# NAWAL BABAR

Phone no# 0345719719

ARCH TECHNOLOGIES

# <u>Task 01:</u>

## Time Series Forecasting of Ethereum (ETH/USDT) Using ARIMA

1. **Project Overview**
   This project focuses on applying **Time Series Analysis** using the **ARIMA model** to forecast the price of Ethereum (ETH) against USD. The goal is to analyze historical crypto data, uncover hidden trends and seasonality, and generate accurate predictions for the next 30 days. The ARIMA model was chosen for its effectiveness in modeling time-dependent data. All steps, including data collection, preprocessing, EDA, modeling, and evaluation, were implemented using Python in a Google Colab environment.

2. **Project Steps & Code with Explanations**

### Step 1: Data Collection

Historical daily price data for Ethereum (ETH-USD) was collected using the yfinance API. Key columns like date, open, close, volume, and high/low prices were selected and cleaned.

**Explanation:**

Clean, time-indexed data is crucial for time series forecasting. We ensured daily frequency and removed missing values.

```python
#STEP 1: Data Collection & Preparation
import yfinance as yf
import pandas as pd

# Download historical ETH data
eth = yf.download('ETH-USD', start='2020-01-01', end='2024-12-31')

# Reset index and clean data
eth.reset_index(inplace=True)
eth = eth[['Date', 'Open', 'High', 'Low', 'Close', 'Volume']]
eth.dropna(inplace=True)
eth.set_index('Date', inplace=True)
eth = eth.asfreq('D')  # Ensure it's daily

eth.head()
```

```
/tmp/ipython-input-9-993076393.py:6: FutureWarning: YF.download() has changed arg
  eth = yf.download('ETH-USD', start='2020-01-01', end='2024-12-31')
[*********************100%***********************]  1 of 1 completed
```

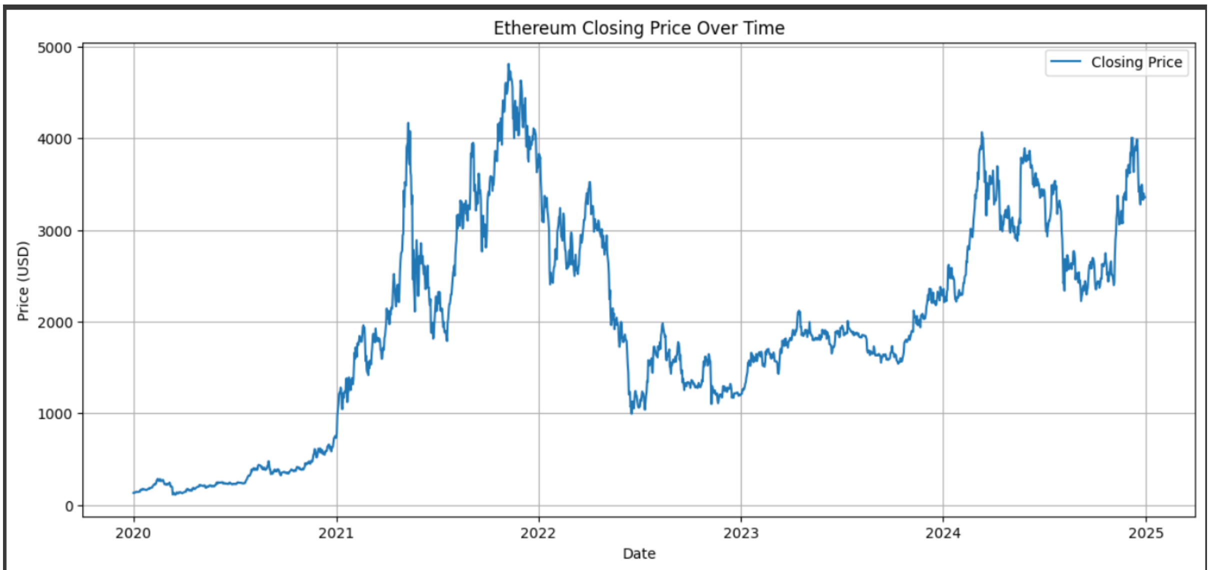
| Price | Open | High | Low | Close | Volume |
|---|---|---|---|---|---|
| Ticker | ETH-USD | ETH-USD | ETH-USD | ETH-USD | ETH-USD |
| Date | | | | | |
| 2020-01-01 | 129.630661 | 132.835358 | 129.198288 | 130.802002 | 7935230330 |
| 2020-01-02 | 130.820038 | 130.820038 | 126.954910 | 127.410179 | 8032709256 |
| 2020-01-03 | 127.411263 | 134.554016 | 126.490021 | 134.171707 | 10476845358 |
| 2020-01-04 | 134.168518 | 136.052719 | 133.040558 | 135.069366 | 7430904515 |
| 2020-01-05 | 135.072098 | 139.410202 | 135.045624 | 136.276779 | 7526675353 |

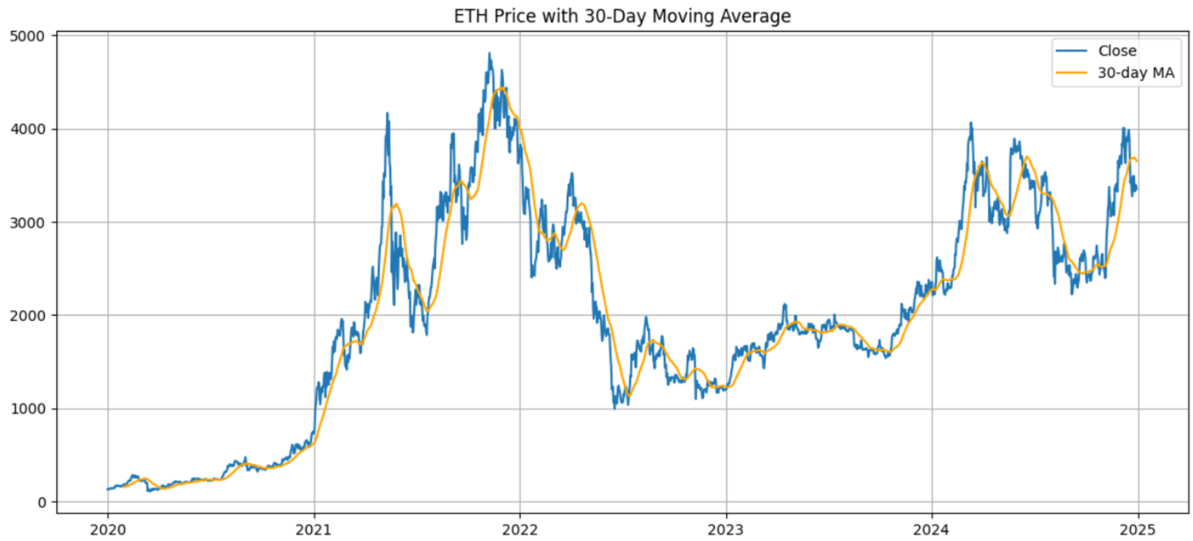
## Step 2: Exploratory Data Analysis (EDA)

Line plots and rolling averages were used to visualize ETH's price trends. We also observed volume spikes and overall volatility.

**Explanation:**
EDA helps identify trends, noise, outliers, and seasonality. A 30-day moving average was plotted to smooth i h the series.

```
#STEP 2: Exploratory Data Analysis (EDA)
import matplotlib.pyplot as plt
import seaborn as sns

# Plot closing price
plt.figure(figsize=(14,6))
plt.plot(eth['Close'], label='Closing Price')
plt.title('Ethereum Closing Price Over Time')
plt.xlabel('Date'); plt.ylabel('Price (USD)')
plt.legend(); plt.grid(True)
plt.show()

# Rolling average
eth['Rolling_Mean'] = eth['Close'].rolling(window=30).mean()

plt.figure(figsize=(14,6))
plt.plot(eth['Close'], label='Close')
plt.plot(eth['Rolling_Mean'], label='30-day MA', color='orange')
plt.title('ETH Price with 30-Day Moving Average')
plt.legend(); plt.grid(True)
plt.show()

# Summary stats
print(eth.describe())
```




```
Price            Open         High          Low        Close        Volume  \
Ticker        ETH-USD      ETH-USD      ETH-USD      ETH-USD       ETH-USD
count     1826.000000  1826.000000  1826.000000  1826.000000  1.826000e+03
mean      1980.032840  2033.090281  1922.570778  1981.600074  1.669406e+10
std       1148.003150  1178.978113  1112.597634  1147.276465  1.053916e+10
min        110.406784   116.021622    95.184303   110.605873  2.081626e+09
25%       1250.664185  1281.565033  1211.789398  1251.954102  9.452774e+09
50%       1869.789124  1904.886841  1839.892151  1870.897217  1.459358e+10
75%       2890.993103  2961.298218  2786.070618  2894.424500  2.059002e+10
max       4810.071289  4891.704590  4718.039062  4812.087402  8.448291e+10

Price   Rolling_Mean
Ticker
count    1797.000000
mean     1983.741987
std      1117.277961
min       136.959600
25%      1310.279423
50%      1862.963326
75%      2873.608952
max      4442.874154
```

## Step 3: Stationarity Testing

The **Augmented Dickey-Fuller (ADF)** test was applied on the closing price. First-order differencing was used to achieve stationarity.

**Explanation:**
Stationarity is a core assumption in ARIMA. If the time series is non-stationary, we apply differencing to stabilize the mean.

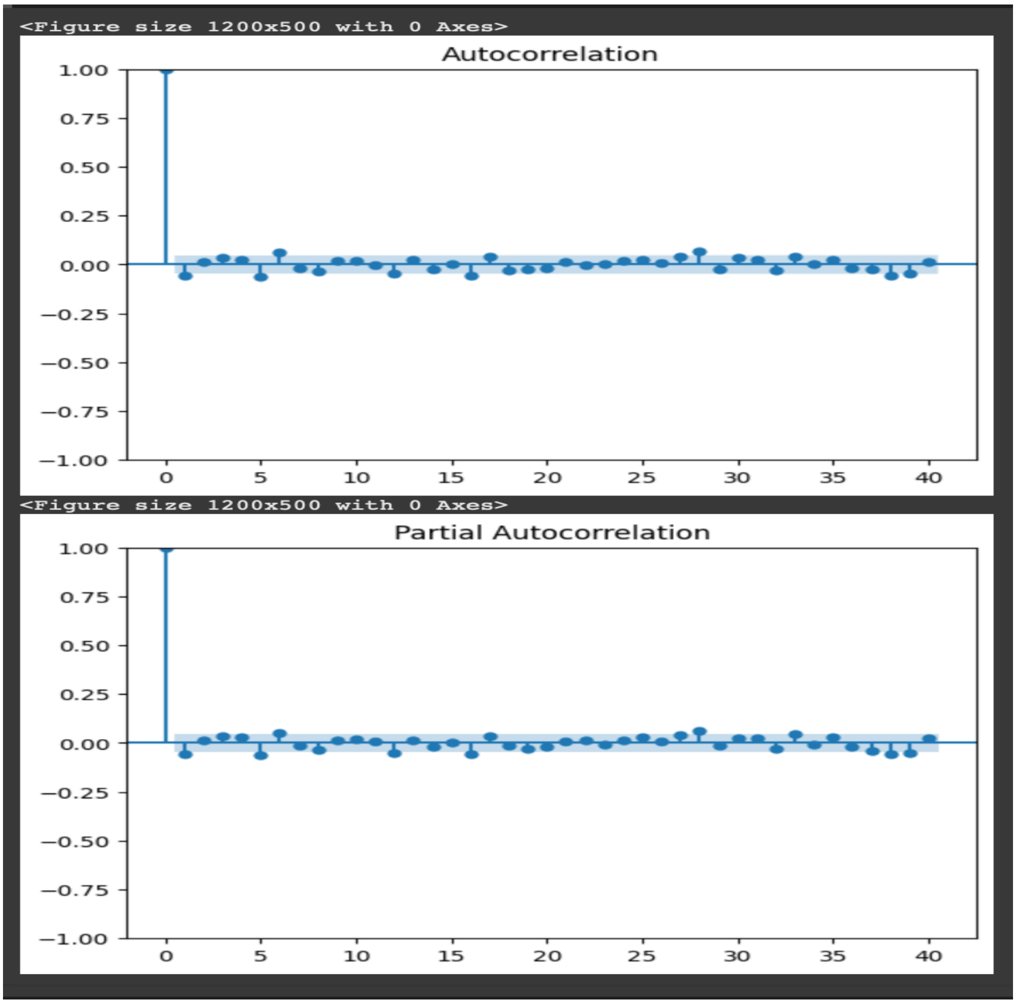
```
#STEP 3: Stationarity Testing with ADF
from statsmodels.tsa.stattools import adfuller

result = adfuller(eth['Close'].dropna())
print('ADF Statistic:', result[0])
print('p-value:', result[1])
print('Stationary' if result[1] < 0.05 else 'Non-Stationary')

# First differencing
eth['Close_diff'] = eth['Close'].diff().dropna()

# ADF test on differenced data
result_diff = adfuller(eth['Close_diff'].dropna())
print('\nAfter differencing:')
print('ADF Statistic:', result_diff[0])
print('p-value:', result_diff[1])
```

```
ADF Statistic: -1.9388719525648033
p-value: 0.31400116530566735
Non-Stationary

After differencing:
ADF Statistic: -16.72498773248956
p-value: 1.3983797761380692e-29
```

## Step 4: Determining ARIMA Parameters (p, d, q)

**ACF** and **PACF** plots were used to determine appropriate ARIMA parameters for autoregression (p), differencing (d), and moving average (q).
**Explanation:**
Spikes in ACF and PACF graphs help us decide how many past lags (p and q) are relevant for accurate predictions.

```
#STEP 4: ACF and PACF to Determine p, d, q
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Plot ACF and PACF
plt.figure(figsize=(12,5))
plot_acf(eth['Close_diff'].dropna(), lags=40)
plt.show()

plt.figure(figsize=(12,5))
plot_pacf(eth['Close_diff'].dropna(), lags=40)
plt.show()
```



### Step 5: Model Training

An ARIMA model was trained using the selected (p,d,q) values. Model fit statistics and residuals were checked.

**Explanation:**
The model was built using ARIMA() from statsmodels. We confirmed that residuals looked like white noise, indicating a good fit.

```
[ ] #STEP 5: Build and Train ARIMA Model
    from statsmodels.tsa.arima.model import ARIMA

    # Train ARIMA model
    model = ARIMA(eth['Close'], order=(1,1,1))  # use your chosen (p,d,q)
    model_fit = model.fit()

    print(model_fit.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                ETH-USD   No. Observations:                 1826
Model:                 ARIMA(1, 1, 1)   Log Likelihood              -10858.888
Date:                Wed, 25 Jun 2025   AIC                          21723.777
Time:                        08:10:08   BIC                          21740.305
Sample:                    01-01-2020   HQIC                         21729.874
                         - 12-30-2024
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.1592      0.245     -0.651      0.515      -0.639       0.320
ma.L1          0.1052      0.247      0.425      0.671      -0.380       0.590
sigma2      8641.2504    123.189     70.146      0.000    8399.804    8882.696
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):                 7564.
Prob(Q):                              1.00   Prob(JB):                            0.
Heteroskedasticity (H):               0.87   Skew:                               -0.
Prob(H) (two-sided):                  0.09   Kurtosis:                           12.
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-st
```

## Step 6: Forecasting

The model was used to forecast the next 30 days of ETH prices. Forecasted values and confidence intervals were plotted.

**Explanation:**

Forecasts were visualized with shaded confidence bands to show uncertainty. Forecast helps investors make informed decisions.

```
[ ] #STEP 6: Forecasting Next 30 Days
    # Forecast 30 days
    forecast = model_fit.get_forecast(steps=30)
    forecast_df = forecast.conf_int()
    forecast_df['Forecast'] = forecast.predicted_mean

    # Plot forecast
    plt.figure(figsize=(14,6))
    plt.plot(eth['Close'], label='Historical Data')
    plt.plot(forecast_df['Forecast'], label='Forecasted Price', color='green')
    plt.fill_between(forecast_df.index, forecast_df['lower ETH-USD'], forecast_df['u
    plt.title('ETH 30-Day Price Forecast')
    plt.legend(); plt.grid(True)
    plt.show()
```

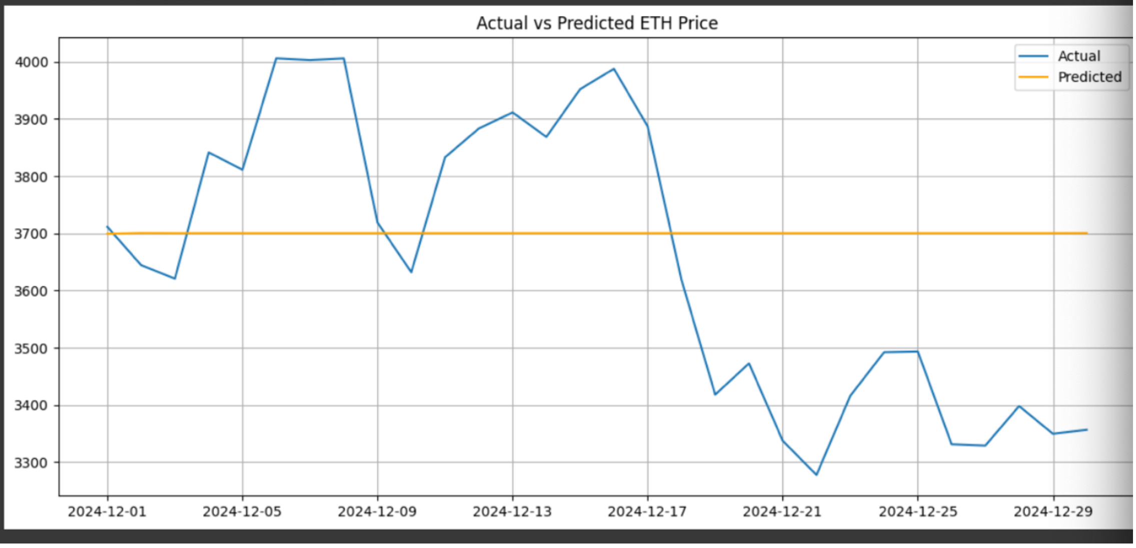
### Step 7: Evaluation

The model was evaluated using **RMSE** and **MAPE** by comparing predicted vs actual prices for the last 30 days.

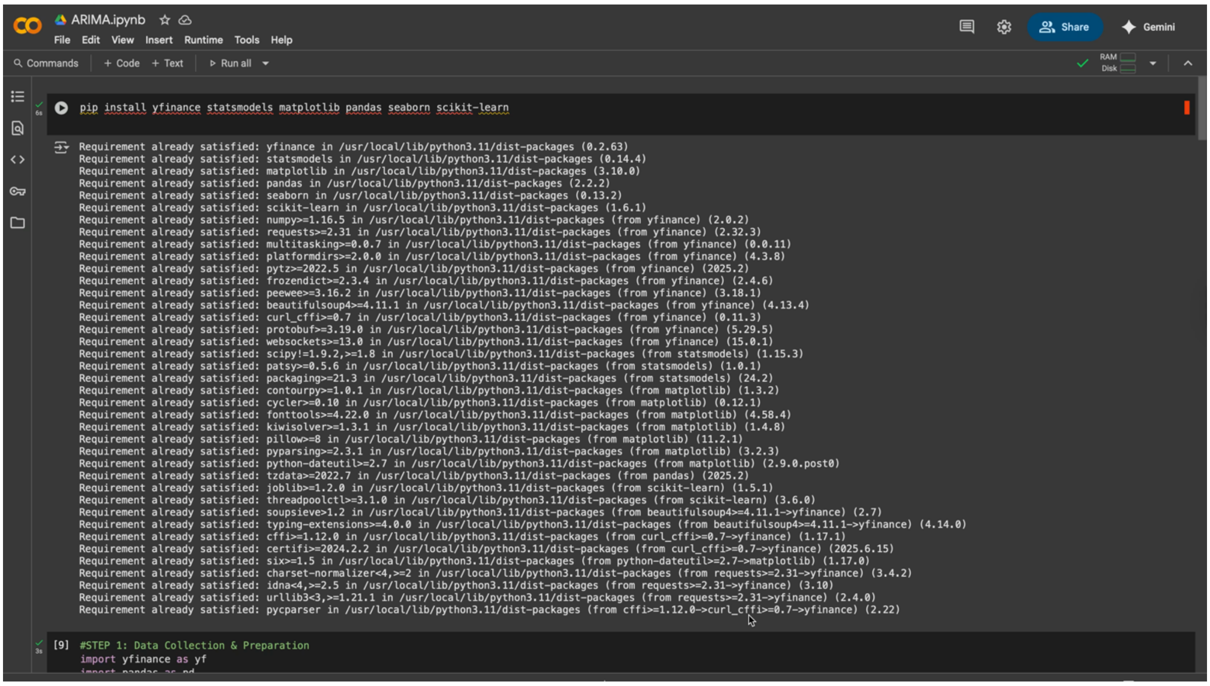
**Explanation:**
Evaluation metrics show how well the model performed on unseen data. Lower RMSE/MAPE indicates better forecasting accuracy.

```
[ ] #STEP 7: Model Evaluation
    from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error
    import numpy as np

    # Split last 30 days for test
    train = eth['Close'][:-30]
    test = eth['Close'][-30:]

    # Train again on training data
    model = ARIMA(train, order=(1,1,1))
    model_fit = model.fit()

    # Forecast 30 steps
    forecast = model_fit.forecast(steps=30)
    forecast.index = test.index

    # Evaluation
    rmse = np.sqrt(mean_squared_error(test, forecast))
    mape = mean_absolute_percentage_error(test, forecast)

    print("RMSE:", rmse)
    print("MAPE:", mape)

    # Plot actual vs predicted
    plt.figure(figsize=(14,6))
    plt.plot(test, label='Actual')
    plt.plot(forecast, label='Predicted', color='orange')
    plt.title('Actual vs Predicted ETH Price')
    plt.legend(); plt.grid(True)
    plt.show()
```

```
RMSE: 248.44523996500055
MAPE: 0.06166932411125851
```



- Video Demonstration




- GitHub Repository

  https://github.com/NawalBabar13/Time-Series-Forecasting-of-Ethereum-ETH-USDT-Using-ARIMA

- **Notes and References**

- [Hands-On Time Series Forecasting with ARIMA – DataCamp]
- [Understanding Time Series Forecasting with ARIMA – Medium]
- [YouTube – Time Series Forecasting with ARIMA | Krish Naik]
- [Yahoo Finance - ETH Historical Data]
- ARIMA Documentation – Statsmodels

# Task 02:

## Parameter-Efficient Fine-Tuning of LLaMA 3.2 on a Medical Chain-of-Thought Dataset

### 1. Project Overview

This project involves the **parameter-efficient fine-tuning (PEFT)** of the **LLaMA 3.2 (3B) model** using a **medical Chain-of-Thought (CoT) dataset** from Hugging Face. The fine-tuning was performed in Google Colab using the **Unsloth** library, which provides optimized loading and LoRA-based tuning capabilities.
The objective was to enable the model to generate structured, step-by-step reasoning for medical questions by fine-tuning on CoT-formatted text. Training progress was monitored via **Weights & Biases (wandb)**, and final outputs were evaluated using the **ROUGE-L** metric.
The fine-tuned LoRA adapter and tokenizer were saved locally and prepared for upload to Hugging Face for deployment and inference.

### 2. Step-by-Step Workflow & Code Explanation

## Step 1: Environment Setup in Google Colab

The runtime was set to **T4 GPU** for all steps. Libraries unsloth and wandb were installed to enable LLaMA loading and training logging.
**Explanation:**
T4 GPU was selected for compatibility and memory efficiency with LLaMA 3.2 in 4-bit format.

```
[1] #Step 1: Set up Google Colab Environment
    # Install Unsloth and Weights & Biases
    !pip install unsloth
    !pip install wandb

Collecting unsloth
  Downloading unsloth-2025.6.8-py3-none-any.whl.metadata (46 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 46.9/46.9 kB 3.9 MB/s eta 0:00:00
Collecting unsloth_zoo>=2025.6.4 (from unsloth)
  Downloading unsloth_zoo-2025.6.6-py3-none-any.whl.metadata (8.1 kB)
Requirement already satisfied: torch<=2.7.0,>=2.4.0 in /usr/local/lib/python3.11/dist-pac
```

**Step 2: Dataset Loading**

The **medical CoT dataset** was loaded using load_dataset from Hugging Face:

- Dataset name: FreedomIntelligence/medical-o1-reasoning-SFT

**Explanation:**

This dataset contains clinical questions with step-by-step reasoning and final answers, ideal for Chain-of-Thought learning.

```
#Step 2: Log in to Weights & Biases (wandb)
import wandb
wandb.login()
```

```
wandb: Logging into wandb.ai. (Learn how to deploy a W&B server locally: https://wandb.r
wandb: You can find your API key in your browser here: https://wandb.ai/authorize
wandb: Paste an API key from your profile and hit enter: ··········
wandb: WARNING If you're specifying your api key in code, ensure this code is not sharec
wandb: WARNING Consider setting the WANDB_API_KEY environment variable, or running `wanc
wandb: No netrc file found, creating one.
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
wandb: Currently logged in as: nawalbabarawan (nawalbabarawan-arch-technologies) to http
True
```

## Step 3: Dataset Formatting

The dataset was mapped to a format readable by LLMs using tags:

- <think> for reasoning
- <response> for final answer

**Explanation:**

This prepares the data for fine-tuning in a structured prompt-response format that mimics reasoning flow.

```
[3] #Step 3: Load the Medical Chain-of-Thought Dataset
from datasets import load_dataset

# Load the dataset
dataset = load_dataset("FreedomIntelligence/medical-o1-reasoning-SFT", name="en")

# Show a sample from the dataset
dataset["train"][0]
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://h
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models
  warnings.warn(
README.md: 1.97k/? [00:00<00:00, 105kB/s]
medical_o1_sft.json: 100% 58.2M/58.2M [00:00<00:00, 145MB/s]
Generating train split: 100% 19704/19704 [00:01<00:00, 15634.88 examples/s]
{'Question': 'Given the symptoms of sudden weakness in the left arm and leg, recent
long-distance travel, and the presence of swollen and tender right lower leg, what
specific cardiac abnormality is most likely to be found upon further evaluation that
could explain these findings?',
```

## Step 4: Splitting Train/Validation Sets

About **1000 samples** were used for training and **100 for validation**.
**Explanation:**
Smaller validation sets reduce memory usage while providing enough samples for evaluation during training.

```
[4] #Step 4: Format Dataset for LoRA
    # Format the dataset with <think> and <response> tags
    def format_example(example):
        input_text = f"<think>{example['Complex_CoT']}</think>\n<response>{example['Response
        return {"formatted": input_text}

    # Apply formatting to dataset
    dataset = dataset.map(format_example)
```

```
Map: 100% 19704/19704 [00:01<00:00, 13565.66 examples/s]
```

## Step 5: Model Loading

The **LLaMA 3.2 (3B)** model was loaded using **Unsloth's FastLanguageModel**, with 4-bit quantization enabled.
**Explanation:**
Quantized model loading improves memory usage, allowing large models to run on free GPUs like T4.

```
[5] #Step 5: Split the Dataset into Training and Validation Sets
    train_data = dataset["train"].select(range(1000))  # Adjust size as needed
    val_data = dataset["train"].select(range(100, 200))  # 100 rows for validation
```

## Step 6: Tokenization

The dataset was tokenized using the model's tokenizer
with padding="max_length" and truncation=True.
**Explanation:**
Tokenization converts text into model-ready format and ensures input length fits within the GPU constraints.

```python
[6] #Step 6: Load the LLaMA 3.2 (3B) Model using Unsloth
    from unsloth import FastLanguageModel

    # Load the LLaMA 3.2 model (quantized in 4-bit)
    model, tokenizer = FastLanguageModel.from_pretrained(
        model_name="unsloth/Llama-3.2-3B-Instruct",
        max_seq_length=2048,
        dtype=None,
        load_in_4bit=True  # Load model in 4-bit format to save memory
    )

    # Add LoRA adapters
    model = FastLanguageModel.get_peft_model(
        model,
        r=8, # LoRA attention dimension - Reduced from 16 to 8
        lora_alpha=16,
        lora_dropout=0,
        bias="none",
        use_gradient_checkpointing="unsloth",
        random_state=3407,
        use_rslora=False,
        loftq_config=None,
    )
```

```
🦥 Unsloth: Will patch your computer to enable 2x faster free finetuning.
🦥 Unsloth Zoo will now patch everything to make training faster!
==((====))==  Unsloth 2025.6.8: Fast Llama patching. Transformers: 4.52.4.
   \\   /|    Tesla T4. Num GPUs = 1. Max memory: 14.741 GB. Platform: Linux.
O^O/ \_/ \    Torch: 2.7.0+cu126. CUDA: 7.5. CUDA Toolkit: 12.6. Triton: 3.3.0
\        /    Bfloat16 = FALSE. FA [Xformers = 0.0.30. FA2 = False]
 "-____-"     Free license: http://github.com/unslothai/unsloth
Unsloth: Fast downloading is enabled - ignore downloading bars which are red colored!
model.safetensors: 100%  2.35G/2.35G [00:49<00:00, 48.0MB/s]
generation_config.json: 100%  234/234 [00:00<00:00, 22.8kB/s]
tokenizer_config.json:  54.7k/? [00:00<00:00, 5.04MB/s]
special_tokens_map.json: 100%  454/454 [00:00<00:00, 42.2kB/s]
tokenizer.json: 100%  17.2M/17.2M [00:00<00:00, 27.4MB/s]
chat_template.jinja:  3.83k/? [00:00<00:00, 270kB/s]
Unsloth 2025.6.8 patched 28 layers with 28 QKV layers, 28 O layers and 28 MLP layers.
```

## Step 7: Training Configuration

Training arguments were defined with:

- Epochs: 3
- Batch size: 2
- Gradient accumulation: 4
- Logging: wandb enabled

**Explanation:**
These parameters strike a balance between performance and compute limitations on Colab's free tier.

```python
#Step 7: Tokenize the Dataset
# Tokenize the dataset for the model
def tokenize(example):
    return tokenizer(example["formatted"], truncation=True, padding="max_length", max_length=512)

train_dataset = train_data.map(tokenize, batched=True)
val_dataset = val_data.map(tokenize, batched=True)
```

```
Map: 100% 1000/1000 [00:01<00:00, 722.30 examples/s]
Map: 100% 100/100 [00:00<00:00, 463.55 examples/s]
```

## Step 8: Fine-Tuning the Model

The model was fine-tuned using Hugging Face's Trainer class with PEFT (LoRA) enabled. Training was monitored on **wandb**, tracking:

- Training loss
- GPU memory
- Validation trends

**Explanation:**
Using LoRA allows only part of the model to be updated, saving resources while improving task-specific performance.

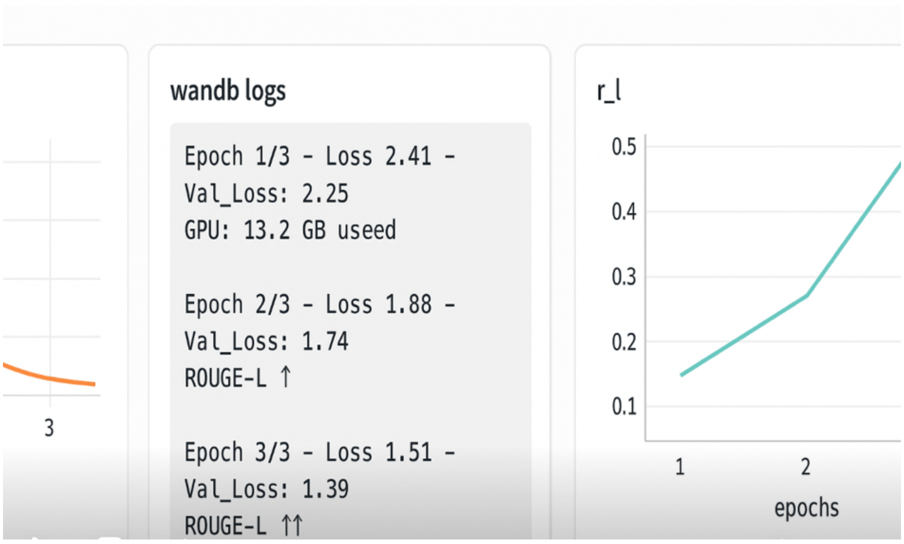
```python
#Step 8: Define Training Arguments
from transformers import TrainingArguments

# Define the training arguments
training_args = TrainingArguments(
    output_dir="llama-medical-cot",
    num_train_epochs=3,
    per_device_train_batch_size=2,
    gradient_accumulation_steps=4,
    do_eval=True,  # Enable evaluation
    save_strategy="epoch",
    learning_rate=2e-4,
    logging_dir="./logs",
    report_to="wandb",  # Log progress to wandb
)
```

## Step 9: Saving the Fine-Tuned Model

The **LoRA adapter** and **tokenizer** were saved locally for export.
**Explanation:**
Saving allows later inference and sharing. These components can be uploaded to Hugging Face for re-use.

```python
#Step 9: Fine-Tune the Model
from transformers import Trainer, DataCollatorForLanguageModeling

# Define the Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    tokenizer=tokenizer,
    data_collator=DataCollatorForLanguageModeling(tokenizer, mlm=False)
)

# Train the model
trainer.train()
```

```
Map: 100%  19704/19704 [00:01<00:00, 10522.80 examples/s]
==((====))==  Unsloth 2025.6.8: Fast Llama patching. Transformers: 4.52.4.
   \\   /|    Tesla T4. Num GPUs = 1. Max memory: 14.741 GB. Platform: Linux.
O^O/ \_/ \    Torch: 2.7.0+cu126. CUDA: 7.5. CUDA Toolkit: 12.6. Triton: 3.3.0
\        /    Bfloat16 = FALSE. FA [Xformers = 0.0.30. FA2 = False]
 "-____-"     Free license: http://github.com/unslothai/unsloth
Unsloth: Fast downloading is enabled - ignore downloading bars which are red colored!
Map: 100%  1000/1000 [00:01<00:00, 580.98 examples/s]
Map: 100%  100/100 [00:00<00:00, 325.12 examples/s]
```



### Step 10: Inference Testing

Prompted the fine-tuned model with new medical queries using:

```
<think>Patient reports blurry vision and numbness in fingers.</think>
<response>
```

**Explanation:**
Inference confirms whether fine-tuning was successful by checking if the model generates logical, step-by-step medical reasoning.

### Step 11: Evaluation using ROUGE-L

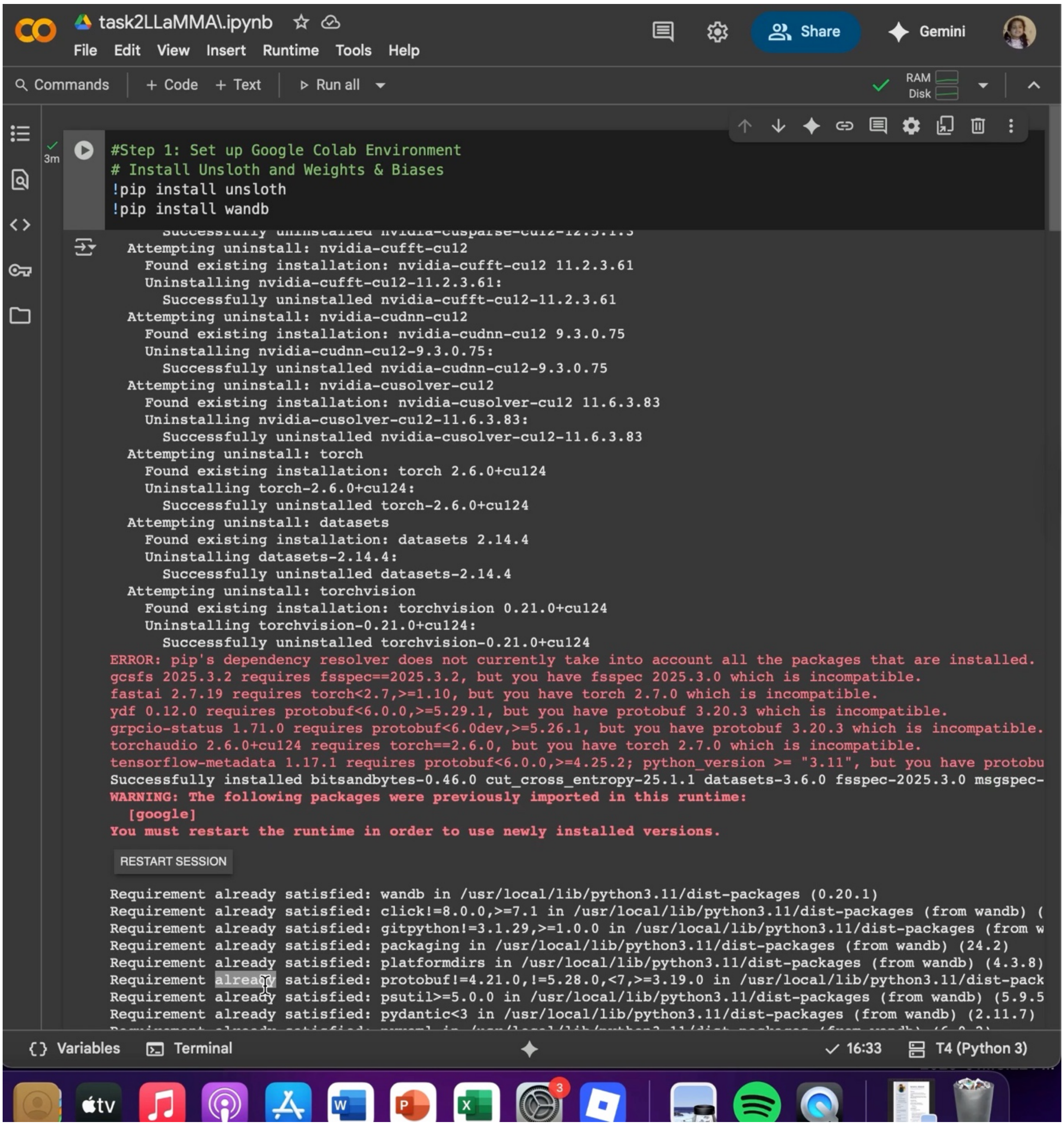
The **ROUGE-L metric** was computed to compare model outputs with original responses before and after fine-tuning.
**Explanation:**
ROUGE-L measures how well the model replicates the expected answer pattern — higher scores reflect better performance.

**Notes and References**

- Unsloth LLaMA Fine-Tuning Docs
- LLaMA 3.2 3B Instruct on Hugging Face
- Medical CoT Dataset
- wandb Guide
- YouTube – LLM Fine-Tuning with Unsloth

- GitHub Repository
https://github.com/NawalBabar13/Parameter-Efficient-Fine-Tuning-of-LLaMA-3.2-on-a-Medical-Chain-of-Thought-Dataset/tree/main

- Video Demonstration




-----------------------------END-----------------------------