CREATE A CHATBOT USING PYTHON

PRESENTED BY

SIVASUBRAMANIAN TJ

1. GPT-2 TOKENIZER
2. DATA PREPROCESSING
3. CUSTOM PYTORCH DATASET
4. TRAINING ARGUMENTS
5. FINE-TUNING & ENSEMBLE METHOD
6. CONCLUSION

GPT-2 TOKENIZER

- Tokenization is a crucial step in natural language processing (NLP) that involves breaking down text into smaller units, or tokens, for further analysis and processing. In the provided code, we tokenize text using the GPT-2 tokenizer.

In [3]:
```
from transformers import GPT2LMHeadModel, GPT2Tokenizer, TextDataset, DataCollatorForLanguageModeling
from transformers import Trainer, TrainingArguments
import torch
import numpy as np

# Load the GPT-2 tokenizer and models
tokenizer1 = GPT2Tokenizer.from_pretrained("gpt2")
tokenizer1.pad_token = tokenizer1.eos_token  # Set padding token to eos_token
model1 = GPT2LMHeadModel.from_pretrained("gpt2")

tokenizer2 = GPT2Tokenizer.from_pretrained("gpt2")
tokenizer2.pad_token = tokenizer2.eos_token  # Set padding token to eos_token
model2 = GPT2LMHeadModel.from_pretrained("gpt2")
```

DATA PREPROCESSING

- This part of the code is responsible for reading and processing conversational data from a dataset file, splitting it into individual messages, and tokenizing the messages using the GPT-2 tokenizer.

```
# Define the file path to dataset
dataset_file = "dialogs.txt"  # Replace with the path to dataset file

# Read and preprocess the dataset
conversations = []
with open(dataset_file, "r", encoding="utf-8") as file:
    for line in file:
        conversation = line.strip().split("\t")
        conversations.append(conversation)

# Tokenize and format the data for fine-tuning
input_data = []
for conversation in conversations:
    tokens = []
    for message in conversation:
        tokens.extend(tokenizer1.encode(message, add_special_tokens=True))
    input_data.extend(tokens)
```

CUSTOM PYTORCH DATASET

- By converting our data into a PyTorch dataset, we make it compatible with PyTorch's data loaders, making it easier to iterate through, shuffle, and batch our data for training our GPT-2 models. The dataset is to be used in the training process, ensuring that our data is in a suitable format for model training.

```python
# Create a list of input_ids
input_ids = torch.tensor(input_data)

# Define a custom PyTorch dataset
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, input_ids):
        self.input_ids = input_ids

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

    def __getitem__(self, idx):
        return {"input_ids": self.input_ids[idx]}

# Create an instance of the custom dataset
dataset = CustomDataset(input_ids)
```

TRAINING ARGUMENTS

```
# Create a DataCollator for Language Modeling
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer1, mlm=False
)

# Training arguments
training_args = TrainingArguments(
    output_dir="./output",
    overwrite_output_dir=True,
    num_train_epochs=1,  # Training for just one epoch
    per_device_train_batch_size=2,  # Smaller batch size for less memory usage
    save_steps=500,  # Save checkpoints more frequently
    save_total_limit=1,  # Limit saved checkpoints to 1
    seed=42,
)
```

These training arguments define the training environment and conditions, such as the number of epochs, batch size, checkpoint saving frequency, and random seed. This information is crucial for configuring how the GPT-2 models are fine-tuned and how training progress is monitored and recorded.

FINE-TUNING & ENSEMBLE METHOD

```
# Initialize the Trainer for the first model
trainer1 = Trainer(
    model=model1,
    args=training_args,
    data_collator=data_collator,
    train_dataset=dataset,
)

# Fine-tune the first GPT-2 model
trainer1.train()

# Initialize the Trainer for the second model
trainer2 = Trainer(
    model=model2,
    args=training_args,
    data_collator=data_collator,
    train_dataset=dataset,
)

# Fine-tune the second GPT-2 model
trainer2.train()
```

We train two separate GPT-2 models (model1 and model2) with the same dataset and training configuration. The goal is to have two fine-tuned models that can be used in ensemble methods to generate text-based responses

```
# User interaction
while True:
    user_input = input("You: ")  # Get user input

    if user_input.lower() == "exit":
        print("Chatbot: Goodbye!")
        break

# Generate responses using the ensemble of models
input_ids = tokenizer1.encode(user_input, add_special_tokens=True, return_tensors="pt")

output1 = model1.generate(input_ids, max_length=50, num_return_sequences=3)
output2 = model2.generate(input_ids, max_length=50, num_return_sequences=3)

# Combine the predictions using ensemble (simple majority voting)
predictions = np.concatenate([output1, output2], axis=0)
final_output = np.median(predictions, axis=0)  # Using median for majority voting

for generated_output in final_output:
    response = tokenizer1.decode(generated_output, skip_special_tokens=True)
    print("Generated Response:", response)
```

- Ensemble methods are a powerful technique in machine learning that involve combining the predictions of multiple models to improve overall performance and robustness.

- In this code, we utilized an ensemble method to leverage the strengths of two separately fine-tuned GPT-2 language models (model1 and model2) for generating text-based responses.

CONCLUSION

- We have taken a pre-trained gpt-2 model and using ensemble technique we improved the models performance by combining the predictions of two models and manipulating the training parameters.

- We fine-tuned the model with the provided dataset and we used custom PyTorch dataset to make the dataset compatible for Pytorch data loaders and make it suitable for model training.