The final objective is to achieve classification accuracy on the test dataset and provide comprehensive documentation of the entire process, including insights into the model's effectiveness and any challenges encountered.

Objective :

The primary objective of this project is to develop a machine learning-based classification system that can accurately identify five distinct cooking activities (brownies, pizza, sandwich, salad, scrambled eggs) from Inertial Measurement Unit (IMU) sensor data. By analysing time-series data collected from a single IMU sensor (ID 2796) across 38 subjects, this project aims to extract meaningful features and train a model capable of distinguishing between the different recipes based on motion patterns. The system will leverage advanced techniques such as Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU) and Bidirectional LSTM RNN (BiLSTM) networks to achieve this goal, providing insights into the effectiveness of IMU sensor data for real-time activity classification.

The particular goals are:

- **Data Collection and Preprocessing:**

Collect and preprocess sensor data from the Inertial Measurement Unit (IMU) (ID 2796) across 38 subjects performing five cooking activities (brownies, pizza, sandwich, salad, scrambled eggs). Ensure the data is cleaned, standardized, and structured for time-series analysis.

- **Feature Extraction and Labelling:**

Extract relevant features from the IMU data, such as accelerometer and gyroscope readings, and accurately label the data with corresponding recipe categories.

- **Model Development:**

Design and implement machine learning models, focusing on Recurrent Neural Networks (RNN) , to capture temporal patterns in the sensor data for recipe classification.

- **Model Training and Evaluation:**

Train the models using 80% of the dataset and evaluate their performance on the remaining 20% using metrics such as accuracy, precision, recall, and F1 score to determine classification effectiveness.

- **Optimize Model Performance:**

Tune model hyperparameters and experiment with different architectures to improve classification accuracy, ensuring the model can generalize well to unseen data.

- **Report Findings and Insights:**

Analyze the results, documenting the model's performance, challenges faced, and potential areas for improvement. Provide insights into how IMU sensor data can be used effectively for activity recognition and classification tasks.

Introduction :

In recent years, the use of Inertial Measurement Units (IMUs) has become increasingly prominent in activity recognition tasks, particularly in fields such as health monitoring, sports performance analysis, and human activity classification. IMUs are compact sensors that measure acceleration, angular velocity, and sometimes magnetic fields to capture motion and orientation data. In this project, an IMU sensor (ID 2796) is employed to collect time-series data from subjects performing various cooking activities. The data from the IMU sensor provides valuable insights into the movements and postures involved in cooking, making it an excellent candidate for machine learning-based activity recognition.

The dataset used in this project comes from the Carnegie Mellon University Multimodal Activity Database (CMU-MMAC), which is designed to capture a wide range of human activities in a kitchen environment. The dataset includes sensor data from 38 subjects performing five different recipes: brownies, pizza, sandwich, salad, and scrambled eggs. For consistency and to reduce variability, this project focuses on the data collected from a single IMU sensor (ID 2796) for each subject across all activities. The dataset is structured into folders, with each subject's data segmented by recipe, and includes accelerometer, gyroscope, and magnetometer readings that form the basis for time-series analysis and classification.

To analyze and classify the cooking activities, this project leverages Recurrent Neural Networks (RNN, which are well-suited for time-series data. RNNs are designed to handle sequential data by maintaining a memory of previous inputs, making them ideal for detecting patterns in continuous sensor readings. LSTM networks, an advanced form of RNNs, are capable of learning long-term dependencies in data, which is crucial for accurately recognizing activities from IMU sensor data that evolves over time. By training and evaluating these models, the project aims to classify cooking activities with high accuracy, offering insights into how deep learning can enhance activity recognition using IMU data.

Inertial measurement unit

An inertial measurement unit (IMU) is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the orientation of the body, using a combination of accelerometers, gyroscopes, and sometimes magnetometers. When the magnetometer is included, IMUs are referred to as IMMUs.[1a]

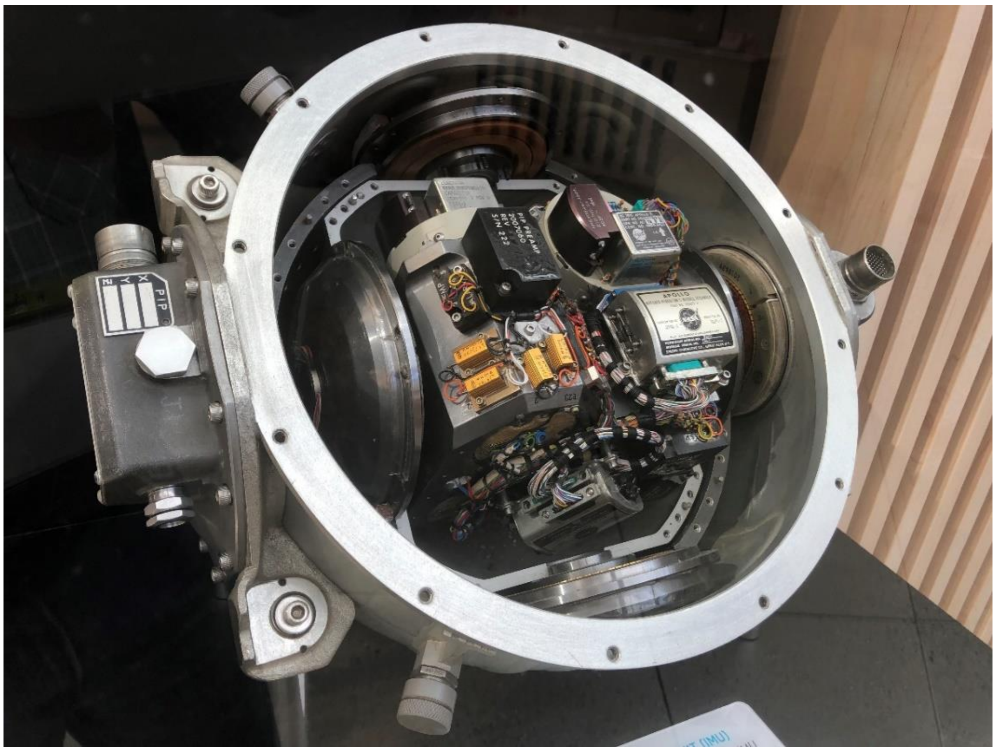


Fig 1: Inertial Measurement Unit [2a]

IMUs are typically used to maneuver modern vehicles including motorcycles, missiles, aircraft (an attitude and heading reference system), including uncrewed aerial vehicles (UAVs), among many others, and spacecraft, including satellites and landers. Recent developments allow for the production of IMU-enabled GPS devices. An IMU allows a GPS receiver to work when GPS signals are unavailable, such as in tunnels, inside buildings, or when electronic interference is present.[1a]

IMUs are used in VR headsets and smartphones, and also in motion-tracked game controllers like the Wii Remote. [1a]

Working of IMU:

An Inertial Measurement Unit (IMU) is a sophisticated sensor that tracks the motion and orientation of an object in space by measuring various physical quantities. The inertial measurement unit works by detecting linear acceleration using one or more accelerometers and rotational rate using one or more gyroscopes. Some also include a magnetometer which is commonly used as a heading reference.[1a]

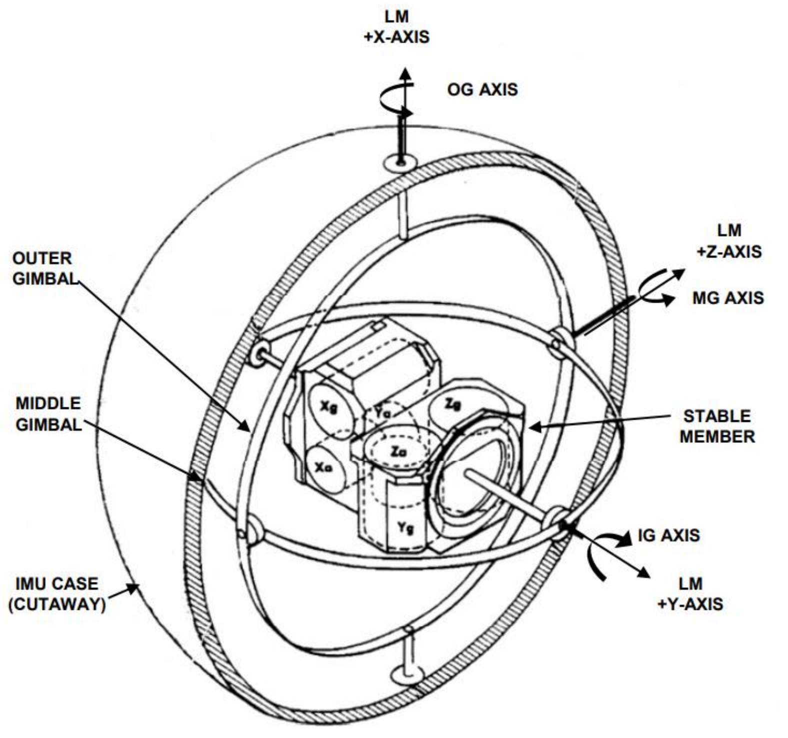


Fig 2: Components of Inertial Measurement Unit [2b]

The IMU typically consists of three main components: an accelerometer, a gyroscope, and sometimes a magnetometer.

- **Accelerometer**:

  The accelerometer measures linear acceleration along the X, Y, and Z axes. This allows the IMU to detect changes in speed and direction. In a cooking scenario, the accelerometer captures movements like stirring, chopping, or mixing by recording the force applied in different directions. For example, when a subject moves their arm to chop vegetables, the accelerometer captures the change in velocity and direction during each chopping motion.

- **Gyroscope**:

  The gyroscope measures angular velocity, i.e., how quickly an object rotates around the X, Y, and Z axes. This is useful for detecting orientation changes, such as wrist rotations during stirring or flipping ingredients. The gyroscope helps the IMU understand rotational movements, which are crucial for detecting subtle actions like flipping a sandwich or whisking ingredients in a bowl.

- **Magnetometer** (if present):

  Some IMUs include a magnetometer, which measures the magnetic field around the device to determine orientation relative to the Earth's magnetic field. It acts as a compass, helping to correct drift in orientation measurements over time. While not always necessary for every application, magnetometers can provide additional orientation data, especially in environments where precision in movement detection is critical.

For instance One accelerometer, gyro, and magnetometer are included in the typical setup, and each of the three main axes operates on the principles of pitch, roll, and yaw.

An aeroplane is the finest illustration of this, and it is discussed below.

- **Pitch**: Refers to the up-and-down rotation around the side-to-side axis (Y-axis). In an airplane, this would be the nose tilting up or down.
- **Roll**: Describes the side-to-side tilting motion around the front-to-back axis (X-axis). In an airplane, this is when one wing dips lower than the other.
- **Yaw**: Refers to the left or right rotation around the vertical axis (Z-axis). In an airplane, yaw is the nose turning left or right. [1a]

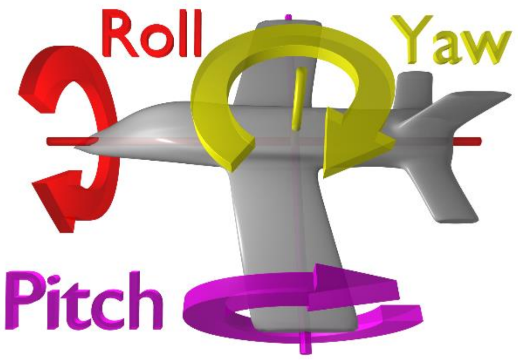


Fig 3: Image of Aeroplane explaining Roll, Pitch, and Yaw [2c]

These principles are measured by the IMU's gyroscope and are crucial for understanding the orientation and rotation of any object, including human motions during cooking tasks.

The IMU works by continuously sampling data from these sensors at a high frequency, generating time-series data that reflects the subject's movements. This data is typically recorded as a sequence of measurements in a three-dimensional space, providing detailed information about how the subject's body moves during a particular task. In the case of the CMU-MMAC dataset, the IMU data collected from subjects as they perform various cooking tasks helps capture the specific motions involved in each recipe.

By combining data from the accelerometer, gyroscope, and possibly the magnetometer, the IMU can create a comprehensive picture of an object's movement and orientation in space. This data can then be fed into machine learning algorithms to recognize and classify different activities based on the patterns and dynamics of motion detected by the IMU.

Use cases of Inertial Measurement Unit:

Inertial Measurement Units (IMUs) are a crucial component of Inertial Navigation Systems (INS), which use raw IMU data to calculate attitude, angular rates, linear velocity, and position relative to a global reference frame. IMU-equipped INS are integral to navigation and control in many commercial and military vehicles, such as aircraft, missiles, ships, submarines, and satellites. These systems also play a key role in uncrewed vehicles like UAVs, UGVs, and UUVs. Simpler INS versions, known as Attitude and Heading Reference Systems (AHRS), use IMUs to calculate vehicle orientation and heading relative to magnetic north, using dead reckoning to track the craft's position.[1a]

Inland vehicles and IMUs are often integrated with GPS-based automotive navigation systems, providing dead reckoning capabilities. This integration enhances data accuracy related to speed, turn rate, heading, and acceleration. Combined with wheel speed sensors and other vehicle data, IMUs improve traffic collision analysis and overall navigation. The data from IMUs is typically presented as Euler angles or quaternions, which represent the rotation on the three primary axes. [1a]

IMUs also serve as orientation sensors in consumer products like smartphones, tablets, and fitness trackers, measuring activities like running and general motion. In gaming, devices like the Nintendo Wii use IMUs to detect movement. Low-cost IMUs have facilitated the growth of the consumer drone industry and are commonly used in sports training, animation, and motion capture technology. IMUs are also central to balancing technologies, such as those used in the Segway Personal Transporter. [1a]

Assembly of Inertial Measurement Unit:

High-performance IMUs, or IMUs designed to operate under harsh conditions, are very often suspended by shock absorbers. These shock absorbers are required to master three effects:

- reduce sensor errors due to mechanical environment solicitations
- protect sensors as they can be damaged by shocks or vibrations
- contain parasitic IMU movement within a limited bandwidth, where processing will be able to compensate for them. [1a]

Suspended IMUs can offer very high performance, even when submitted to harsh environments. However, to reach such performance, it is necessary to compensate for three main resulting behaviors:

- coning - a parasitic effect induced by two orthogonal rotations
- sculling - a parasitic effect induced by an acceleration orthogonal to a rotation
- centrifugal accelerations effects. [1a]

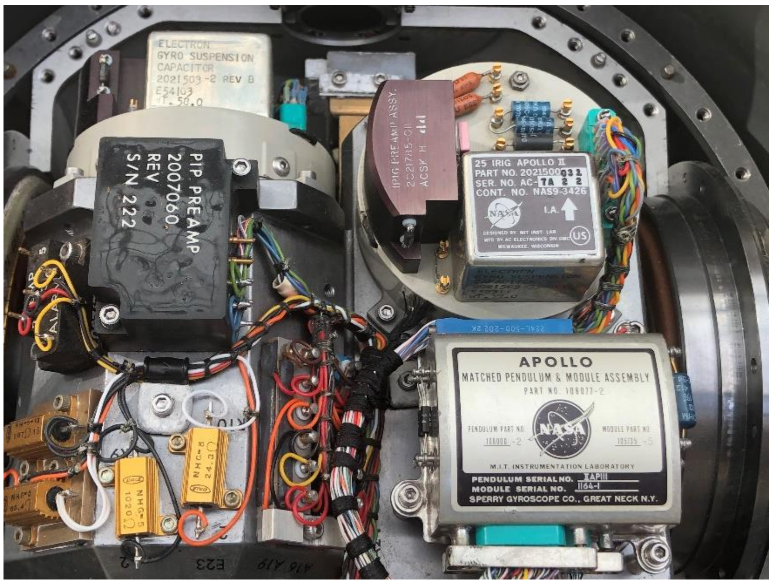

ELECTRON GYRO SUSPENSION CAPACITOR
PIP PREAMP 2007060 REV S/N 222
25 IRIG APOLLO II
APOLLO MATCHED PENDULUM & MODULE ASSEMBLY
SPERRY GYROSCOPE CO., GREAT NECK N.Y.

Fig 4: Assembly of Inertial Measurement Unit [2d]

Decreasing these errors tends to push IMU designers to increase processing frequencies, which becomes easier using recent digital technologies. However, developing algorithms able to cancel these errors requires deep inertial knowledge and strong intimacy with sensors/IMU design. On the other hand, if suspension is likely to enable IMU performance to increase, it has a side effect on size and mass. [1a]

A wireless IMU is known as a **WIMU.**

What is Recipe Classification?

Recipe classification is the process of identifying and categorizing different cooking activities based on data captured during the preparation of specific recipes. The goal is to use data-driven approaches, particularly machine learning models, to analyze patterns of movement, ingredient usage, or other relevant factors to determine which recipe a person is preparing. In the context of sensor data, recipe classification involves recognizing the distinct physical actions involved in preparing various dishes, such as stirring, chopping, mixing, or frying, and mapping these actions to a specific recipe.

For example, in a dataset collected from sensors like accelerometers or gyroscopes, the movement data captured during the preparation of a recipe is analyzed to understand the unique signature of that activity. Each recipe typically involves a set of repetitive and characteristic movements (like kneading dough for pizza or chopping vegetables for a salad), which can be used to train machine learning models to recognize and classify different recipes based on these movement patterns.

In this project, the classification task involves using data from an Inertial Measurement Unit (IMU) sensor, which captures motion and orientation during the preparation of five different recipes. The aim is to develop a system that can classify which recipe is being prepared by analyzing time-series data from the sensor. By leveraging techniques such as Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) networks, the classification system learns to recognize the cooking activities associated with each recipe based on the subject's movements.

What is the CMU-MMAC Dataset?

The Carnegie Mellon University Multimodal Activity Database (CMU-MMAC) contains multimodal measures of the human activity of subjects performing the tasks involved in cooking and food preparation. The CMU-MMAC database was collected in Carnegie Mellon's Motion Capture Lab. A kitchen was built and to date twenty-five subjects have been recorded cooking five different recipes: brownies, pizza, sandwich, salad, and scrambled eggs.The CMU-MMAC aims to provide researchers with data that reflects real-world human behavior while interacting with kitchen tools and appliances, making it a valuable resource for activity recognition, motion analysis, and multimodal learning research. [1b]

The CMU-MMAC dataset contains recordings from 16 different cooking tasks performed by 44 subjects, with sensor data collected from various body parts using accelerometers, gyroscopes, and magnetometers. Each subject is equipped with multiple sensors, and their activities are segmented into different phases of the cooking process. The dataset offers detailed information about the subject's body movement and orientation while performing tasks like stirring, chopping, mixing, and other actions essential to preparing specific recipes.[1b]

In this project, a subset of the CMU-MMAC dataset is utilized, specifically focusing on data from a single IMU sensor (ID 2796) placed on the subject's body. This dataset captures the motion data of 38 subjects preparing five specific recipes: brownies, pizza, sandwich, salad, and scrambled eggs. By using this subset, the project aims to classify the recipes based on the motion patterns identified in the sensor data, exploring how different cooking activities can be recognized and distinguished through machine learning techniques.

Recipe Classification Using the CMU-MMAC Dataset:

The CMU Multimodal Activity Database (CMU-MMAC) captures human activities in a kitchen environment using various data modalities, including motion sensors, video, and audio. For recipe classification, this project focuses on sensor data from an Inertial Measurement Unit (IMU) (ID 2796), which records motion and orientation data as subjects prepare five recipes: brownies, pizza, sandwich, salads, and scrambled eggs.

The IMU collects time-series data using accelerometers, gyroscopes, and magnetometers, capturing the movements involved in each recipe. Machine learning models can then analyze these patterns to classify the recipe being prepared based on motion dynamics. This process is critical for activity recognition tasks where understanding subtle human movements is key.

Recurrent Neural Networks (RNN), specifically Long Short-Term Memory (LSTM) models, are employed to process the sequential IMU data. These models are ideal for capturing temporal dependencies, enabling accurate classification of cooking actions. The classification workflow includes data preprocessing, feature extraction, model training, and performance evaluation to predict the correct recipe based on the recorded movements.

Fig 5: A Glimpse of collecting data from IMU sensor in the kitchen (CMU-MMAC) [2e]

Neural Networks

A neural network is a software solution that leverages machine learning (ML) algorithms to 'mimic' the operations of a human brain. Neural networks process data more efficiently and feature improved pattern recognition and problem-solving capabilities when compared to traditional computers. Neural networks are also known as artificial neural networks (ANNs) or simulated neural networks (SNNs). [1c]

Neural networks are a subtype of machine learning and an essential element of deep learning algorithms. Just like its functionality, the architecture of a neural network is also based on the human brain. Its highly interlinked structure allows it to imitate the signaling processes of biological neurons.[1c]

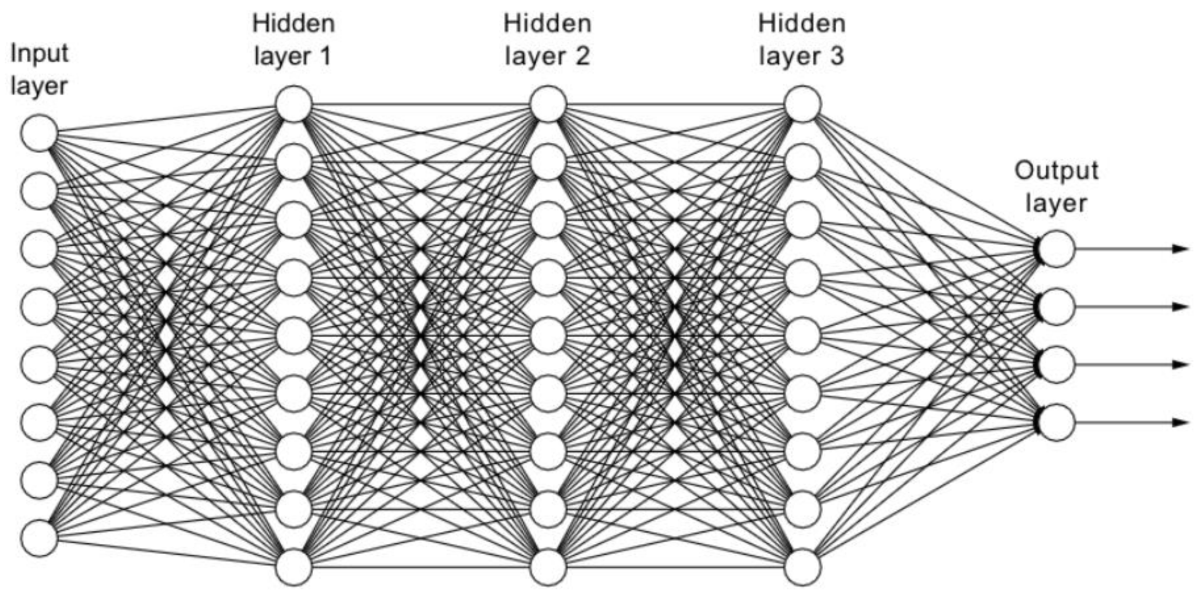


Fig 6: The Architecture of a Neural Network [2f]

The architecture of a neural network comprises node layers that are distributed across an input layer, single or multiple hidden layers, and an output layer. Nodes are '**artificial neurons**' linked to each other and are associated with a particular weight and threshold. Once the output of a single node crosses its specified threshold, that particular node is activated, and its data is transmitted to the next layer in the network. If the threshold value of the node is not crossed, data is not transferred to the next network layer. [1c]

Unlike traditional computers, which process data sequentially, neural networks can learn and multitask. In other words, while conventional computers only follow the instructions of their programming, neural networks continuously evolve through advanced algorithms. It can be said that neural computers 'program themselves' to derive solutions to previously unseen problems. [1c]

Additionally, traditional computers operate using logic functions based on a specific set of calculations and rules. Conversely, neural computers can process logic functions and raw inputs such as images, videos, and voice. [1c]

While traditional computers are ready to go out of the box, neural networks must be **‘trained’** over time to increase their accuracy and efficiency. Fine-tuning these learning machines for accuracy pays rich dividends, giving users a powerful computing tool in artificial intelligence (AI) and computer science applications. [1c]

Neural networks are capable of classifying and clustering data at high speeds. This means, among other things, that they can complete the recognition of speech and images within minutes instead of the hours that it would take when carried out by human experts. The most commonly used neural network today is Google search algorithms. [1c]

How Does a Neural Network Work?

The ability of a neural network to **‘think’** has revolutionized computing as we know it. These smart solutions are capable of interpreting data and accounting for context.

Four **critical steps** that neural networks take to operate effectively are:

- **Associating** or training enables neural networks to ‘remember’ patterns. If the computer is shown an unfamiliar pattern, it will associate the pattern with the closest match present in its memory. [1c]
- **Classification** or organizing data or patterns into predefined classes.
- **Clustering** or the identification of a unique aspect of each data instance to classify it even without any other context present.
- **Prediction,** or the production of expected results using a relevant input, even when all context is not provided upfront. [1c]

What are neural networks used for?

Neural networks have several use cases across many industries, such as the following:

- Medical diagnosis by medical image classification
- Targeted marketing by social network filtering and behavioral data analysis
- Financial predictions by processing historical data of financial instruments
- Electrical load and energy demand forecasting
- Process and quality control
- Chemical compound identification [1d]

We give four of the important applications of neural networks below.

Computer vision

Computer vision is the ability of computers to extract information and insights from images and videos. With neural networks, computers can distinguish and recognize images similar to humans. Computer vision has several applications, such as the following:

- Visual recognition in self-driving cars so they can recognize road signs and other road users
- Content moderation to automatically remove unsafe or inappropriate content from image and video archives
- Facial recognition to identify faces and recognize attributes like open eyes, glasses, and facial hair
- Image labeling to identify brand logos, clothing, safety gear, and other image details[1d]

Speech recognition

Neural networks can analyze human speech despite varying speech patterns, pitch, tone, language, and accent. Virtual assistants like Amazon Alexa and automatic transcription software use speech recognition to do tasks like these:

- Assist call center agents and automatically classify calls
- Convert clinical conversations into documentation in real time
- Accurately subtitle videos and meeting recordings for wider content reach[1d]

Natural language processing

Natural language processing (NLP) is the ability to process natural, human-created text. Neural networks help computers gather insights and meaning from text data and documents. NLP has several use cases, including in these functions: [1d]

- Automated virtual agents and chatbots
- Automatic organization and classification of written data
- Business intelligence analysis of long-form documents like emails and forms
- Indexing of key phrases that indicate sentiment, like positive and negative comments on social media
- Document summarization and article generation for a given topic[1d]

The Biological Inspiration for Neural Networks:

The concept of ANNs draws inspiration from how neurons in the human brain process information. Neurons communicate through electrical impulses, and ANNs replicate this idea by passing signals between artificial neurons. These neurons are organized into layers: an input layer, one or more hidden layers, and an output layer.[1e]

Recurrent Neural Networks (RNNs)

Recurrent neural networks (RNNs) are the state-of-the-art algorithm for sequential data and are used by Apple's Siri and Google's voice search. It is the first algorithm that remembers its input, due to an internal memory, which makes it perfectly suited for machine learning problems that involve sequential data. It is one of the algorithms behind the scenes of the amazing achievements seen in deep learning over the past few years. [1f]

Recurrent Neural Networks (RNNs) were introduced to address the limitations of traditional neural networks, such as FeedForward Neural Networks (FNNs), when it comes to processing sequential data. FNN takes inputs and process each input independently through a number of hidden layers without considering the order and context of other inputs. Due to which it is unable to handle sequential data effectively and capture the dependencies between inputs. As a result, FNNs are not well-suited for sequential processing tasks such as, language modeling, machine translation, speech recognition, time series analysis, and many other applications that requires sequential processing. To address the limitations posed by traditional neural networks, RNN comes into the picture.[1g]

Because of their internal memory, RNNs can remember important things about the input they received, which allows them to be very precise in predicting what's coming next. This is why they're the preferred algorithm for sequential data like time series, speech, text, financial data, audio, video, weather and much more. Recurrent neural networks can form a much deeper understanding of a sequence and its context compared to other algorithms. [1f]

Think of it like a "memory-keeping" calculator that not only processes the current data but also remembers what it has seen before.

Architecture of Recurrent Neural Network

A basic Recurrent Neural Network (RNN) is a type of neural network designed for sequence data. Unlike traditional feedforward neural networks, RNNs have connections that form directed cycles, allowing them to maintain a hidden state that captures information about previous inputs in the sequence. This makes RNNs well-suited for tasks involving sequential or time-series data.[1h]

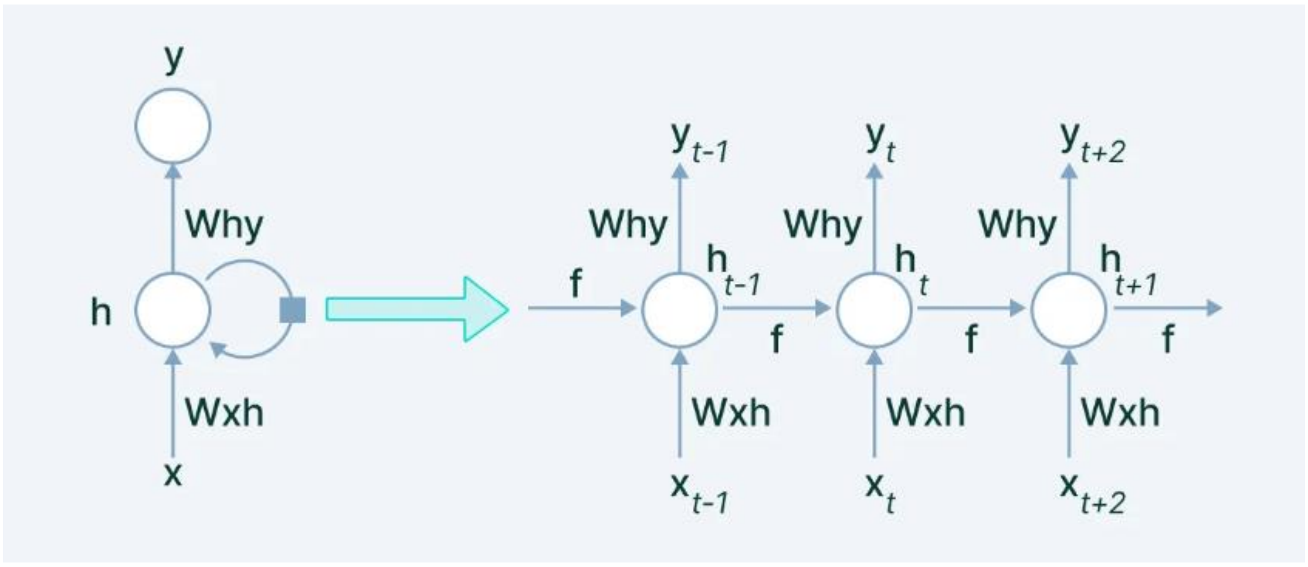


Fig 7: The Architecture of a Recurrent Neural Network [2g].

The architecture of a basic RNN consists of the following components:

1. **Input Layer:**

This is where the input data is fed into the model. Each input is processed one step at a time, like feeding words from a sentence or steps from a time series.

2. **Recurrent Connection:**

The RNN has a special connection that helps it "remember" information from previous steps. At each step, it combines what it learned from the past with the new input to make better predictions.

3. **Hidden State:**

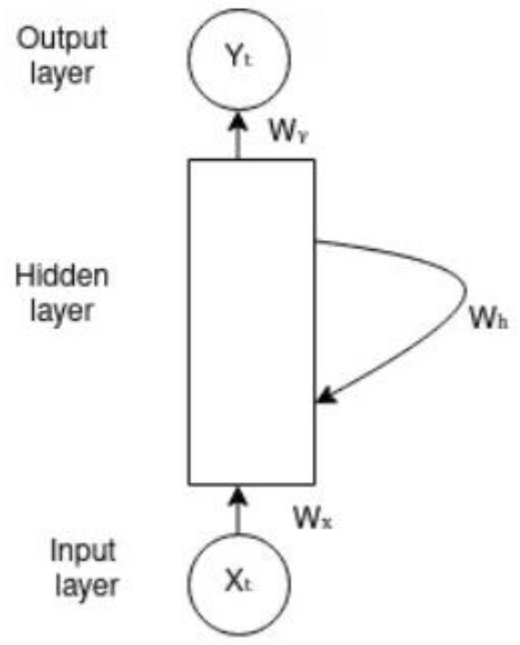
This is the memory of the RNN. It stores what the model has learned from the previous inputs and gets updated with each new input.

4. **Output Layer:**

The output layer produces the output for the current time step based on the current input and the hidden state. [1h]

RNN overcomes these limitations by introducing a recurrent connection that allows information to flow from one time-step to the next. This recurrent connection enables RNNs to maintain internal memory, where the output of each step is fed back as an input to the next step, allowing the network to capture the information from previous steps and utilize it in the current step, enabling the model to learn temporal dependencies and handle input of variable length.[1g]

Image visualization describing the single layer of RNN is shown below –



How Do Recurrent Neural Networks Work?

To understand RNNs properly, you'll need a working knowledge of "normal" feed-forward neural networks and sequential data.

Sequential data is basically just ordered data in which related things follow each other. Examples are financial data or the DNA sequence. The most popular type of sequential data is perhaps time series data, which is just a series of data points that are listed in time order.

RNNs and feed-forward neural networks get their names from the way they channel information.

In a feed-forward neural network, the information only moves in one direction — from the input layer, through the hidden layers, to the output layer. The information moves straight through the network.[1f]

Feed-forward neural networks have no memory of the input they receive and are bad at predicting what's coming next. Because a feed-forward network only considers the current input, it has no notion of order in time. It simply can't remember anything about what happened in the past except its training. [1f]

In an RNN, the information cycles through a loop. When it makes a decision, it considers the current input and also what it has learned from the inputs it received previously. [1f]

The two images below illustrate the difference in information flow between an RNN and a feed-forward neural network.

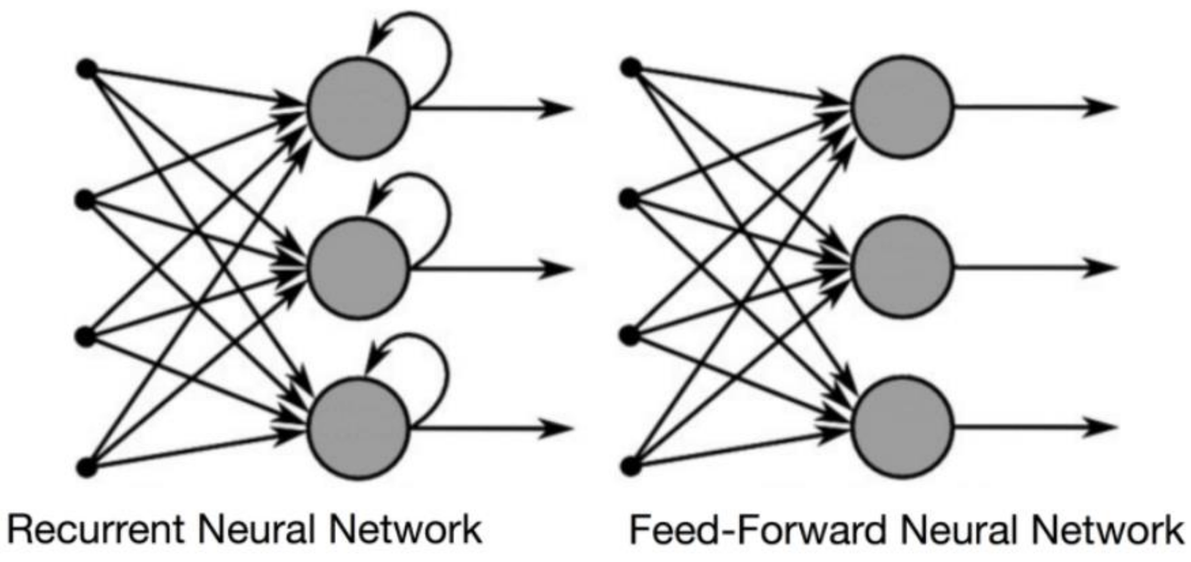


Fig 8: Explaining the working of RNN with comparison of Feed Forward network [2h].

A usual RNN has a short-term memory. In combination with an LSTM, they also have a long-term memory (more on that has been later in different variants of RNN) [1f]

Another good way to illustrate the concept of a recurrent neural network's memory is to explain it with an example: Imagine you have a normal feed-forward neural network and give it the word "neuron" as an input and it processes the word character by character. By the time it reaches the character "r," it has already forgotten about "n," "e" and "u," which makes it almost impossible for this type of neural network to predict which character would come next.

A recurrent neural network, however, is able to remember those characters because of its internal memory. It produces output, copies that output and loops it back into the network. [1f]

Simply put: Recurrent neural networks add the immediate past to the present.

Therefore, an RNN has two inputs: the present and the recent past. This is important because the sequence of data contains crucial information about what is coming next, which is why an RNN can do things other algorithms can't. [1f]

An example of how an RNN functions in relation to a human –

Imagine you're reading a book. Each word you read gives you some information, and you keep a mental note of what you have read so far to understand the story better.

An RNN works similarly:

- **Current Input:** It looks at the current piece of data (like the word you're reading right now).
- **Memory (Hidden State):** It remembers what it has seen before (like the part of the story you've read so far).
- **Prediction:** It uses both the current input and what it remembers to make a decision (like guessing what might happen next in the story).

When It's said *"The hidden layer uses both the current input and its previous memory,"* What does it refer to?

It refers to the memory stored in the **hidden state** of the **same neuron** from the previous time step. Here's a more detailed explanation:

What is "Previous Memory"?

- **Previous Memory** refers to the hidden state (ht-1) of the same neuron at the previous time step t-1.

How It Works:

1. **Same Neuron, Different Time Steps**:
    - In an RNN, each neuron in the hidden layer is associated with a hidden state that updates at every time step.
    - At time step T, the neuron's hidden state ht is calculated using:
        - The current input xt at time step T.
        - The hidden state from the previous time step ht-1, which is the "previous memory."
    - This means the neuron doesn't just rely on the current input, but also remembers what it computed in the previous time step, allowing it to maintain context over time.

2. **Memory of the Same Neuron**:
    - Each neuron in the hidden layer has its own hidden state, which gets updated as the sequence progresses.
    - The "previous memory" refers to the hidden state of the **same neuron** from the last time step, not the state of a different neuron.
    - This recurrent connection allows the neuron to consider both the current input and what it has learned from all previous inputs.

Example:

- Imagine you're reading a book one word at a time.
  - At each word (time step T), you think about the current word Xt and also remember what you read just before (this memory is like ht-1).
  - As you move to the next word (time step t+1), you combine the new word with your memory from the previous step to understand the sentence better.

Visualization:

- **Time Step t−1**:
  - Neuron processes input xt-1 and updates its hidden state to ht-1.
- **Time Step t**:
  - The same neuron now processes the new input xt and combines it with the hidden state ht-1 (its "previous memory") to produce a new hidden state ht.

Why Choose Recurrent Neural Networks (RNNs) over other Neural Networks?

Recurrent Neural Networks (RNNs) are specifically designed to handle sequential data, making them ideal for tasks where the order of inputs is crucial. Unlike feedforward neural networks, RNNs have connections that allow information to persist, meaning they can maintain context by using previous inputs to influence current processing. This is essential in time-series data, natural language processing, and tasks like recipe classification, where the temporal sequence of movements or events is important.

RNNs are particularly advantageous when there are dependencies between different time steps in the data. For example, in recipe classification, the movements performed at one point in time are influenced by actions performed earlier. RNNs can capture these dependencies more effectively than other networks that process data in isolation.

While there are many types of neural networks, RNNs are preferred for problems where patterns unfold over time. Their ability to handle variable-length input sequences and retain information across those sequences makes them the go-to choice for tasks involving sequential or temporal data, like speech recognition, activity recognition, and sensor-based classification.

Where are Recurrent Neural Networks used?

Text Prediction: Predicting the next word in a sentence (like when your phone suggests the next word while typing).

Speech Recognition: Converting spoken words into text.

Time Series Prediction: Predicting future events based on past data, like forecasting stock prices or weather.

Simple Example -

If you train an RNN on the phrase "hello world," it learns that after "hel", the next letter is likely "l". So, if you give it "hel", it will predict "l" as the next letter.

Types Of Recurrent Neural Network (RNN)

There are four types of RNNs based on the number of inputs and outputs in the network.

1. One to One
2. One to Many
3. Many to One
4. Many to Many

One to One:

This type of RNN behaves the same as any simple Neural network it is also known as Vanilla Neural Network. In this Neural network, there is only one input and one output. [1i]

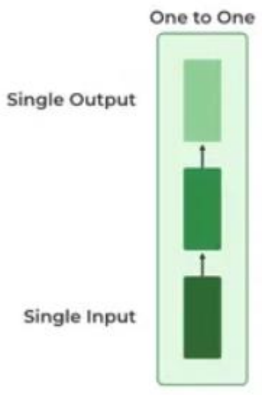


Fig 9: One to One RNN [2i].

One To Many:

In this type of RNN, there is one input and many outputs associated with it. One of the most used examples of this network is Image captioning where given an image we predict a sentence having Multiple words. [1i]

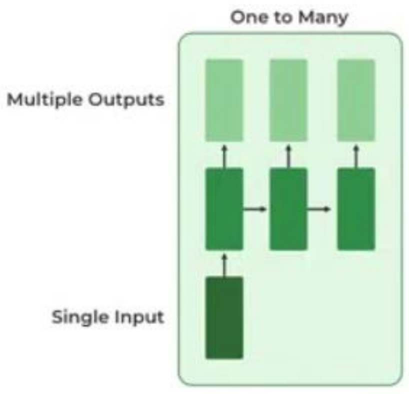


Fig 10: One to Many RNN [2j].

Many to One:

In this type of network, Many inputs are fed to the network at several states of the network generating only one output. This type of network is used in the problems like sentimental analysis. Where we give multiple words as input and predict only the sentiment of the sentence as output. [1i]

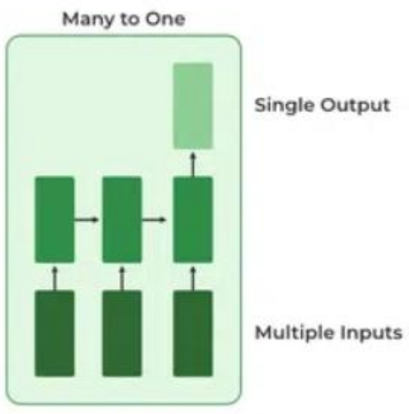


Fig 11:Many to One RNN [2k].

Many to Many:

In this type of neural network, there are multiple inputs and multiple outputs corresponding to a problem. One example of this problem will be language translation. In language translation, we provide multiple words from one language as input and predict multiple words from the second language as output. [1i]

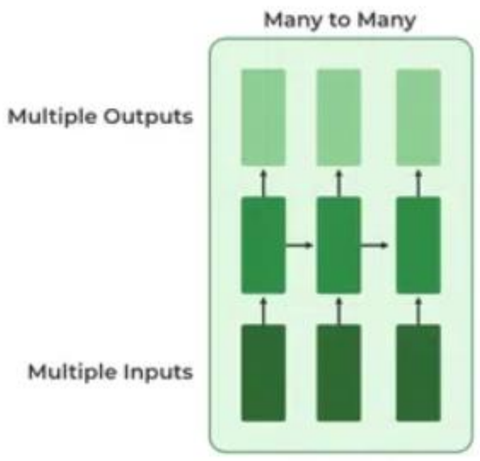


Fig 12: Many to Many RNN [2l].

What are some variants of Recurrent Neural Network architecture?

The RNN architecture laid the foundation for ML models to have language processing capabilities. Several variants have emerged that share its memory retention principle and improve on its original functionality**.**

1. Simple RNN (Recurrent Neural Network) :

Simple RNN is the basic form of a neural network that can handle sequences of data, like time series or sentences. Unlike regular neural networks, which process inputs independently, a Simple RNN has a "memory" that allows it to remember information from previous steps in a sequence.

In a Simple RNN, at each time step:

1. An input is provided, which is processed to update the hidden state.
2. The hidden state contains information about previous inputs.
3. The network produces an output based on the current input and the hidden state.

Here’s how it works:

- Imagine you’re trying to understand a story where each sentence depends on the previous one. Simple RNN works by passing the information from each step (or word) along to the next step. This is done by using something called a "hidden state" that remembers information from previous inputs.

- At each step, the Simple RNN takes the current input and combines it with the hidden state from the previous step. It uses this to make a prediction or decision, and then passes the updated hidden state to the next step in the sequence.

However, **Simple RNNs have a limitation:** they struggle to remember things from far back in a sequence (a problem known as "vanishing gradients"). So, if the sequence is long, they might forget important information from earlier steps. That’s why more advanced models like LSTM and GRU were developed to fix this issue.

Simple RNN is a basic neural network that can remember some previous information, making it useful for sequential tasks. **But it’s not great at handling long sequences because it tends to forget earlier information.**

2. Long Short-Term Memory Networks (LSTM)

Long Short-Term Memory (LSTM) networks are an advanced variant of RNNs, specifically designed to overcome the limitations of traditional RNNs, which struggle to retain information over long sequences. While standard RNNs can only remember short-term dependencies, LSTMs can store and retrieve information over longer periods, making them ideal for time-series data or sequential tasks that require memory of earlier inputs. LSTMs achieve this through special memory blocks, or "cells," in the hidden layers. These cells are controlled by three gates: the input gate (which decides what new information is stored), the forget gate (which determines what information is discarded), and the output gate (which regulates what part of the cell's stored information is used for the current output). This structure enables LSTMs to retain useful long-term information while discarding irrelevant data, allowing them to make more accurate predictions. For example, in natural language processing, LSTMs can retain context from earlier parts of a sentence to correctly predict a word several words later.

It's like a smart memory system that can decide what information to keep and what to forget. Here's how it works:

- Imagine you are reading a long book. To understand the story, you need to remember key events from earlier chapters, but you also don't need to remember every single detail. LSTM works similarly. It tries to remember important information from the past (like key events) and ignore unimportant stuff.

- The LSTM has special parts called **gates**:
    1. **Input gate**: This decides what new information should be added to memory.
    2. **Forget gate**: This chooses what old information should be forgotten.
    3. **Output gate**: This controls what information is passed forward to help make decisions.

These gates help the LSTM focus on important details and keep useful information around for as long as needed, while getting rid of unnecessary data.
Because of this, LSTMs are very good at tasks where you need to understand sequences over time, like predicting the next word in a sentence, understanding videos, or forecasting stock prices.
In short, LSTM is a smart tool that learns to remember important things over time and helps solve problems that involve sequences or time-based data.

3. Bidirectional Long Short-Term Memory (BiLSTM)

A Bidirectional LSTM (BiLSTM) combines the bidirectional structure of a BRNN with the memory capabilities of an LSTM. Like BRNNs, BiLSTMs process data in both forward and backward directions. However, instead of using standard RNN layers, BiLSTMs use LSTM layers in both directions.

This means that a BiLSTM:

- Processes the sequence from past to future with one LSTM layer.
- Simultaneously processes the sequence from future to past with another LSTM layer.
- Combines the outputs from both directions, either by concatenation or summation.

Here's how it works:

- In a regular **LSTM**, the network processes the data in one direction, usually from the beginning of the sequence to the end. So, it only remembers what happened before the current step.

- In a **BiLSTM**, the data is processed in both directions. There are two LSTM layers:
  - **Forward LSTM layer**: This goes through the sequence from start to finish, just like a regular LSTM.
  - **Backward LSTM layer**: This goes through the sequence from the end to the beginning.

- These two layers run at the same time and then combine their results. This way, the BiLSTM has a fuller picture of the sequence because it has seen both what came before and what came after the current step.

For example, if you're trying to understand a sentence, knowing both the words before and after a word helps in understanding the context better. This is why BiLSTM is especially useful in tasks like **speech recognition, language translation, and time-series analysis**, where future context can be as important as past context.

BiLSTM enhances the LSTM by processing sequences in both directions—giving it the ability to look at both past and future data—making it more accurate in tasks where full context is important

4. Gated Recurrent Units (GRU)

Gated Recurrent Units (GRUs) are another type of RNN that simplifies the structure of LSTMs while retaining similar functionality. GRUs combine the hidden state and memory into a single mechanism, controlled by two gates: the update gate and the reset gate. The update gate controls how much of the previous information should be retained, while the reset gate determines how much of the past information should be forgotten. This streamlined architecture makes GRUs computationally less expensive than LSTMs, as they have fewer parameters. GRUs are often chosen for tasks where computational efficiency is important but still require the network to learn long-term dependencies, such as speech recognition or text generation.

To make it easier to understand , explained below in simpler terms:

Just like LSTM, GRU also has gates, but it only has two:

- **Update gate:** This decides how much of the past information should be remembered and how much new information should be added. It combines the role of both the input and forget gates in LSTM.

- **Reset gate:** This decides how much of the past information should be ignored or "reset" when processing new data.

GRU doesn't have as many parts as LSTM, so it's easier for the model to learn and run faster. However, it still performs well in tasks where you need to remember things over time, like understanding text, speech, or sequences of events.

In short, GRU is a simpler, faster version of LSTM that's good for remembering important information over time, but it uses fewer resources to do so. It's often used when you want a quicker solution without losing too much memory capability.

Training through RNN

1. A single-time step of the input is provided to the network.
2. Then calculate its current state using a set of current input and the previous state.
3. The current ht becomes ht-1 for the next time step.
4. One can go as many time steps according to the problem and join the information from all the previous states. [1i]
5. Once all the time steps are completed the final current state is used to calculate the output.
6. The output is then compared to the actual output i.e the target output and the error is generated.
7. The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained using Backpropagation through time.[1i]

Advantages and Disadvantages of Recurrent Neural Network

Advantages

1. An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
2. Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.[1i]

Disadvantages

1. Gradient vanishing and exploding problems.
2. Training an RNN is a very difficult task.
3. It cannot process very long sequences if using **tanh or relu** as an activation function.[1i]

Applications of Recurrent Neural Network

1. Language Modelling and Generating Text
2. Speech Recognition
3. Machine Translation
4. Image Recognition, Face detection
5. Time series Forecasting [1i]

Important Hyperparameters of Recurrent Neural Networks:

1. **Padding**: In the context of RNNs (Recurrent Neural Networks), **padding** refers to adding extra values (usually zeros) to sequences to make them all the same length. This is necessary because RNNs usually expect input sequences of the same length, but in real-world tasks like language processing or time-series data, sequences can vary in length.
   For example:

   If you have two sequences of lengths 5 and 3, you pad the second one with zeros to make it length 5.
   Sequence 1: [2, 5, 8, 3, 1]
   Sequence 2: [4, 7, 6] → [4, 7, 6, 0, 0] (after padding)

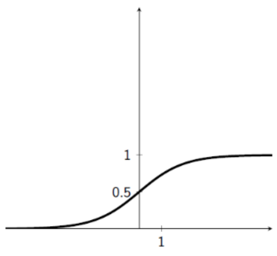
   Padding is crucial for handling variable-length inputs without losing information, as RNNs will still focus on the actual data while ignoring the padded parts.

2. **Activation Function:** An Activation Function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction.[1j]

   There are several commonly used activation functions such as -

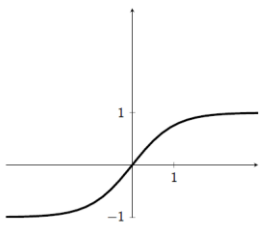
- **The sigmoid function -** For a binary classification in the model.
  The sigmoid function is a mathematical function defined as shown in figure which maps any real-valued number to a value between 0 and 1, making it useful for binary classification tasks in neural networks.
  It has an S-shaped curve, making it a smooth, non-linear function that can help model complex relationships. However, it suffers from the vanishing gradient problem, where gradients become very small for large positive or negative inputs, slowing down the training of deep networks. [1j]

$$g(z) = \frac{1}{1 + e^{-z}}$$



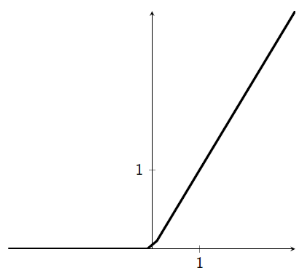
- **The hyperbolic tangent (tanh) function –**
  the tanh function, which is similar to the sigmoid function, is applied more often as an activation function (e.g., in recurrent neural networks). [1j]
  The only difference is that it is symmetric around the origin.
  The range of values, in this case, is from -1 to 1.[1j]

$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



- **The Rectified Linear Unit (ReLU) function -** The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.[1j] It is widely used in neural networks because it helps mitigate the vanishing gradient problem, allowing models to learn faster and perform better. However, ReLU can suffer from the "dying ReLU" problem, where neurons can become inactive and only output zero, especially when learning rates are high.[1j]

$$g(z) = \max(0, z) = \begin{cases} z, & \text{if } z > 0, \\ 0, & \text{otherwise} \end{cases}$$



- **The Softplus function -** The Softplus function is similar to ReLU but significantly less applied since it requires a more complex calculation.[1j]

- **The Softmax function -** It is used in multinomial logistic regression and is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

  ReLU of tanh are often used as activation functions for hidden layers and softmax for the output layer.

3. **Optimizer:** An **optimizer** in neural networks is an algorithm used to adjust the model’s weights and biases to reduce the error or loss during training. The optimizer helps the model learn from data by updating these parameters to minimize the difference between predicted and actual values. Common optimizers include **Gradient Descent**, **Adam**, and **RMSprop**, each of which updates the model’s parameters in different ways.

Gradient Descent:

Gradient Descent is one of the most popular methods for optimizing neural networks. The goal is to minimize the **loss function** (a measure of how far the model's predictions are from the actual values). Here’s how it works in simpler terms:

1. **Start with some initial weights**: The model begins with random values for its parameters (weights and biases).
2. **Calculate the loss**: After making predictions, the model computes the error (loss) by comparing the predicted values with the actual results.
3. **Find the gradient**: The optimizer computes the **gradient**, which is the direction and rate of the steepest ascent of the loss function. The gradient is determined by calculating the **partial derivatives** of the loss function with respect to each model parameter (weight or bias). These partial derivatives show how much a small change in each parameter will affect the loss.
4. **Update the parameters**: Since the goal is to minimize the loss, the optimizer adjusts the parameters in the opposite direction of the gradient (going downhill on the loss curve).

5. **Repeat**: The process is repeated for many iterations until the loss is minimized, and the model predictions become more accurate.

The **learning rate** controls how large the steps are when updating the parameters. If the learning rate is too high, the model might skip over the optimal solution, and if it's too low, the model might take too long to find the best solution or get stuck.

Variants of Gradient Descent:

- **Stochastic Gradient Descent (SGD)**: Instead of using the entire dataset to compute the gradient, SGD updates the parameters after processing each individual data point. This speeds up the training process but can introduce more noise in the updates.
- **Mini-batch Gradient Descent**: This method strikes a balance between standard gradient descent and SGD by using small batches of data to update the parameters. It’s faster than standard gradient descent and more stable than SGD.

Advanced Optimizers:

- **Adam (Adaptive Moment Estimation)**: Adam is an advanced optimizer that combines the advantages of **Momentum** (which helps smooth out updates by considering past gradients) and **RMSprop** (which adapts the learning rate for each parameter). It’s particularly useful for complex models and datasets, as it helps the model converge faster and improves accuracy.
- **RMSprop**: This optimizer adjusts the learning rate for each parameter individually by dividing the gradient by a moving average of recent gradients. This helps the model avoid large updates in areas where the loss changes rapidly.
- **Adagrad**: Adagrad adapts the learning rate based on the frequency of updates. It works well for sparse data but can make the learning rate shrink too much over time, slowing down training.

These optimizers ensure that neural networks learn effectively by adjusting how they navigate the parameter space, making them crucial for achieving better performance and faster convergence.

4. Loss Functions:

A loss function in neural networks quantifies the difference between the predicted output and the actual target values. It provides a measure of how well or poorly the model is performing, guiding the optimization process. Common loss functions include Mean Squared Error (MSE) for regression tasks and Cross-Entropy Loss for classification tasks.

For Regression

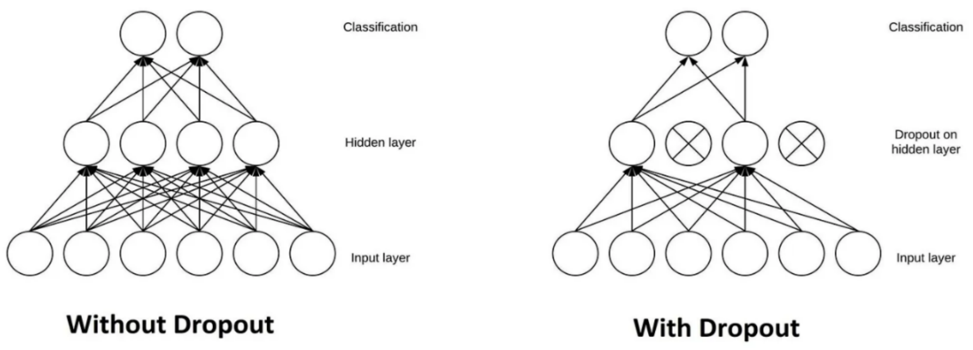
- **Mean Squared Error (MSE):**
  - ❖ Measures the average of the squares of the errors, that is, the average squared difference between the estimated values and the actual value.
  - ❖ This is one of the simplest and most effective cost or loss functions that we can use. It can also be called the **quadratic cost function or the sum of squared errors**.

- **Mean Absolute Error (MAE):**
    - ❖ Measures the average of the absolute differences between predicted values and actual values.
- **Huber Loss:**
    - ❖ Combines the best properties of MAE and MSE by being less sensitive to outliers than MSE and more sensitive to small errors than MAE.

For Classification

- **Binary Cross entropy:**
    - ❖ Used for binary classification problems.
    - ❖ Measures the performance of a classification model whose output is a probability value between 0 and 1.
- **Categorical Cross entropy:**
    - ❖ Used for multi-class classification problems.
    - ❖ Measures the performance of a classification model whose output is a probability distribution across multiple categories.
- **Sparse Categorical Cross entropy:**
    - ❖ Similar to categorical cross-entropy but for integer labels instead of one-hot encoded labels.

5. **Dropout:** Another typical characteristic of RNNs is a Dropout layer. The Dropout layer is a mask that nullifies the contribution of some neurons toward the next layer and leaves unmodified all others.[1j]



6. **Epochs:** Epochs in neural networks refer to the number of complete passes(Iterations) through the entire training dataset during the training process. Each epoch allows the model to learn and adjust its weights based on the training data. Multiple epochs can improve model accuracy by iteratively refining the weights.

7. **Verbose:** It is a parameter that controls the level of logging and output during the training process. Setting verbose to 0 means no output, 1 provides progress bars, and 2 gives more detailed logging for each epoch, helping users monitor the training process.

8. **Batch Size:** Batch size is the number of training samples processed before the model updates its internal parameters. Smaller batch sizes offer more frequent updates and can lead to a more stable training process. Larger batch sizes make better use of hardware acceleration and can improve training speed but might require more memory.

Evaluation Measures:

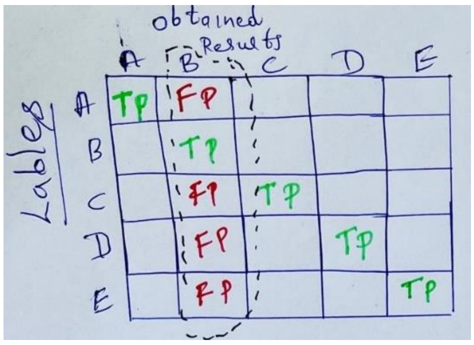
Evaluation measures for the Recurrent Neural Network on the CMU-MMAC dataset are important for assessing the performance of the classifier in a multi-class classification task, such as recognizing recipes. Here are the evaluation measures commonly used for RNN on the CMU-MMAC dataset:

1. **Confusion Matrix:** A much better way to evaluate the performance of a classifier is to look at the confusion matrix. The confusion matrix is a tabular representation that summarizes the performance of a classification model by presenting the counts of true positive, true negative, false positive, and false negative predictions for each class. The general idea is to count the number of times instances of class A are classified as class B. Each row in a confusion matrix represents an actual class, while each column represents a predicted class. For example, to know the number of times the classifier confused images of 5s with 3s, you would look in the 5th row and 3rd column of the confusion matrix. A perfect classifier would have only true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal (top left to bottom right) [1l]

2. **Accuracy:** Accuracy measures the proportion of correctly classified samples out of the total number of samples. It is calculated as:
   Accuracy = Number of correct predictions/ Total Number of predictions.

3. **Precession:** You can find a lot of information in the confusion matrix, but occasionally you might want a shorter metric. An interesting one to look at is the accuracy of the positive predictions; this is called the precision of the classifier. [1l]
   Precision measures the ability of the classifier to correctly identify positive samples (true positives) out of all samples predicted as positive. It is calculated as:

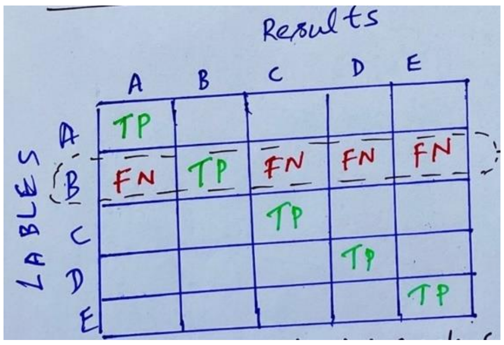
$$P = \frac{\text{true positive}}{\text{true positive} + \text{false positive}} .$$



Precision is useful when the cost of false positives is high.

4. **Recall:** Recall measures the ability of the classifier to correctly identify positive samples (true positives) out of all actual positive samples. It is calculated as:

$$R = \frac{\text{true positive}}{\text{true positive} + \text{false negative}} .$$



5. **F1 Score**: It is often convenient to combine precision and recall into a single metric called the F1 score, in particular, if you need a simple way to compare two classifiers. The F1 score is the harmonic mean of precision and recall. Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values. As a result, the classifier will only get a high F1 score if both recall and precision are high.[1l]
This Harmonic mean provides a balance between the two metrics.
It is calculated as :

$$F_1 = \frac{2}{P^{-1} + R^{-1}} = \frac{2PR}{P+R} \; . \qquad [1k]$$

6. **Macro-average Metrics:** In multi-class classification tasks like MNIST, macro-average metrics compute the average of individual metrics (precision, recall, F1-score) calculated for each class. This approach treats all classes equally and is useful for assessing overall classifier performance across multiple categories.

Understanding the Output View (with a random example)

1. **Epoch Progress:**
    - Epoch 1/30 indicates the first epoch out of a total of 10 epochs.
    - loss: Training loss after the epoch.
    - accuracy: Training accuracy after the epoch.

2. **Training Metrics:**
30/30 [==============================] - 64s 33ms/step indicates the number of batches processed (30 batches), the total time taken for the epoch (64s), and the time per step (33ms/step).

3. **Loss and Accuracy:**
loss: 0.0071 - Accuracy: 0.4531 shows the loss and accuracy for the training data after the first epoch.

4. **Validation Metrics:**
- val_loss: 0.0028 - val_accuracy: 0.3833 shows the loss and accuracy for the validation data after the first epoch.
- val_loss: Validation loss after the epoch.
- val_accuracy: Validation accuracy after the epoch

Outlining the Functionality of the Program (What's happening within the code)

The functionality of the program is explained below, following
Having completed numerous trials, I have experimented with various hyperparameters and functions. Selected one of the programs and explained it here.
(Program File: RNN_LSTM_Logic2_Trial1.ipynb)

1. Loading and Preprocessing the Data

➢ **load_data() Function**:

- **Goal**: Load sensor data from files and prepare it for the machine learning model.

- **Base Path**: The base directory (base_path) is where the dataset is stored.

Two empty lists are created:

  - `data`: This will store the sensor data for each recipe as a sequence of numeric values (likely accelerometer and other sensor readings).
  - `labels`: This will store the corresponding label for each data sequence, which is the name of the recipe (e.g., "Brownie", "Pizza").
    These lists will eventually hold the entire dataset in memory: each sequence in `data` corresponds to a recipe label in `labels`.

- **Recipes**: List of five recipes to classify: Brownie, Eggs, Pizza, Salad, and Sandwich. This is a predefined list of the five recipes that you are interested in classifying: "Brownie", "Eggs", "Pizza", "Salad", and "Sandwich".

- **Loop through Subjects**: The function iterates over subject folders, where each folder corresponds to a subject's data.

- **Recipe Folder Path**: A corresponding recipe folder (e.g., S01_Brownie_3DMGX1) is constructed inside each subject folder.

For example, if subject = "S01_Brownie_3DMGX1", then:
subject.split('_') # Output: ['S01', 'Brownie', '3DMGX1']

- This is used to access the **second element** in the list that was produced by the split('_') method.
- In Python, list indexing starts from 0, so [0] would refer to the first element, and [1] refers to the second element.
- For example:

subject.split('_')[1] # Output: 'Brownie'

recipe_path = os.path.join(subject_path, f"S{subject.split('_')[1]}_{recipe}_3DMGX1")
if the subject is "Subject_01" and the recipe is "Brownie", it creates a path like: /data/Subject_01/S01_Brownie_3DMGX1.

- **Reading Sensor Data**: The function loads the sensor data from a text file starting with '2796'. The pandas function read_csv() is used to read the file, with the delimiter set to tab (\t). This tells pd.read_csv to **skip the first 2 rows** of the file.

  Example: If the file looks like this:

  ```
  # Some metadata - Skipped row 1
  # More metadata – Skipped row 2
  Time    Accel_X    Accel_Y    Accel_Z
  0.0     1.23       4.56       7.89
  1.0     2.34       5.67       8.90
  ```

  - The first 2 rows are skipped (metadata).
  - The data is read into a DataFrame with columns like Time, Accel_X, Accel_Y, Accel_Z.

- **Data Cleaning**: The numeric columns (sensor readings) are identified, and non-numeric values are replaced with NaN. The rows with NaN values are dropped.

  Example:

  Let's say your DataFrame looks like this before applying these lines:

Time	Accel_X	Accel_Y	Accel_Z
1.0	0.5	1.2	3.1
2.0	0.6	abc	2.9
3.0	0.8	1.5	xyz

  pd.to_numeric would convert valid numbers and change any non-numeric values (like "abc" and "xyz") to NaN:

Time	Accel_X	Accel_Y	Accel_Z
1.0	0.5	1.2	3.1
2.0	0.6	NaN	2.9
3.0	0.8	1.5	NaN

  dropna(subset=numeric_columns) would then remove the rows that contain NaN values:

Time	Accel_X	Accel_Y	Accel_Z
1.0	0.5	1.2	3.1

- **Appending Clean Data**: Cleaned numeric data is stored in data[], and the recipe label (class) is stored in labels[].

2. Data Filtering :

- **Filter Empty Sequences**: The code removes empty sequences after cleaning. This ensures only valid data is used.

  Example:

  Let’s say **x** initially contains:

  ```
  X = [
  np.array([[1, 2], [3, 4]]),   # Valid sequence
  np.array([]),                 # Empty sequence
  np.array([[5, 6], [7, 8]])    # Valid sequence ]
  ```

  After filtering with **`X_filtered = [x for x in X if len(x) > 0]`**, you would get:

  ```
  X_filtered = [
  np.array([[1, 2], [3, 4]]),
  np.array([[5, 6], [7, 8]])]
  ```

- **Consistent Feature Size**: It checks that all data sequences have the same number of features (columns), and sequences with mismatched feature sizes are removed.

Example:

Let’s say `X` initially contains:

```
X = [
    np.array([[1, 2], [3, 4]]),     # Valid sequence with 2 features
    np.array([]),                   # Empty sequence
    np.array([[5, 6], [7, 8, 9]]), # Invalid sequence with 3 features
    np.array([[9, 10], [11, 12]])  # Valid sequence with 2 features]

And y contains the corresponding labels:

y = ['Brownie', 'Pizza', 'Eggs', 'Salad']

After filtering for non-empty sequences with the correct number of features
(let's assume n_features = 2), only the valid sequences remain:

X_filtered = [
    np.array([[1, 2], [3, 4]]),
    np.array([[9, 10], [11, 12]])]

So, the corresponding labels y_filtered would be:
y_filtered = ['Brownie', 'Salad']
```

The labels for the invalid or empty sequences are removed.

Padding Sequences: All sequences are padded to the same length using pad_sequences(). Padding is done by adding zeros to the end of the sequence if it's shorter than others.

If you have two sequences like this:

```
X_filtered = [
np.array([[1, 2], [3, 4]]),      # Length = 2
np.array([[5, 6], [7, 8], [9, 10]])  # Length = 3]
```

After applying `pad_sequences`:

```
    X_padded = [
   [[1, 2], [3, 4], [0, 0]],# Padded with zeros to make length 3
[[5, 6], [7, 8], [9, 10]]  # Already length 3, so no padding needed]
```

Scaling: The data is standardized using StandardScaler(), which normalizes the sensor readings to have a mean of 0 and a standard deviation of 1.

StandardScaler applies the following formula to each feature:

$$\text{Standardized value} = \frac{(x - \mu)}{\sigma}$$

Where:

- **x** is the original feature value.
- **µ (mu)** is the **mean** of the feature across all the data points.
- **σ (sigma)** is the **standard deviation** of the feature.

Example 1:

Let's say you have a sequence:

```
X_padded = np.array([[1.0, 200.0], [0.5, 150.0], [0.2, 180.0]])
```

Before scaling:

- The first column ranges from 0.2 to 1.0.
- The second column ranges from 150 to 200.

After applying **`StandardScaler`**:

```
X_scaled = [
    [-0.53, 1.22],  # Scaled first feature and second feature
    [ 0.23, -1.22],
    [ 0.53, 0.0]]
```

Now, both features are centred around zero and scaled, improving model performance

Example 2:

Let's say you have the following data points for a feature (e.g., sensor readings):
Original Data (x) = 10, 20, 30

1. Mean (μ):
   - $\mu = \frac{10+20+30}{3} = 20$
2. Standard Deviation (σ):
   - $\sigma = \sqrt{\frac{(10-20)^2+(20-20)^2+(30-20)^2}{3}} = 8.16$
3. Standardization:
   - For each value in the column:
   - $10 : \frac{(10-20)}{8.16} = -1.22$
   - $20 : \frac{(20-20)}{8.16} = 0$
   - $30 : \frac{(30-20)}{8.16} = 1.22$

So the normalized values become:

$$[-1.22, 0, 1.22]$$

Label Encoding: Recipe labels (like "Brownie") are converted into numeric values (e.g., Brownie → 0, Pizza → 1) using LabelEncoder().

recipes = ["Brownie", "Pizza", "Eggs", "Salad", "Sandwich"]
# After applying LabelEncoder encoded_labels = [0, 1, 2, 3, 4]

Printing the Label Distribution to make sure all the instances are labelled correctly:

- **unique**: The unique elements (the different labels in **y_filtered**).
- **counts**: The number of occurrences for each unique element (how many times each label appears).
- **zip(unique, counts)**: Pairs each unique label with its count.
- **dict()**: Converts the pairs into a dictionary.

Example:
unique = ['Brownie', 'Pizza', 'Salad']
counts = [1, 3, 1]
label_distribution = dict(zip(unique, counts))
# label_distribution will be: {'Brownie': 1, 'Pizza': 3, 'Salad': 1}

3. Train-Test Split

- **80-20 Split**: The data is split into training (80%) and testing (20%) sets using train_test_split().
- **Stratify**: The stratify=y_encoded parameter ensures that the split maintains the same proportion of classes (recipes) in both the training and test sets.

Let's say you have a dataset with 100 samples, and the distribution of classes is:

- **Pizza**: 50 samples
- **Brownie**: 30 samples
- **Salad**: 20 samples

Without stratification,

- **Training set** (80 samples): 45 Pizza, 25 Brownie, 10 Salad
- **Test set** (20 samples): 5 Pizza, 5 Brownie, 10 Salad

With **stratify=y_encoded**, the split will maintain the original distribution:

- **Training set**: 80 samples, with 40 Pizza, 24 Brownie, 16 Salad
- **Test set**: 20 samples, with 10 Pizza, 6 Brownie, 4 Salad

4. LSTM RNN Model Architecture

- **Sequential Model**: The model is built layer by layer using the Sequential() API from Keras.

- **Input Layer**: The input layer defines the shape of the data fed into the LSTM, where X_train.shape[1] is the sequence length, and X_train.shape[2] is the number of features.

- **LSTM Layer**: A single LSTM layer with 16 neurons is used. LSTM is ideal for sequential data because it can retain information from previous steps in the sequence.

- **Dense Layers**: There are 12 fully connected (Dense) layers with varying numbers of neurons (128, 64, 32, 16). ReLU and Tanh activation functions are used for non-linearity.

- **Output Layer**: The final layer uses softmax activation, which outputs the probabilities for each class (recipe). The number of neurons is based on the number of unique classes (5 recipes).

5. Model Compilation

- **Optimizer**: The model uses RMSprop as the optimizer, which is commonly used for RNNs.

- **Loss Function**: The loss function is categorical_crossentropy, which is ideal for multi-class classification.

- **Metrics**: The model tracks accuracy during training.

6. Model Training

- **Training**: The model is trained for 50 epochs with a batch size of 100. X_train and y_train are used for training, while X_test and y_test are used for validation.

- **One-Hot Encoding**: The labels are converted to one-hot encoded format using tf.keras.utils.to_categorical() before training.

7. Evaluation

- **Prediction**: After training, the model is used to predict the class labels for X_test.

- **Confusion Matrix**: A confusion matrix is computed using confusion_matrix() to compare the true labels (y_test) with the predicted labels (y_pred).

- **Confusion Matrix Visualization**: The confusion matrix is visualized using seaborn.heatmap(). The cmap='viridis' defines the color scheme used.

8. Model Metrics

- The following metrics are calculated for model evaluation:
    - **Accuracy**: Overall classification accuracy.
    - **Precision**: How many of the predicted positive cases were actually positive (calculated per class).
    - **Recall**: The ability to find all the positive cases (calculated per class).
    - **F1 Score**: A balance between precision and recall.
    - **Macroaverage**: The average precision, recall, and F1 score across all classes.

Summary of Program:

This LSTM-based RNN model is designed to classify five recipes based on sensor data. The model is structured with multiple layers: an LSTM layer to capture sequential dependencies, followed by fully connected (Dense) layers for feature extraction and classification. The model uses RMSprop as the optimizer and categorical cross-entropy for the loss function.
The data is carefully preprocessed, including scaling, label encoding, and padding. The model is evaluated based on accuracy, precision, recall, and F1 score, with a confusion matrix providing a detailed view of performance per class.

Possible Improvements for Future Work:

- **Hyperparameter tuning**: Experiment with different numbers of LSTM units, learning rates, and optimizers.
- **Increase LSTM layers**: Adding more LSTM layers might capture more complex sequential patterns.
- **Data Augmentation**: If the dataset is small, data augmentation could help improve generalization.

My Observations in this Project:

1. Having too many dense layers makes it harder to train the model; the best results come from having 1 to 4 layers of dense layers with a Drop out of 0.5.

2. The ideal RNN architecture giving good results with all variants of RNN was

```
model.add(LSTM(10, return_sequences=False))
model.add(Dropout(0.5))
model.add(Dense(60, activation='tanh'))
model.add(Dense(30, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(60, activation='tanh'))
model.add(Dense(30, activation='relu'))
model.add(Dropout(0.5))
```

NOTE: Although more neurons in the RNN layer will produce better results, due to resource constraints, more than 16 neurons could only be used in a few trials. RNNs performed well when there were more than 16 number neurons.

3. After multiple trials with different Optimizers like Adam, SGD, and Loss functions like – Sparse Categorical Cross entropy, and Hueber loss.
   The RMS-prop optimizer and Categorical Cross entropy loss function gave the best results(So Mostly Used in the project).

4. Also, the sequence of order in which the files are read from the dataset changes from platform to platform (IDEs)

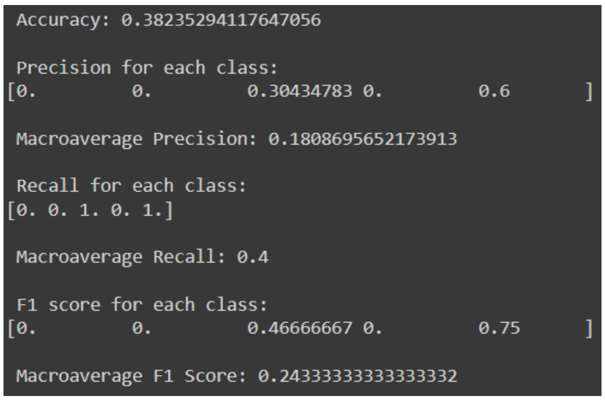
5. One of the reasons for less accuracy is that the data set may have less number of files as it might be less to train the model and get good accuracy. Also as the data is from the sensors and might be prone to noise and distortion.

6. I Also did a special trial where I defined how much the maximum length of the sequence should be taken as the input like some percentage of the original sequence to be taken as the input for the model.

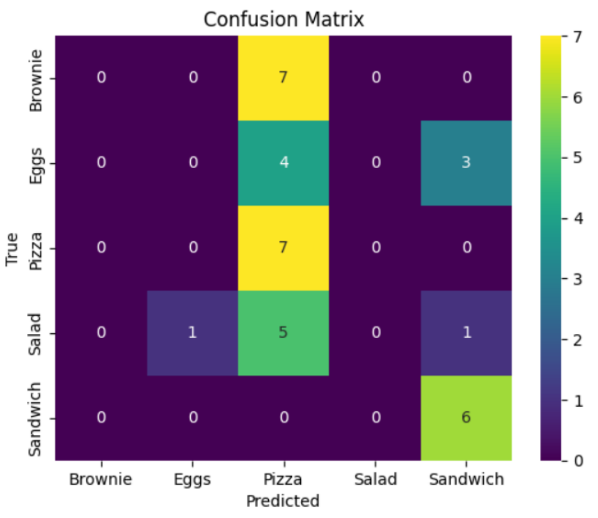
Trial 1:

SIMPLE RNN OUTPUT

LAYERS	OPTIMIZER	LOSS	EPOCHS	BATCH SIZE
There are 1 Input Layer; 1 Simple RNN Layer with 32 neurons, and 4 Dense layers with numbers of neurons (128, 64, 32, 16) with ReLU and Tanh activation functions and 1 output Layer	adam	categorical_ crossentropy	30	128



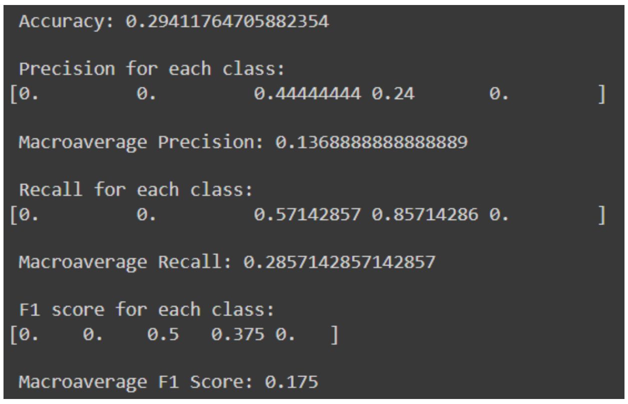
```
Accuracy: 0.38235294117647056

Precision for each class:
[0.         0.         0.30434783 0.         0.6       ]

Macroaverage Precision: 0.1808695652173913

Recall for each class:
[0. 0. 1. 0. 1.]

Macroaverage Recall: 0.4

F1 score for each class:
[0.         0.         0.46666667 0.         0.75      ]

Macroaverage F1 Score: 0.24333333333333332
```



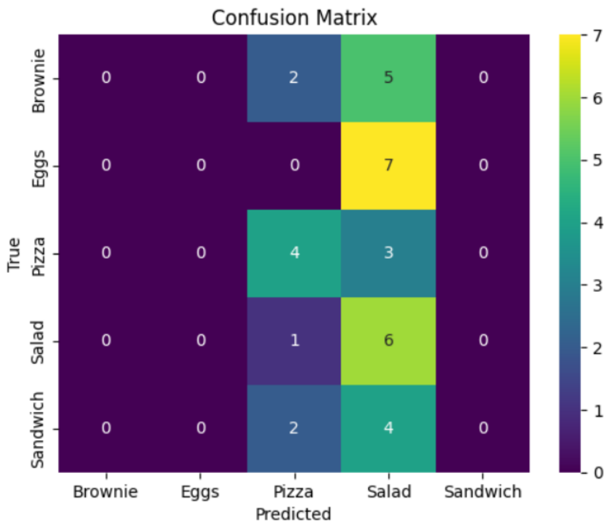
==

Trial 2: SIMPLE RNN

There are 1 Input Layer; 1 Simple RNN Layer with 32 neurons, and 6 Dense layers with numbers of neurons (128 128,64 64,32 32) with ReLU activation functions , Dropout - 0.3, and 1 output Layer	adam	categorical_ crossentropy	30	64



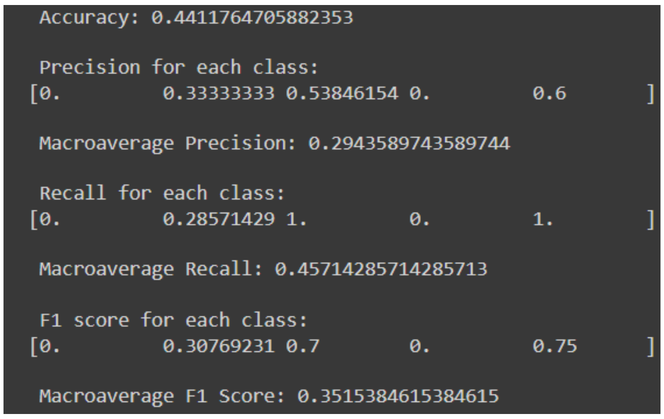
```
Accuracy: 0.29411764705882354

Precision for each class:
[0.         0.         0.44444444 0.24       0.        ]

Macroaverage Precision: 0.1368888888888889

Recall for each class:
[0.         0.         0.57142857 0.85714286 0.        ]

Macroaverage Recall: 0.2857142857142857

F1 score for each class:
[0.    0.    0.5   0.375 0.   ]

Macroaverage F1 Score: 0.175
```



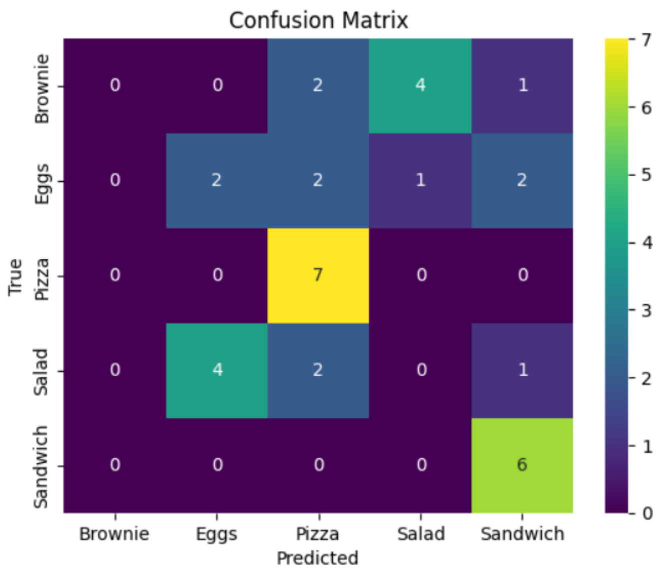
==

Trial 3: SIMPLE RNN

There are 1 Input Layer; 1 Simple RNN Layer with 32 neurons, and 9 Dense layers with numbers of neurons (128 128 128,64 64 64,32 32 32) with ReLU activation functions and 1 output Layer	adam	categorical_ crossentropy	50	128

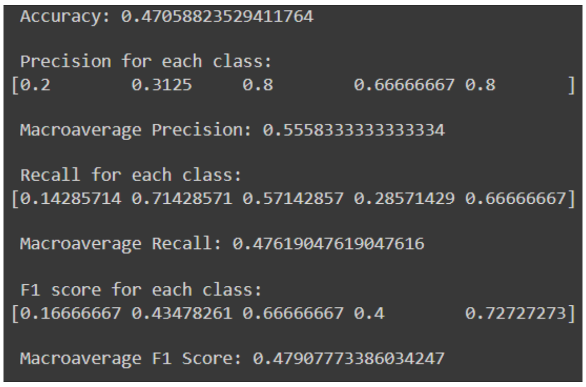


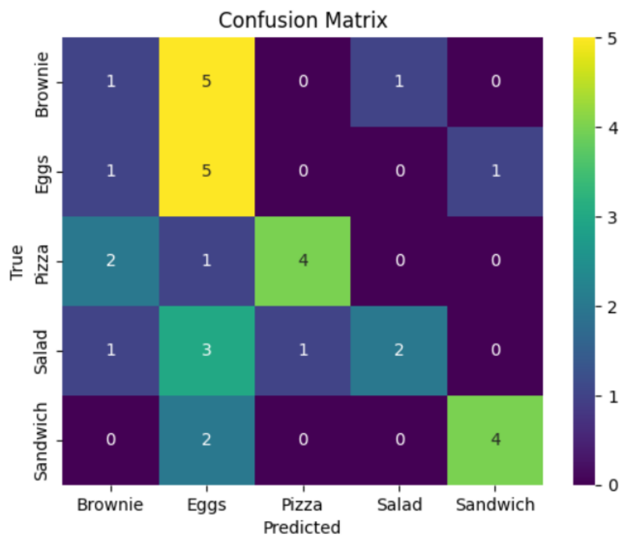
```
Accuracy: 0.4411764705882353

Precision for each class:
[0.         0.33333333 0.53846154 0.         0.6       ]

Macroaverage Precision: 0.2943589743589744

Recall for each class:
[0.         0.28571429 1.         0.         1.        ]

Macroaverage Recall: 0.45714285714285713

F1 score for each class:
[0.         0.30769231 0.7        0.         0.75      ]

Macroaverage F1 Score: 0.3515384615384615
```



Trial 4: SIMPLE RNN

There are 1 Input Layer; 1 Simple RNN Layer with 32 neurons, and 12 Dense layers with numbers of neurons (128 128 128,64 64 64,32 32 32,16 16 16) with ReLU and Tanh activation functions , and 1 output Layer	RMSprop	categorical_ crossentropy	50	100

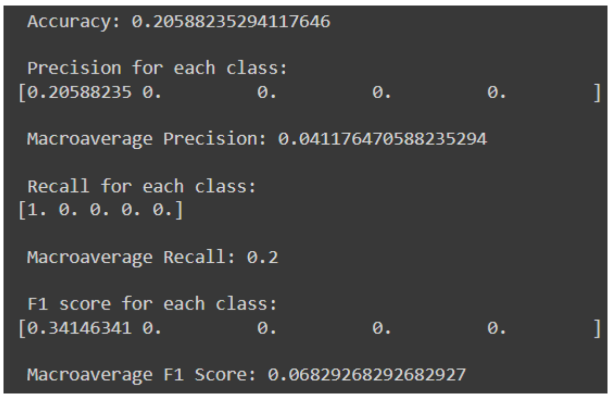

```
Accuracy: 0.47058823529411764

Precision for each class:
[0.2        0.3125     0.8        0.66666667 0.8       ]

Macroaverage Precision: 0.5558333333333334

Recall for each class:
[0.14285714 0.71428571 0.57142857 0.28571429 0.66666667]

Macroaverage Recall: 0.47619047619047616

F1 score for each class:
[0.16666667 0.43478261 0.66666667 0.4        0.72727273]

Macroaverage F1 Score: 0.47907773386034247
```

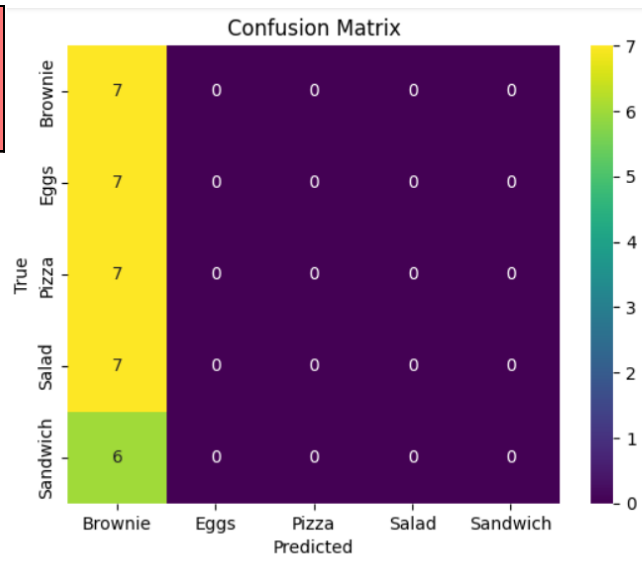


===

Trial 5: SIMPLE RNN

There are 1 Input Layer; 1 Simple RNN Layer with 8 neurons, 4 Dense layers with numbers of neurons (64,32,16,64) with Sigmoid and relu activation functions ,and 1 output Layer	SGD	sparse_categorical_crossentropy	40	80


```
Accuracy: 0.20588235294117646

Precision for each class:
[0.20588235 0.         0.         0.         0.        ]

Macroaverage Precision: 0.041176470588235294

Recall for each class:
[1. 0. 0. 0. 0.]

Macroaverage Recall: 0.2

F1 score for each class:
[0.34146341 0.         0.         0.         0.        ]

Macroaverage F1 Score: 0.06829268292682927
```

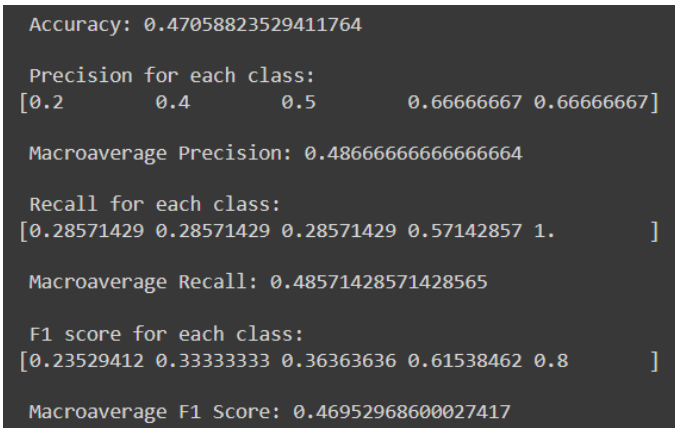


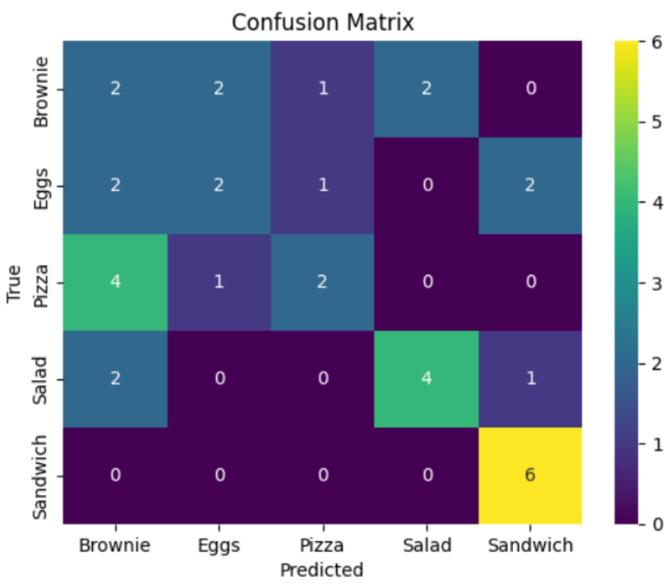
===

LSTM RNN OUTPUTS

Trial 1:

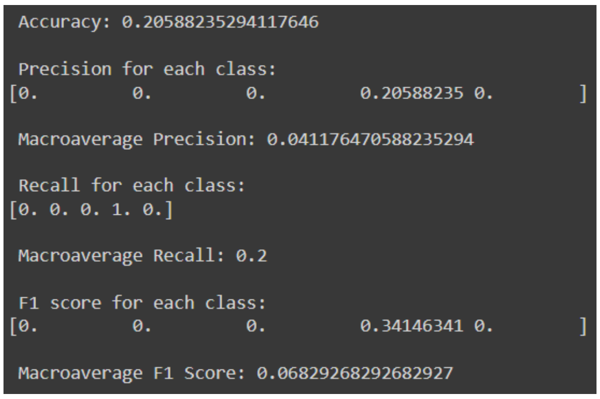
LAYERS	OPTIMIZER	LOSS	EPOCHS	BATCH SIZE
There are 1 Input Layer; 1 LSTM RNN Layer with 16 neurons, 12 Dense layers with numbers of neurons (28 128 128,64 64 64,32 32 32,16 16 16) with ReLU & tanh activation functions ,and 1 output Layer	RMSprop	categorical_crossentropy	50	100


```
Accuracy: 0.47058823529411764

Precision for each class:
[0.2        0.4        0.5        0.66666667 0.66666667]

Macroaverage Precision: 0.48666666666666664

Recall for each class:
[0.28571429 0.28571429 0.28571429 0.57142857 1.        ]

Macroaverage Recall: 0.48571428571428565

F1 score for each class:
[0.23529412 0.33333333 0.36363636 0.61538462 0.8       ]

Macroaverage F1 Score: 0.46952968600027417
```

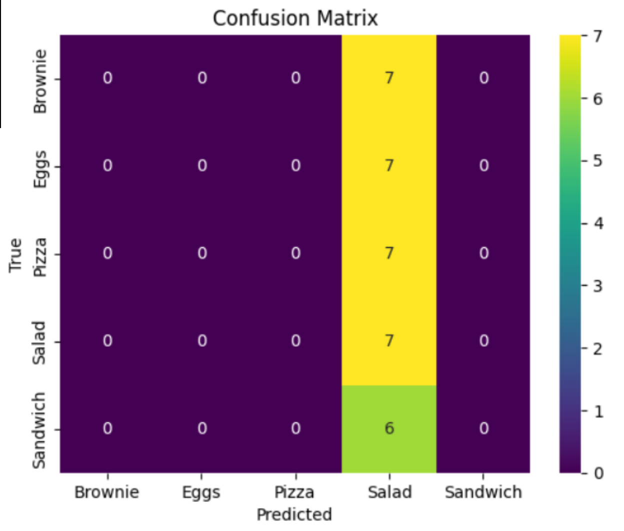


Trial 2: LSTM

There are 1 Input Layer; 1 LSTM RNN Layer with 18 neurons, 28 Dense layers with numbers of neurons (four of 128 , four of 64,four of 32,four of 16 with ReLU activation functions ,1 Dropout 0.2 at end ,and 1 output Layer	adam	categorical_crossentropy	55	32



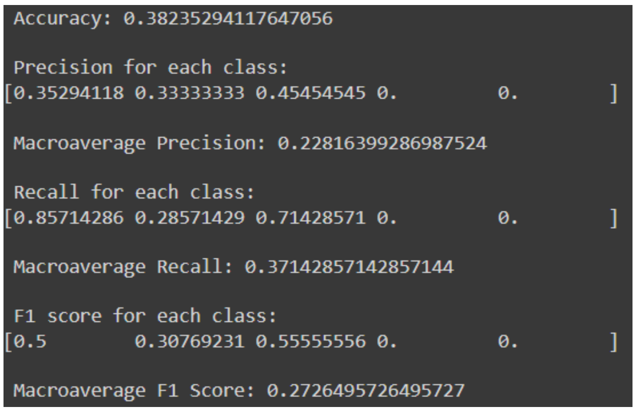
```
 Accuracy: 0.20588235294117646

 Precision for each class:
[0.         0.         0.         0.20588235 0.        ]

 Macroaverage Precision: 0.041176470588235294

 Recall for each class:
[0. 0. 0. 1. 0.]

 Macroaverage Recall: 0.2

 F1 score for each class:
[0.         0.         0.         0.34146341 0.        ]

 Macroaverage F1 Score: 0.06829268292682927
```



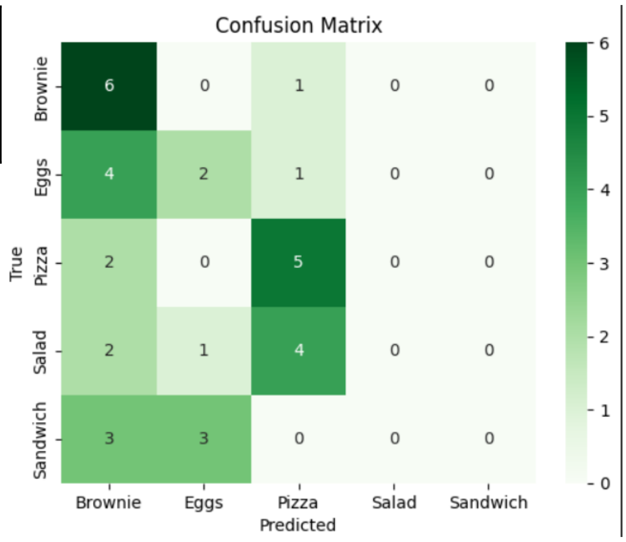
==

Trial 3: LSTM

There are 1 Input Layer; 1 LSTM RNN Layer with 8 neurons, 7 Dense layers with numbers of neurons (128,64,32,16,8,4,2) with all tanh activation functions ,and 1 output Layer	RMSprop	categorical_crossentropy	30	128



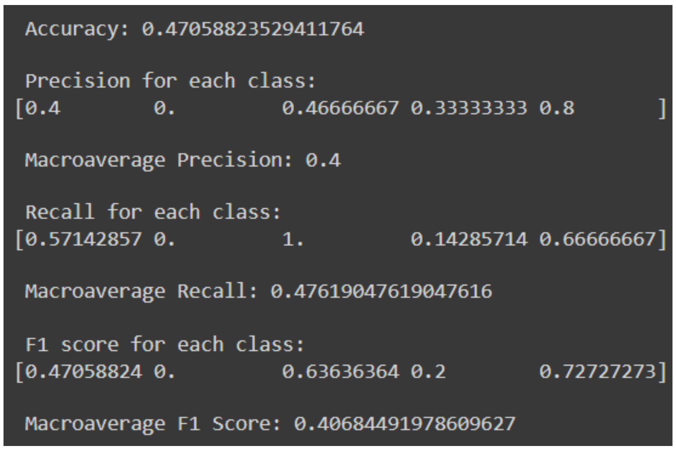
```
 Accuracy: 0.38235294117647056

 Precision for each class:
[0.35294118 0.33333333 0.45454545 0.         0.        ]

 Macroaverage Precision: 0.22816399286987524

 Recall for each class:
[0.85714286 0.28571429 0.71428571 0.         0.        ]

 Macroaverage Recall: 0.37142857142857144

 F1 score for each class:
[0.5        0.30769231 0.55555556 0.         0.        ]

 Macroaverage F1 Score: 0.2726495726495727
```



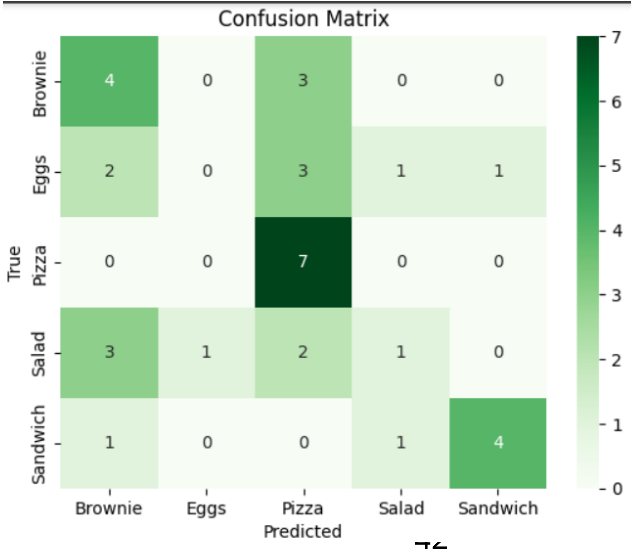
==

Trial 4: LSTM

There are 1 Input Layer; 1 LSTM RNN Layer with 10 neurons,Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) and again Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) with ReLU & tanh activation functions , and again Dropout Layer - 0.5 and, 1 output Layer	RMSprop	categorical_crossentropy	50	100

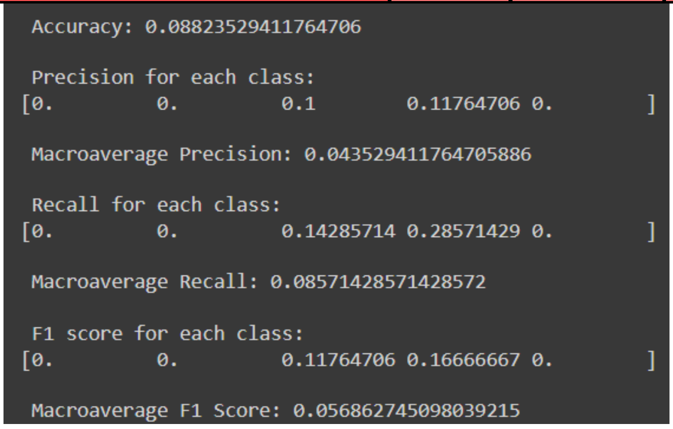


```
 Accuracy: 0.47058823529411764

 Precision for each class:
[0.4        0.         0.46666667 0.33333333 0.8       ]

 Macroaverage Precision: 0.4

 Recall for each class:
[0.57142857 0.         1.         0.14285714 0.66666667]

 Macroaverage Recall: 0.47619047619047616

 F1 score for each class:
[0.47058824 0.         0.63636364 0.2        0.72727273]

 Macroaverage F1 Score: 0.40684491978609627
```

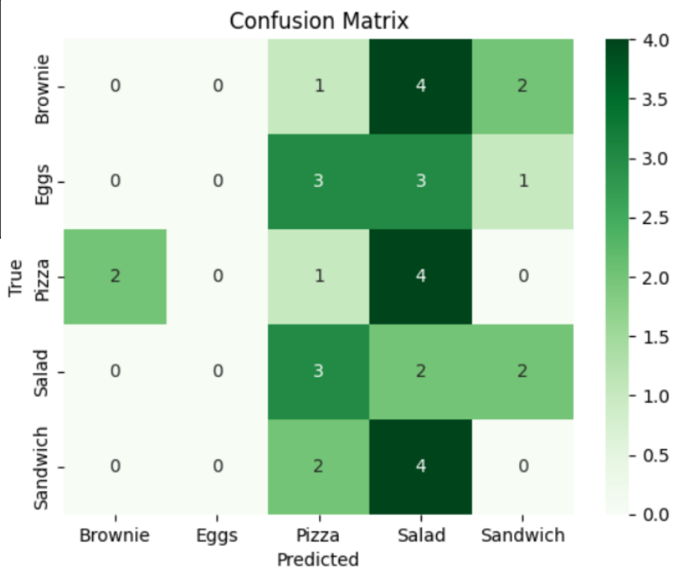


Trial 5: LSTM

There are 1 Input Layer; 1 LSTM RNN Layer with 10 neurons,Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) and again Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) with ReLU & tanh activation functions , and again Dropout Layer - 0.5 and, 1 output Laye	SGD	categorical_crossentropy	50	100



```
 Accuracy: 0.08823529411764706

 Precision for each class:
[0.         0.         0.1        0.11764706 0.        ]

 Macroaverage Precision: 0.043529411764705886

 Recall for each class:
[0.         0.         0.14285714 0.28571429 0.        ]

 Macroaverage Recall: 0.08571428571428572

 F1 score for each class:
[0.         0.         0.11764706 0.16666667 0.        ]

 Macroaverage F1 Score: 0.056862745098039215
```

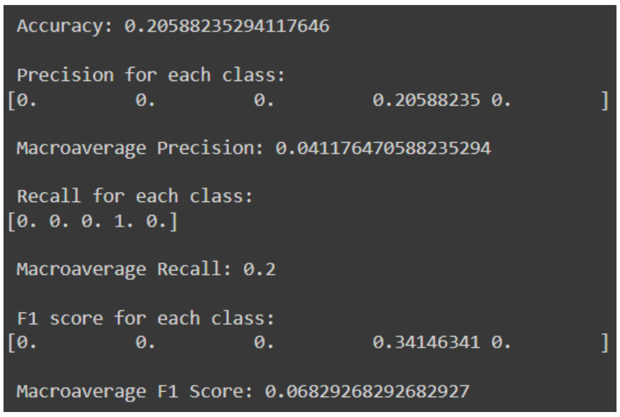


==

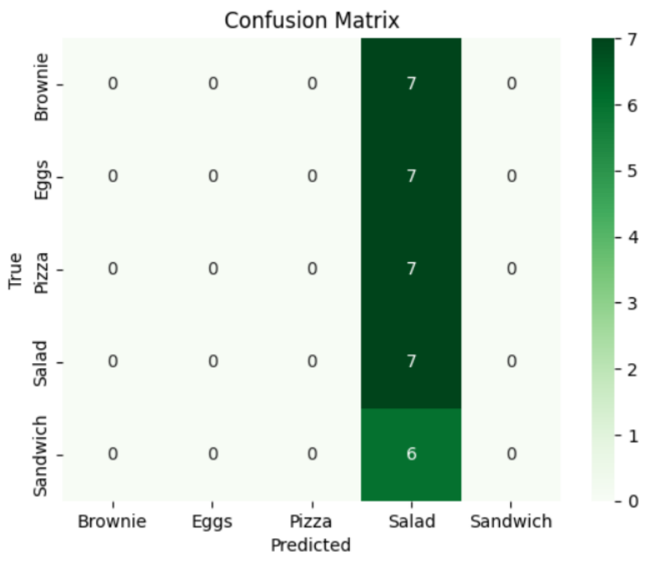
GRU RNN OUTPUT

Trial 1: GRU

There are 1 Input Layer; 1 GRU RNN Layer with 16 neurons, and 28 Dense layers with numbers of neurons (128,64,32,16,8,4,2) with all ReLU activation functions , and 1 output Layer	adam	categorical_crossentropy	55	80



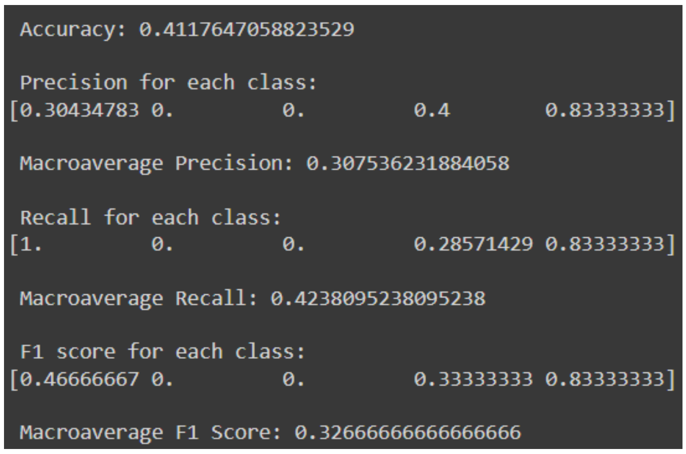
```
Accuracy: 0.20588235294117646

Precision for each class:
[0.         0.         0.         0.20588235 0.        ]

Macroaverage Precision: 0.041176470588235294

Recall for each class:
[0. 0. 0. 1. 0.]

Macroaverage Recall: 0.2

F1 score for each class:
[0.         0.         0.         0.34146341 0.        ]

Macroaverage F1 Score: 0.06829268292682927
```



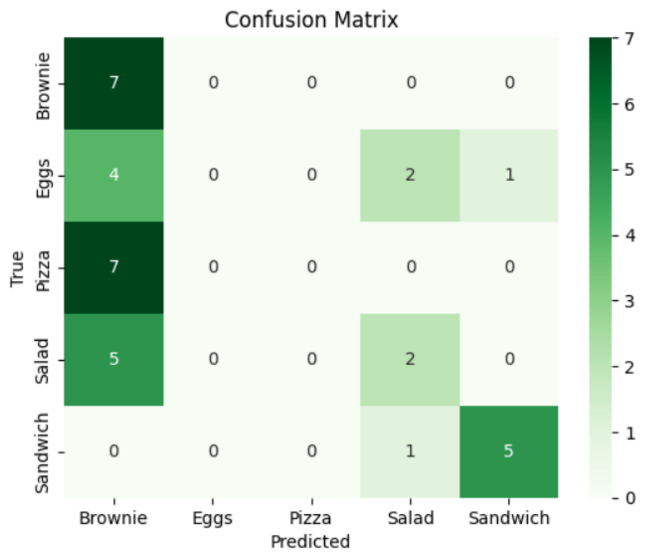
==

Trial 2: GRU

There are 1 Input Layer; 1 GRU RNN Layer with 8 neurons, and 10 Dense layers with numbers of neurons (150,100,80,50,30,18,72,90,4,4) with ReLU & tanh activation functions , and 1 output Layer	RMS_prop	focal_loss	45	650

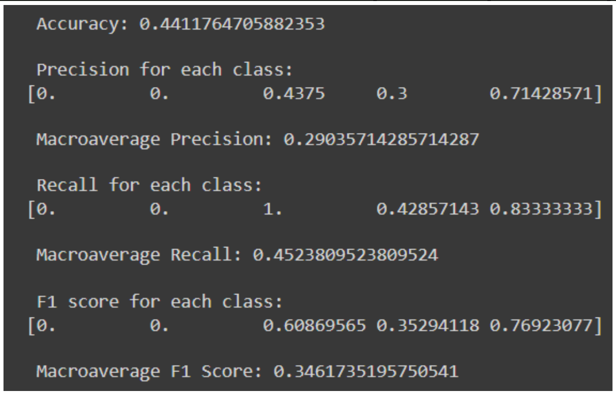


```
 Accuracy: 0.4117647058823529

 Precision for each class:
[0.30434783 0.         0.         0.4        0.83333333]

 Macroaverage Precision: 0.307536231884058

 Recall for each class:
[1.         0.         0.         0.28571429 0.83333333]

 Macroaverage Recall: 0.4238095238095238

 F1 score for each class:
[0.46666667 0.         0.         0.33333333 0.83333333]

 Macroaverage F1 Score: 0.32666666666666666
```

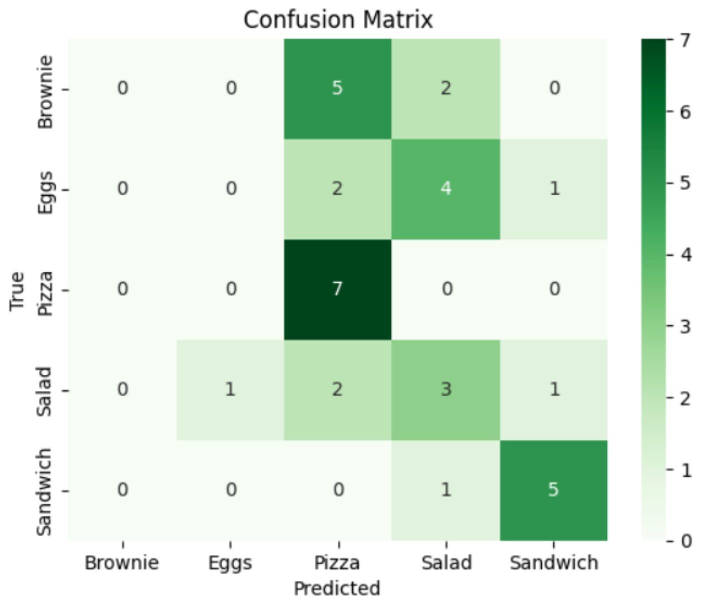


Trial 3: GRU

There are 1 Input Layer; 1 GRU RNN Layer with 10 neurons,Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) and again Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) with ReLU & tanh activation functions , and again Dropout Layer - 0.5 and, 1 output Layer	RMS_prop	categorical_ crossentropy	50	420



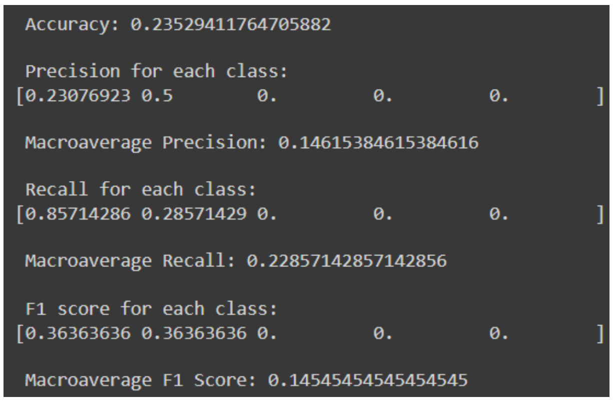
```
 Accuracy: 0.4411764705882353

 Precision for each class:
[0.         0.         0.4375     0.3        0.71428571]

 Macroaverage Precision: 0.29035714285714287

 Recall for each class:
[0.         0.         1.         0.42857143 0.83333333]

 Macroaverage Recall: 0.4523809523809524

 F1 score for each class:
[0.         0.         0.60869565 0.35294118 0.76923077]

 Macroaverage F1 Score: 0.3461735195750541
```



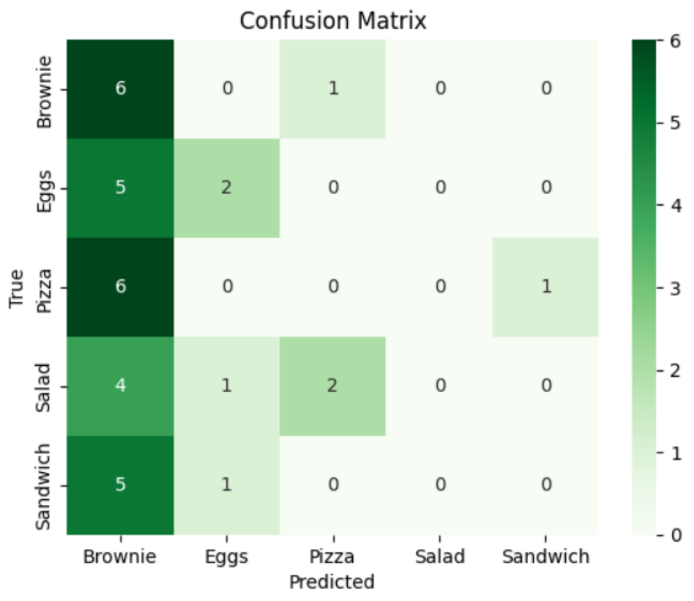
==

Trial 4: GRU

There are 1 Input Layer; 1 GRU RNN Layer with 10 neurons,Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) and again Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) with ReLU & tanh activation functions , and again Dropout Layer - 0.5 and, 1 output Layer	SGD	Huber Loss	50	420



```
 Accuracy: 0.23529411764705882

 Precision for each class:
[0.23076923 0.5        0.         0.         0.         ]

 Macroaverage Precision: 0.14615384615384616

 Recall for each class:
[0.85714286 0.28571429 0.         0.         0.         ]

 Macroaverage Recall: 0.22857142857142856

 F1 score for each class:
[0.36363636 0.36363636 0.         0.         0.         ]

 Macroaverage F1 Score: 0.14545454545454545
```



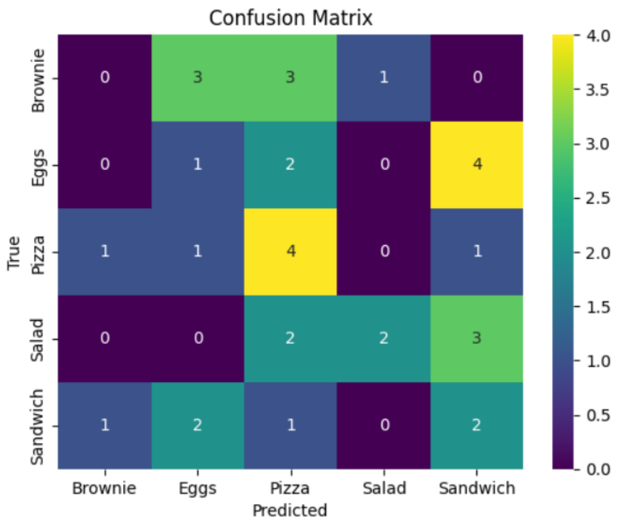
==

BiLSTM RNN OUTPUT

Trial 1: BiLSTM

There are 1 Input Layer; 1 BiLSTM RNN Layer with 8 neurons, and 3 Dense layers with numbers of neurons (32,16,8) with tanh and ReLU activation functions , and 1 output Layer	RMSprop	categorical_ crossentropy	30	32

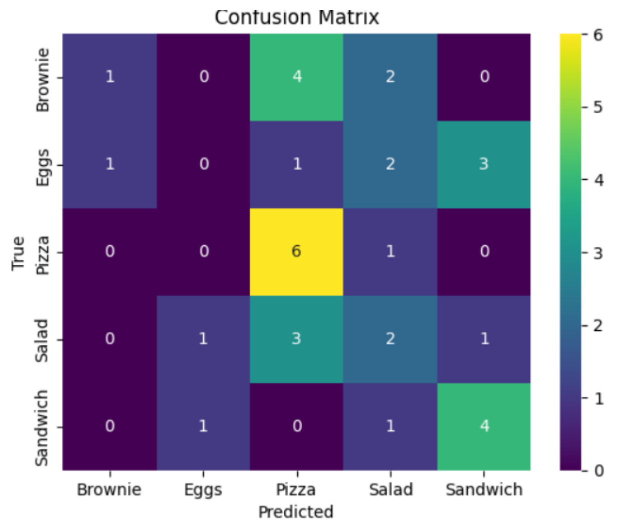
```
 Accuracy: 0.2647058823529412

 Precision for each class:
[0.         0.14285714 0.33333333 0.66666667 0.2        ]

 Macroaverage Precision: 0.26857142857142857

 Recall for each class:
[0.         0.14285714 0.57142857 0.28571429 0.33333333]

 Macroaverage Recall: 0.26666666666666666

 F1 score for each class:
[0.         0.14285714 0.42105263 0.4        0.25       ]

 Macroaverage F1 Score: 0.24278195488721802
```



Trial 2: BiLSTM

There are 1 Input Layer; 1 BiLSTM RNN Layer with 7 neurons, and 8 Dense layers with numbers of neurons (128,64,32,16) with ReLU and tanh activation functions , 2 **NO Dropout layers** , and 1 output Layer	RMSprop	categorical_ crossentropy	30	90

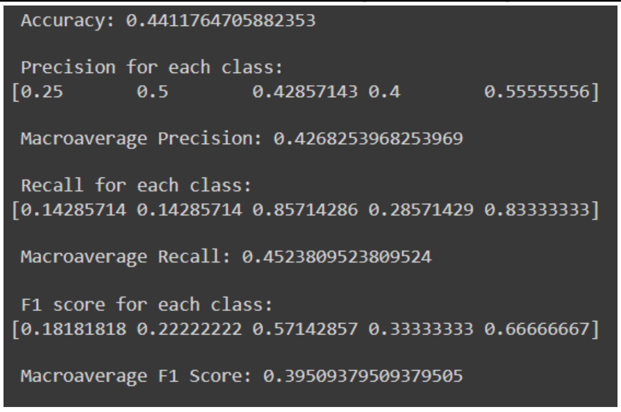
```
 Accuracy: 0.38235294117647056

 Precision for each class:
[0.5        0.         0.42857143 0.25       0.5       ]

 Macroaverage Precision: 0.33571428571428574

 Recall for each class:
[0.14285714 0.         0.85714286 0.28571429 0.66666667]

 Macroaverage Recall: 0.3904761904761904

 F1 score for each class:
[0.22222222 0.         0.57142857 0.26666667 0.57142857]

 Macroaverage F1 Score: 0.32634920634920633
```



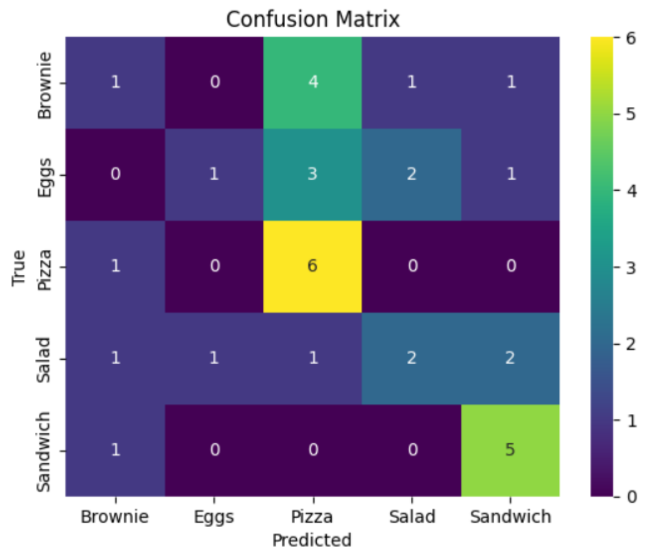
===

Trial 3: BiLSTM

There are 1 Input Layer; 1 BiLSTM RNN Layer with 10 neurons,Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) and again Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) with ReLU & tanh activation functions , and again Dropout Layer - 0.5 and, 1 output Layer	RMSprop	categorical_ crossentropy	50	64



```
 Accuracy: 0.4411764705882353

 Precision for each class:
[0.25       0.5        0.42857143 0.4        0.55555556]

 Macroaverage Precision: 0.4268253968253969

 Recall for each class:
[0.14285714 0.14285714 0.85714286 0.28571429 0.83333333]

 Macroaverage Recall: 0.4523809523809524

 F1 score for each class:
[0.18181818 0.22222222 0.57142857 0.33333333 0.66666667]

 Macroaverage F1 Score: 0.39509379509379505
```

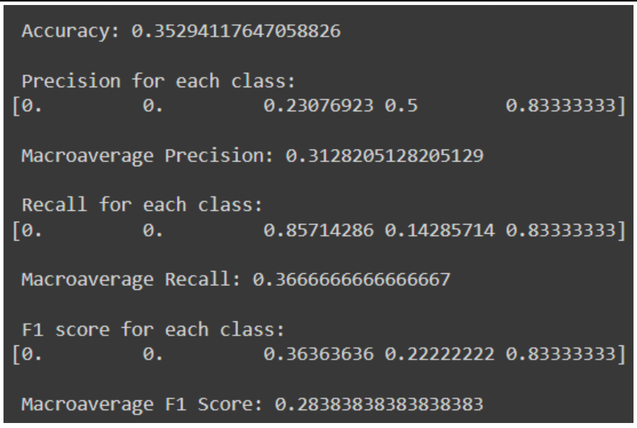


===

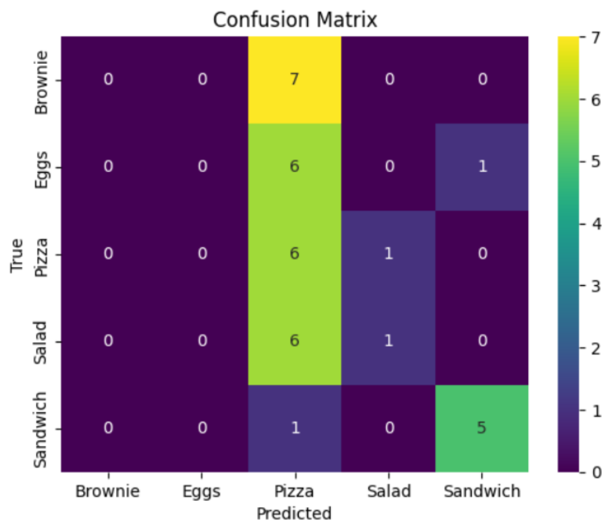
OUTPUT OF SPECIAL TRIALS OF ALL VARIANTS OF RNN

LSTM

	LAYERS	OPTIMIZER	LOSS	EPOCHS	BATCH SIZE
Special Trial Percentage of 25% which means 25% of the sequence is considered	There are 1 Input Layer; 1 LSTM RNN Layer with 10 neurons,Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) and again Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) with ReLU & tanh activation functions , and again Dropout Layer - 0.5 and, 1 output Layer	RMSprop	categorical_crossentropy	50	64

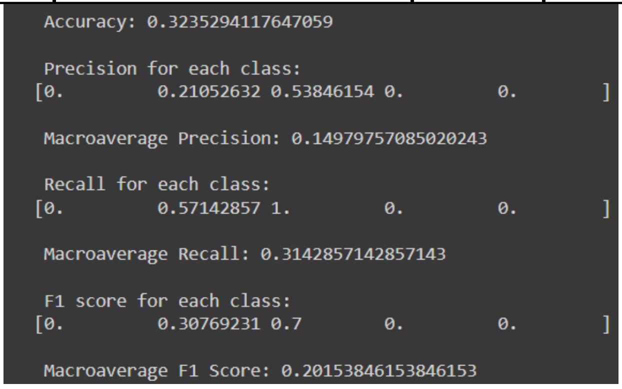


```
 Accuracy: 0.35294117647058826

 Precision for each class:
[0.         0.         0.23076923 0.5        0.83333333]

 Macroaverage Precision: 0.3128205128205129

 Recall for each class:
[0.         0.         0.85714286 0.14285714 0.83333333]

 Macroaverage Recall: 0.36666666666666667

 F1 score for each class:
[0.         0.         0.36363636 0.22222222 0.83333333]

 Macroaverage F1 Score: 0.28383838383838383
```

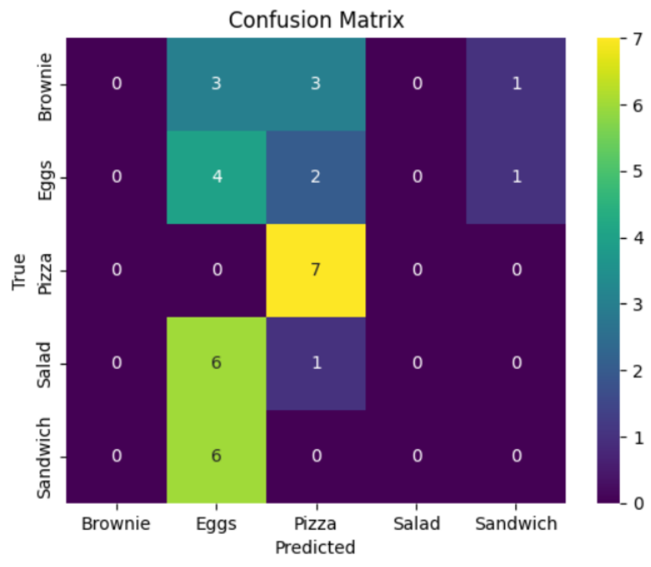


BiLSTM

Special Trial Percentage of 60% which means 60% of the sequence is considered	There are 1 Input Layer; 1 BiLSTM RNN Layer with 10 neurons,Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) and again Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) with ReLU & tanh activation functions , and again Dropout Layer - 0.5 and, 1 output Layer	RMSprop	categorical_ crossentropy	50	64



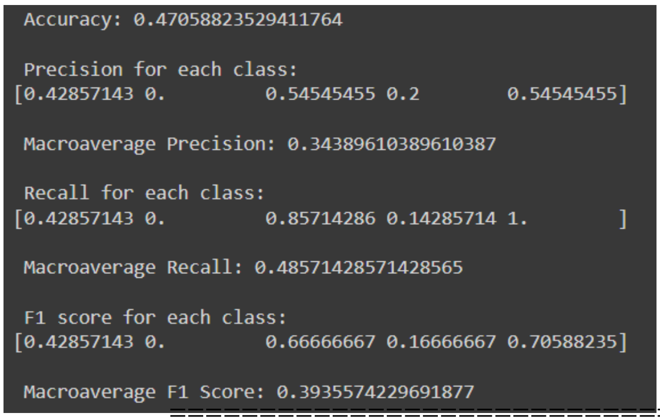
```
 Accuracy: 0.3235294117647059

 Precision for each class:
[0.         0.21052632 0.53846154 0.         0.        ]

 Macroaverage Precision: 0.14979757085020243

 Recall for each class:
[0.         0.57142857 1.         0.         0.        ]

 Macroaverage Recall: 0.3142857142857143

 F1 score for each class:
[0.         0.30769231 0.7        0.         0.        ]

 Macroaverage F1 Score: 0.20153846153846153
```



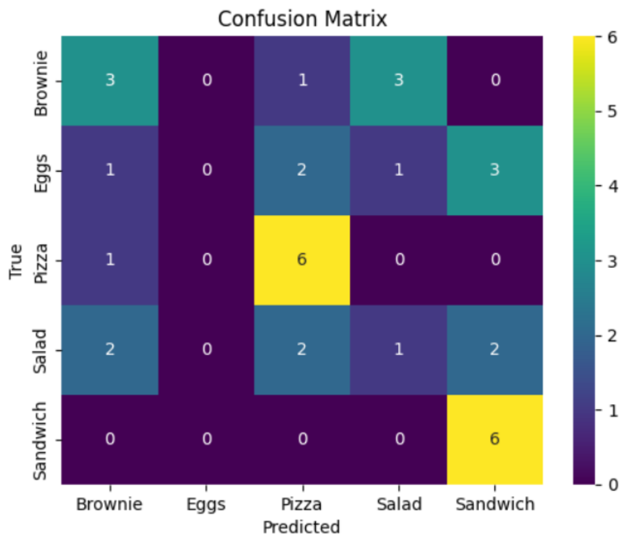
==

GRU

Special Trial Value 5000 which means 5000 sequences will be considered	There are 1 Input Layer; 1 GRU RNN Layer with 10 neurons,Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) and again Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) with ReLU & tanh activation functions , and again Dropout Layer - 0.5 and, 1 output Layer	RMSprop	categorical _crossentro py	50	50



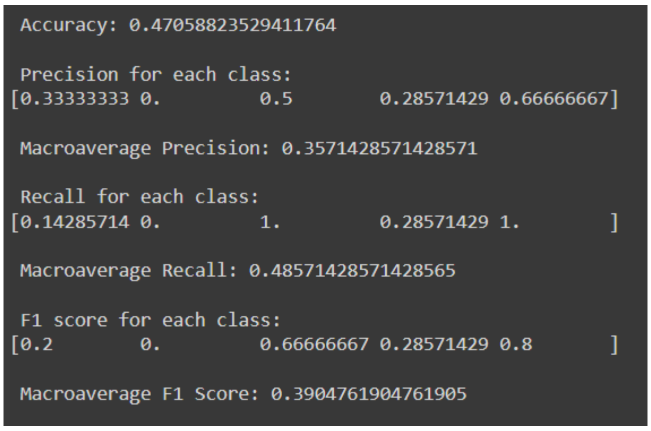
```
 Accuracy: 0.47058823529411764

 Precision for each class:
[0.42857143 0.         0.54545455 0.2        0.54545455]

 Macroaverage Precision: 0.34389610389610387

 Recall for each class:
[0.42857143 0.         0.85714286 0.14285714 1.        ]

 Macroaverage Recall: 0.48571428571428565

 F1 score for each class:
[0.42857143 0.         0.66666667 0.16666667 0.70588235]

 Macroaverage F1 Score: 0.3935574229691877
```



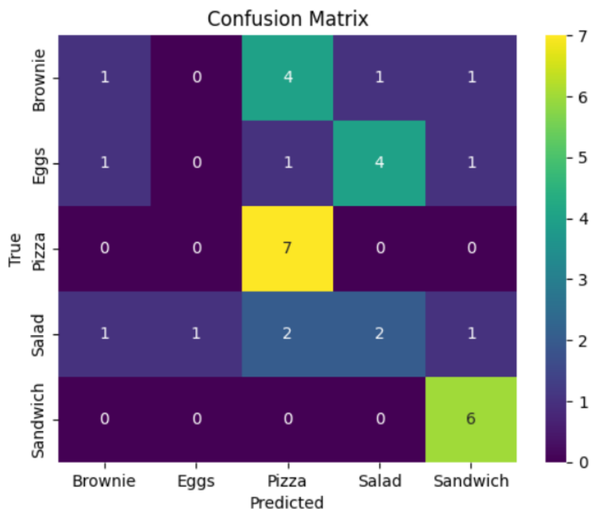
==

SIMPLE RNN

Special Trial Value 125, which means 125 sequences will be considered	There are 1 Input Layer; 1 Simple RNN Layer with 10 neurons,Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) and again Dropout Layer - 0.5 and 2 Dense layers with numbers of neurons (60,30) with ReLU & tanh activation functions , and again Dropout Layer - 0.5 and, 1 output Layer	RMSprop	categorical_ crossentropy	50	50



```
 Accuracy: 0.47058823529411764

 Precision for each class:
[0.33333333 0.         0.5        0.28571429 0.66666667]

 Macroaverage Precision: 0.3571428571428571

 Recall for each class:
[0.14285714 0.         1.         0.28571429 1.        ]

 Macroaverage Recall: 0.48571428571428565

 F1 score for each class:
[0.2        0.         0.66666667 0.28571429 0.8        ]

 Macroaverage F1 Score: 0.3904761904761905
```



The full Excel sheet of outputs for the RNN strategy is embedded below.

RNN_Project_Strategy
_Nikhil_Kumar.xlsx

Challenges I faced:

1. **Loading and Preprocessing the Dataset:**
   It was a challenging task to load and preprocess the large, complex dataset with multiple sensor readings, making the initial steps time-consuming and error-prone.

2. **RAM Issues:**
   Processing large datasets can consume significant memory, leading to potential RAM limitations that affect execution.

3. **Scalability:**
   Preprocessing large datasets can become computationally intensive, requiring efficient memory and time management, especially for high-frequency sensor data.

4. **Sensor Noise:**
   The presence of noise in the sensor data could introduce errors in the model's predictions, affecting its overall accuracy. Real-world sensor data often contains noise or errors that need to be cleaned, but excessive cleaning might result in loss of important information

Conclusion

In this project, we explored recipe classification using the CMU-MMAC dataset, leveraging advanced machine-learning techniques such as RNNs, LSTMs, and GRUs. By utilizing data from an Inertial Measurement Unit (IMU), we aimed to capture the intricate motion patterns involved in cooking activities. The results demonstrated the potential of deep learning in recognizing complex patterns within sequential data, highlighting its applicability in real-world scenarios like smart kitchens and automated cooking assistance. However, challenges such as noise and distortion in sensor readings significantly impacted the accuracy of our models, indicating the need for robust data cleaning and preprocessing techniques.

After many trials and observations, I was able to determine that 47% is the project's highest accuracy using the dataset CMU-MMAC

Reference:

1. Text Reference:

a) https://en.wikipedia.org/wiki/Inertial_measurement_unit

b) http://kitchen.cs.cmu.edu/

c) https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-a-neural-network/

d) https://aws.amazon.com/what-is/neural-network/#:~:text=A%20neural%20network%20is%20a,that%20resembles%20the%20human%20brain

e) https://medium.com/@muhammed.muzammil2001/artificial-neural-networks-end-to-end-b4bd8a90141b

f) https://builtin.com/data-science/recurrent-neural-networks-and-lstm#:~:text=Recurrent%20neural%20networks%20(RNNs)%20are,problems%20that%20involve%20sequential%20data.

g) https://medium.com/@poudelsushmita878/recurrent-neural-network-rnn-architecture-explained-1d69560541ef

h) https://medium.com/@navarai/the-architecture-of-a-basic-rnn-eb5ffe7f571e

i) https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/

j) http://gnjatovic.info/misc/feedforward.neural.networks.pdf

k) Lecture Notes - http://gnjatovic.info/misc/naive.bayesian.classification.pdf

l) https://github.com/yanshengjia/ml-road/blob/master/resources/Hands%20On%20Machine%20Learning%20with%20Scikit%20Learn%20and%20TensorFlow.pdf
Book - Hands-on machine learning with Scikit-learn and TensorFlow. (Since the book was referred from a library, I'm sharing the GitHub reference link for the same book in PDF format.)

2. Image Reference:

a) https://upload.wikimedia.org/wikipedia/commons/thumb/3/35/Apollo_IMU_at_Draper_Hack_the_Moon_exhibit.agr.jpg/330px-Apollo_IMU_at_Draper_Hack_the_Moon_exhibit.agr.jpg

b) https://upload.wikimedia.org/wikipedia/commons/thumb/1/1e/Apollo_Inertial_Measurement_Unit.png/330px-Apollo_Inertial_Measurement_Unit.png

c) https://upload.wikimedia.org/wikipedia/commons/thumb/5/54/Flight_dynamics_with_text.png/330px-Flight_dynamics_with_text.png

d) https://upload.wikimedia.org/wikipedia/commons/thumb/8/8d/Apollo_IMU_at_Draper_Hack_the_Moon_exhibit_detail_1.agr.jpg/1024px-Apollo_IMU_at_Draper_Hack_the_Moon_exhibit_detail_1.agr.jpg

e) http://kitchen.cs.cmu.edu/MultiKitchen.jpg

f) https://freecontent.manning.com/wp-content/uploads/duerr_NNA_01.png

g) https://miro.medium.com/v2/resize:fit:1400/format:webp/1*EeRwi4cBjHyBuhmoR7IGYw.png

h) https://builtin.com/data-science/recurrent-neural-networks-and-lstm#:~:text=Recurrent%20neural%20networks%20(RNNs)%20are,problems%20that%20involve%20sequential%20data.

i) https://media.geeksforgeeks.org/wp-content/uploads/20231204131135/One-to-One-300.webp

j) https://media.geeksforgeeks.org/wp-content/uploads/20231204131304/One-to-Many-300.webp

k) https://media.geeksforgeeks.org/wp-content/uploads/20231204131355/Many-to-One-300.webp

l) https://media.geeksforgeeks.org/wp-content/uploads/20231204131436/Many-to-Many-300.webp

Dataset Reference:

http://kitchen.cs.cmu.edu/main.php