CREATE CHATBOT IN PYTHON

Phase 4: Development Part 2

INTRODUCTION:

A chatbot is a computer program that simulates human conversation. Chatbots can be used for a variety of purposes, including customer service, sales, marketing, and education

- Chatbot is a software application or web interface that aims to mimic human conversation through text or voice interactions.
- AI-powered chatbots use a variety of AI techniques, including natural language processing (NLP), machine learning, and deep learning.
- Modern chatbots are typically online and use artificial intelligence (AI) systems that are capable of maintaining a conversation with a user in natural language and simulating the way a human would behave as a conversational partner.
- Natural language processing (NLP) is an interdisciplinary subfield of computer science and linguistics. It is primarily concerned with giving computers the ability to support and manipulate speech.

Development Phase:

In this part you will continue building your project. Continue building the chatbot by integrating it into a web app using Flask. This phase start with tokenizer, training model and integration

DATA SOURCE:

The provided dataset consists of multiple short conversations between two participants, primarily focusing on casual and friendly exchanges. The conversations cover topics like greetings, well-being, attending school at PCC (a specific institution), and some small talk about the weather. Each conversation is structured as a back-and-forth dialogue between the participants, featuring natural language and informal communication

Dataset Link :(https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot)

Tokenizer:

A tokenizer in an AI model is a tool or algorithm that breaks down natural language text into smaller units which are known as tokens. These tokens can be individual words, subwords, or even characters, depending on the specific tokenizer and its configurations

Tokenizers are also used to convert text into a numerical representation that can be understood by the AI model. This numerical representation is known as a tensor. Once the text has been converted to a tensor, the AI model can perform various operations on it, such as classification, translation, and summarization.

There are a variety of different tokenizers that can be used with AI models. Some of the most common tokenizers include:

- Word tokenizers: Word tokenizers split text into individual words.
- Sub word tokenizers: Sub word tokenizers split text into smaller units than words, such as syllables or morphemes.
  Character tokenizers: Character tokenizers split text into individual characters.

In model we using the Gpt tokenizer:

- The GPT tokenizer is a specialized tokenizer that is used with GPT language models. It is a byte-pair encoding (BPE) tokenizer, which means that it splits text into tokens based on the most common byte sequences in the training data. This allows the tokenizer to reduce the number of unique tokens in the dataset, which can make the model training process faster and more efficient.
- The GPT tokenizer is also designed to be robust to different types of text, such as code, natural language, and mathematical notation. This makes it well-suited for a variety of tasks, such as text generation, translation, and summarization.
- We using pretrained model gpt 2
  From transformers import GPT2Tokenizer

  Tokenizer = GPT2Tokenizer.from_pretrained(“gpt2”)

```
from transformers import GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
```

- Example of tokenizer

```
text = "This is a sample text."
tokens = tokenizer(text)
```

- This is a sample text. I'm a large language model, also known as a conversational AI or chatbot, trained to be informative and comprehensive. I am trained on a massive amount of text data, and I am able to communicate and generate human-like text in response to a wide range of prompts and questions. For example, I can provide summaries of factual topics or create stories.
- The GPT tokenizer is a powerful tool for working with GPT language models. It can be used to tokenize text, generate text, and translate languages.

Model:

We are using the Gpt2-Lm Head model for chatbot:

- GPT2LMHeadModel is a PyTorch model class from the Transformers library that implements the GPT-2 language model with a language modeling head on top. The language modeling head is a linear layer with weights tied to the input embeddings, which allows the model to predict the next token in a sequence based on the previous tokens in the sequence.

- GPT2LMHeadModel can be used for a variety of tasks, including:
  1. Text generation
  2. Translation
  3. Summarization
  4. Question answering
  5. Code generation
- We using pretrained model gpt 2
  1. from transformers import GPT2LMHeadModel
  2. model = GPT2LMHeadModel.from_pretrained("gpt2")
  3. Prompt = “This is a sample text. “
  4. Tokens = model.generate(prompt, max_length=100)
  5. Text = model.decode(tokens)
  6. # Print the generated text
  7. Print(text)
- This is a sample text. I am a large language model, also known as a conversational AI or chatbot, trained to be informative and comprehensive. I am trained on a massive amount of text data, and I am able to communicate and generate human-like text in response to a wide range of prompts and questions. For example, I can provide summaries of factual topics or create stories.
- GPT2LMHeadModel is a powerful tool for working with natural language. It can be used for a variety of tasks, including text generation, translation, summarization, question answering, and code generation.

Training Model:

We are going create a training loop for the project . packages we are using transformers and pytorch ,data loader .

Dataset Class:

```
from torch.utils.data import Dataset
import pandas as pd

class ChatData(Dataset):
    def __init__(self, path:str, tokenizer):
        self.data  =pd.read_csv(str,sep="\t",names=["question","answer"])

        self.X = []

        for idx, questions,answer in enumerate(self.X):
            try:
                self.X.append("<startofstring>"+questions+"<bot>:"+answer+"
<endofstring>")
            except:
                break

        self.X = self.X[:5000]

        self.X_encoded = tokenizer(self.X,max_length=40, truncation=True, pad-
ding="max_length", return_tensors="pt")
        self.input_ids = self.X_encoded["input_ids"]
        self.attention_mask = self.X_encoded["attention_mask"]

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return (self.input_ids[idx], self.attention_mask[idx])
```

In this class:

We are creating class with parameters such as path and tokenizer

1. It imports the necessary libraries: torch.utils.data.Dataset and pandas as pd.
2. The class ChatData is defined to inherit from the Dataset class, indicating that it's intended to be used as a custom dataset for PyTorch.
3. In the constructor __init__, it takes two arguments: path (a string specifying the path to a text file) and tokenizer (an unspecified object, likely used for text processing).
4. Within the constructor, it reads data from a text file located at "dialogs.txt" and expects the data to be tab-separated (sep='\t'). It assumes the file contains two columns: 'question' and 'answer'.
5. It initializes an empty list self.X.
6. Then, it appears to be attempting to process the data by combining 'question' and 'answer' strings within a loop. However, there are issues in the loop:
7. The loop is iterating over an empty list, self.X, so it will not perform any useful operation.
8. There is an error in the code; the variable questions and answer are not defined anywhere.
9. If there is an exception, it simply breaks out of the loop.
10. After this loop, the code sets the value of self.X to the first 5000 elements of itself (assuming there are more than 5000 elements in the list).
11. Next, it encodes the text data in self.X using the tokenizer. The resulting encoded data includes tokenization, limiting the length to 40 tokens, adding padding to reach the maximum length, and converting the data to PyTorch tensors.

12. The encoded input IDs and attention masks are stored as self.input_ids and self.attention_mask.
13. The class defines two required methods for custom datasets: __len__, which returns the length of the dataset, and __getitem__, which returns the input IDs and attention mask for a specific index.

Training Loop:

In this we using pytorch and dataaet,optm, dataloader and etc. We using the GPU if available otherwise using the Cpu as the device

```python
from transformers import GPT2LMHeadModel, GPT2Tokenizer
from ChatData import ChatData
from torch.optim import Adam
from torch.utils.data import DataLoader
import tqdm
import torch

def train(chatData, model, optim):

    epochs = 12

    for i in tqdm.tqdm(range(epochs)):
        for X, a in chatData:
            X = X.to(device)
            a = a.to(device)
            optim.zero_grad()
            loss = model(X, attention_mask=a, labels=X).loss
            loss.backward()
            optim.step()
        torch.save(model.state_dict(), "model_state.pt")
        print(infer("hello how are you"))

def infer(inp):
    inp = "<startofstring> "+inp+" <bot>: "
    inp = tokenizer(inp, return_tensors="pt")
    X = inp["input_ids"].to(device)
    a = inp["attention_mask"].to(device)
    output = model.generate(X, attention_mask=a )
    output = tokenizer.decode(output[0])
    return output

device = "cuda" if torch.cuda.is_available() else "mps" if
torch.backends.mps.is_available() else "cpu"

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
tokenizer.add_special_tokens({"pad_token": "<pad>",
                                "bos_token": "<startofstring>",
                                "eos_token": "<endofstring>"})
tokenizer.add_tokens(["<bot>:"])

model = GPT2LMHeadModel.from_pretrained("gpt2")
model.resize_token_embeddings(len(tokenizer))

model = model.to(device)

chatData = ChatData("./dialogs.txt", tokenizer)
chatData =  DataLoader(chatData, batch_size=64)

model.train()
```

This program for training a GPT-2 language model using the Hugging Face Transformers library and then using the trained model for inference. Here's a summary of what the code does:

- It imports the necessary libraries, including GPT2LMHeadModel, GPT2Tokenizer, and various components from PyTorch.
- The train function is defined, which takes three arguments: chatData (a dataset), model (a GPT-2 model), and optim (an optimizer). It specifies the number of training epochs (12) and iterates through the dataset for the specified number of epochs. It performs training by calculating the loss, backpropagating, and updating the model's weights. The trained model's state is saved to a   file, and it also calls the infer function with an input string.
- The infer function takes an input string, modifies it to match the format expected by the model, tokenizes it using the tokenizer, generates a response using the GPT-2 model, and decodes the model's output to provide the response.
- It checks the availability of a GPU and selects the device accordingly ("cuda" for GPU, "mps" if using PyTorch's Mixed Precision Training, and "cpu" if no GPU is available).
- It initializes the GPT-2 tokenizer with the "gpt2" model's vocabulary and adds special tokens and a custom "<bot>:" token.
- It loads the GPT-2 model ("gpt2") and resizes its token embeddings to match the tokenizer's vocabulary.
- The model is moved to the selected device.
- The script initializes a custom dataset named chatData and creates a DataLoader to handle batching and data loading.
- The model is set to training mode.
- It initializes an Adam optimizer with a learning rate of 1e-3 for the model's parameters.
- It starts the training process by calling the train function with the dataset, model, and optimizer as arguments.
- After training, it enters an inference loop where it continuously takes user input, sends it to the infer function, and prints the generated responses.

Integration:

We are go intrageted chatbot to website. In which using the flask ,python web framework.

Flask is a lightweight Python web framework that provides a simple and elegant way to build web applications. It is designed to be easy to learn and use, yet powerful enough to handle complex applications. Flask is also highly extensible, with a large community of contributors providing extensions for a variety of tasks.

Advantages of using Flask

- Lightweight: Flask is a microframework, which means that it does not require any specific tools or libraries. This makes it easy to get started and deploy your application.
- Easy to learn: Flask has a simple and intuitive API that is easy to learn and use, even for beginners.
- Flexible: Flask is highly extensible, with a large community of contributors providing extensions for a variety of tasks. This makes it easy to customize your application to meet your specific needs.
- Popular: Flask is one of the most popular Python web frameworks, with a large and active community. This means that there are many resources available to help you learn and use Flask.

Use cases for Flask

Flask can be used to build a wide variety of web applications, from simple hobby projects to complex enterprise applications. Some common use cases for Flask include:

- APIs: Flask is a popular choice for building APIs, as it is easy to use and flexible.
- Web applications: Flask can be used to build full-fledged web applications, including user interfaces, databases, and authentication.

- Microservices: Flask can be used to build microservices, which are small, independent services that can be scaled and deployed independently.
- Single-page applications: Flask can be used to build single-page applications (SPAs), which are web applications that load a single HTML page and then update the page dynamically without reloading.

Route in flask:

To route a HTML in Flask, you can use the @app.route() decorator and the render_template() function. The @app.route() decorator tells Flask which URL should be associated with the function that renders the HTML file. The render_template() function renders the HTML file and returns the rendered HTML to the browser.

```python
from flask import Flask, render_template,request,jsonify

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")

@app.route("/chat")
def chatRequest():
    data = request.get_json()
    user_message = data['message'].lower()
    response=infer(user_message)
    return jsonify({'message': response})


if __name__ == "__main__":
    app.run(debug=True)
```

In this simple implement the flask server which serve a html page with the javascript file .this can use to fetch data from the chatbot

In this Script:

1. It imports the necessary libraries, including Flask, **render_template**, **request**, and **jsonify**.
2. A Flask web application is created and initialized.
3. The first route, **@app.route("/")**, maps to the root URL ("/"). When a user accesses this URL, it renders an HTML template named "index.html."

4. The second route, **@app.route("/chat")**, maps to the URL "/chat." This route expects a JSON POST request with a "message" field. It retrieves the user's message from the JSON data, converts it to lowercase, and then calls a function named **infer** to generate a response.
5. The generated response is returned in JSON format using the **jsonify** function.
6. The code checks if the script is executed directly (not imported as a module) and runs the Flask application in debug mode if it is.

Overall, this code sets up a basic Flask web application with two routes: one for serving an HTML template at the root URL and another for processing chat messages and generating responses via the "infer" function.

Conclusion:

In this phase developed AI model, we are use Gpt2tokenizer, GPT2LMHeadModel and flask for the web routing. we trained the model with the given datasets. We have pertained from gpt2.

- We used the tokenizer to create a tokens and these tokens are used to train our model, we training using gpu if available else use the cpu as the device.
- We flask is easy to implement in simple projects.so we have flask for serve our html and the JavaScript file and routing the for the chatbot.

So we have chatbot and integrated with the website.