Experiment -1

```
→ from transformers import pipeline
sentiment_analyzer = pipeline ("sentiment-analysis")
text = "I am feeling happy"
sentiment_result = sentiment_analyzer (text)
print ("Sentiment Analysis")
print (f "Input : {text}")
print (f "Sentiment : {sentiment_result [0] ['label']} (
                Score : {sentiment_result [0] ['score']
                :.2f}) \n")
```

Output

```
Sentiment Ay Analysis
Input : I am feeling happy
Sentiment : POSITIVE
```

```
→ translator_en_to_fr = pipeline ("translation_en_to_fr", model =
                "Helsinki-NLP/opus-mt-en-fr")

→ translator_output = translator_en_to_fr (text)
print ("Language Translation:")
print (f" Input : {text}")
print (f "Output : {translator_output} \n")
```

O/P verified [illegible]

Output

```
Language Translation:
Input : I am feeling happy
Output : [{'translation_text': 'Je me sens heureux.'}]
```

Experiment-2

```
→ import pytesseract
from PIL import Image
from transformers import pipeline
→ input = "/content/download (1).jpg"
input-image = Image.open(input)

→ import re
Text-output = pytesseract.image-to-string(input-image,
                    lang = 'eng'). strip()

Text-Output = re.sub(r'\s+', '', Text-output)

→ if not Text-output:
      print("No text found in the image")

else:
    print(f "Extracted Text: {Text-Output}")
    Sentiment-analyzer = pipeline("sentiment-analysis")
    Sentiment-result = sentiment-analyzer (Text-output)
                                                [0]

    print("\n Sentiment Analysis:")
    print(f "Text: {Text-output}")
    print(f "Sentiment: {Sentiment-result['label']}
       (Score: Sentiment-result['score']:.2f})\n")
```

Op

11/8/25

Output

```
Sentiment Analysis:
Text: Best Summer Ever.
Sentiment: POSITIVE
```

Experiment - 3

Algorithm

Variational Auto-encoder (VAE) for image Data

→ Import Required Libraries
  * import numpy, tensorflow, keras modules
  * import required layers from keras.

→ Define Sampling layer
  * Create a custom layer class Sampling that:
    1. Accepts mean and log-var as inputs
    2. Samples a latent vector z using reparameterization trick

$$z = \text{mean} + \exp(0.5 \times \log\text{-var}) \times \varepsilon$$

→ Build Encoder Network
  * Input: Image of Shape (28, 28, 1)
  * Apply:
    1. Conv2D with 64 filters, kernel size 3, stride 2, padding "Same"
    2. Conv2D with 128 filters, kernel size 3, stride 2, padding "Same"
    3. Flatten layer
    4. Dense layer with 16 neurons and ReLU activation
  * Output:
    * Mean Vector
    * Log variance vector
    * Latent vector z from class Sampling

→ Build Decoder Network

* Input: Latent vector of shape
* Apply:
  1. Dense layer
  2. Conv2D Transpose with 64 filters
  3. Conv2D Transpose with 128 filters
  4. Conv2D Transpose with 1 filters.

→ Defin VAE Model

* Train step:

  Encode → z, mean, log. value

  Decode → reconstruction

  Compute KL loss

  ~~KL = -0.5 {sum wi + text{log var} - 1 text~~

  $$\boxed{KL = -0.5 \times \Sigma(1 + \log var - mean^2 - e^{\log var})}$$

→ Training: Instantiate, Compile, fit on dataset, monitor losses.

O/p verified

[illegible] 10/8/25

Experiment -4

Algorithm

→ Import required libraries - Tensor flow, keras, Numpy
→ Load and preprocess the MNIST dataset; normalize images to range [0,1]
→ Prepare training datasets using batching and shuffle
→ Define a Generator model that takes random noise and ~~op~~ outputs fake image
→ Define a Discriminator model that takes an image & outputs a probability of it being real or fack.
→ Instantiate the generator models
→ Define the loss functions for both generator and discriminator
→ Define the optimizers for both models
→ Create a training loop.
    * for each batch of real images:
        a) Generate a batch of fake images using random noise passed to the generator
        b) Train the discriminator on both real & fake image
        c) Train the generator to fool the discriminator
→ Repeat training over several epochs
→ After each epoch, generate sample fake images to visualize training process.

Of ✓
19/8/25 [illegible]

Experiment - 5

Human Face Generation

Algorithm

→ Install all necessary deep learning and diffusion-based libraries

→ Import PyTorch and the PixArt diffusion pipeline from Hugging face diffusers

→ Specify the pretrained model ID to be used (PixArt-XL)

→ Load the diffusion model pipeline with Apropiate data type (eg. torch.float16)

→ Move the model to GPU (CUDA) for faster processing

→ Enable memory-efficient techniques like CPU-offload and VAE slicing on the pipeline

→ Write a natural-language text prompt that describes the kind of image to generate

→ Feed this prompt into the PixArt Pipeline's generate function

→ Allow the diffusion model to iteratively synthesize an image from the text prompt

→ Display the final generated image and optionally save it to a file.

9/10 [illegible] 20/8/25

Experiment - 6

Using Notebook and Comparing between Other LLMS

→ Open the NotbookLM.

→ According to your project choose atleast 10 reasearch papers and take some youtube videos ~~and~~

→ After getting all those pdf's and ~~vido~~ videos links add in the Notebook LM and summerize the all pdfs and video link and make a report of that.

→ And take on csv file add in Notebook LM ask summery of that.

→ Notebook LM can't do that while other LLMS can summerize that.

→ ~~So all this~~ After this process a make a report.

O/p verified

[illegible] 18/9/25

Experiment - 7

→ Initialize parameters: batch size, learning rate, laten dimension, epochs, classes, image size

→ Load MNIST dataset with normalization and create DataLoader

→ Define Generator (G): takes noise + label → output fake image

→ Define Discriminator (D): takes image + label → output real / fake probability

→ Set loss function (Binary Cross Entropy) and optimizers (Adam)

→ For each epoch:
  * Get real images and labels from dataset
  * Generate fake images using G.
  * Train D with real and fake images
  * Train G to fool D (make fake look real).

→ Periodically generate and display sample images (digits 0-9)

O/P verified

[illegible]

Experiment - 9

1) What is LLM?

* LLM stand for Large Language Model
* LLM's are neural Network (usally based on the Transform architecture) with billions of parameters that can predict and generate text sequences

2) Open Source LLMs ?

* Star Coder | Santa Coder (Hugging Face + Service Now)
* Bio GPT
* Vicuna
* Open Assistant

3) Any 4 popular LLM's Used in our Daily life

* GPT (Chat GPT)
* Gemini (Google Deep Mind)
* Claude
* LLaMA

4) Real-Time Applications

* Chatbots & Virtual Assistants
* Productivity Tools
* Search & Knowledge Retrivel
* Education & E-Learning

9

Exp

Algorithm

→ Download Miniconda from Anaconda.Com,

→ Open Anaconda Prompt and run "Conda env list" Command.

→ Create a name as "Conda create --name genai python=3.9"

→ Activate the genai. "Conda activate genai.

→ Download Ollama from ollama.com

→ Open a GitHub and a select a model and add that in anaconda prompt.

→ So your own LLM is ready.

→ To come out use Command "bye".

→ Add ~~from~~ Model files and run a code ~~for~~. and that can generate name for our LLMs.

Eg. Mario

O/p verified

[illegible] 15/9/25

Experiment - 10

Algorithm

→ Install Dependencies
 * ensure that the Gradio library is installed

→ Import Required Library
 * Import the gradio module

→ Define Function (Example 1: Greetting)
 * Create a function greet(name) that takes a string input and returns a greeting message

→ Build Interface for Greeting
 * Use gr.Interface() to connect the function with the
  * Input: Textbox for entering a name
  * Output: Label to display the greeting

→ Lanch Interface
 * Run the app locally using .lanch(share=False)

→ In same way for other functions as texts, audios, videos etc...

o/p verified

[illegible] 15/9/25

Github link verified for first ten experiments

[illegible] 8/10/2025

Experiment-11a

Deploying chat based Applications using Fast API

1) what is Fast API?

Fast API is a modern, fast (high-performance) web frame work for building API's with python 3.7+ based on standard python type hints. It's asynchronous & designed for quick development and automatic docs

2) Use of Uvicorn

Uvicorn is a ASGI server used to run fast API applications It handles asynchronous request efficiently and serves your app in production (or) development

3) Ngrok:-

Ngrok is a tool that creats secure tunnels from a public URL to your local machine, allowing you to expose your local server to the internet for testin (or) demos

4) Fast API vs flask vs REST API

Fast API:- Modern, async, type annotated, ~~and~~ auto generated docs, faster performance

Flask:- Older, synchronous, flexible but less autom features

Rest API:- An architectural style for designing ~~netw~~ networked applications; both Fast API and flask can build Rest API.

5) Applications of FAST API?

→ Building high performance API's and microservices
→ Real time apps using web sockets
→ Data science and ML-model serving
→ Authentication and back end services
→ Any synchronous web applications required speed & scalability.

Algorithm

→ Install Required libraries:- fastapi, unicorn, nest_asyncio.

→ Import modules:-
import FastAPI for creating the web API, Pydantic for data models, and ngrok for creating a public URL.

→ Kill any old ngrok processes to avoid conflicts.

→ Apply nest_asyncio to allow running FastAPI inside environments like google colab

→ Initialize a FastAPI app named "Fast API chat App (Colab-safe)"

O/P [illegible] 11/10/2025

→ Create a message class using Pydantic with fields sender and text.

Experiment - 11b

Algorithm

→ Install sentence-transforms and faiss-cpu for text embeddings and similarity search

→ Import SentenceTransformer from sentence-transformers and faiss for vector indexing.

→ Create a list of short text documents.

→ Load a pre-trained sentence transformer model and generate embeddings for each document

→ Initialize a FAISS index with the embedding dimension and add all document embeddings to it.

→ * Encode the user query into a vector

* Search the FAISS index for the top k most similar documents

* Return those retrived documents

→ Input a sample question

→ Use the retrival function to find and print the most relevant documents based on similarity.

O/P verified
[illegible] 11/10/2025

eleventh exp updated in github
[illegible] 18/10/2025

Experiment.12

Algorithm

→ Initialize:
Create Q table with all zeroes for state action pairs

→ Set Hyperparameters:
Learning rate (α), discount factor (γ), exploration rate (ϵ), & episodes

→ For each episode:-
* Reset environment → get initial state
* Repeat until goal reached:
  (a) Choose an action using ϵ-greedy policy
  (b) Perform action → observe next state & reward
  (c) Update Q values using:
  $Q(s,a) = Q(s,a) + \alpha[r + \gamma \max Q(s',a') - Q(s,a)]$
  (d) Set next state as current state

→ Decay ϵ over episodes to reduce exploration
→ After training, evaluate agent using learned Q-table
→ LLM integration
Use LLM to interpret policy decisions & recommend tuning for α, γ, of rewards

29/10/2025