MODEL 1:

INTRODUCTION TO GENAI AND SIMPLE LLM INFERENCE ON CPU AND FINE-TUNING OF LLM MODEL TO CREATE A CUSTOM CHATBOT

1. INTRODUCTION

Overview of the Report

This report explores the development and fine-tuning of chatbots using two Jupyter notebooks, build_chatbot_on_spr.ipynb and single_node_finetuning_on_spr, on the Intel Developer Cloud. It covers the setup and configuration of the chatbot pipeline, the fine-tuning process, and practical applications of the developed chatbot. The goal is to demonstrate the capabilities of Intel's AI tools in creating efficient and optimized chatbot solutions.

Purpose and Objectives

The purpose of this report is to demonstrate the creation and fine-tuning of chatbots using Intel Developer Cloud's AI tools. The objectives are to:

1. Illustrate the chatbot development process.
2. Explore fine-tuning techniques on a single node.
3. Evaluate the effectiveness of Intel's AI extensions.
4. Identify practical applications of the developed chatbot.
5. Suggest areas for future improvements.

This report aims to provide concise and practical insights into using Intel's AI technologies for chatbot development.

2. INTEL DEVELOPER CLOUD

Introduction to Intel Developer Cloud

Intel Developer Cloud offers scalable, high-performance computing resources and advanced AI tools to help developers build, optimize, and deploy AI and machine learning applications efficiently.

Benefits and Features

Intel Developer Cloud provides scalability, high performance, and access to advanced AI tools, enabling efficient development and deployment of AI and machine learning applications.

3. INTEL EXTENSION FOR TRANSFORMERS

Overview of Intel Extension for Transformers

The Intel Extension for Transformers enhances transformer models with optimizations for performance and efficiency, including features like mixed precision training and inference.

Key Features and Capabilities

The Intel Extension for Transformers enhances performance with optimized algorithms and supports mixed precision, improving efficiency in AI model processing.

4. BUILDING A CHATBOT

Overview

Chatbots are AI-powered applications designed to simulate human conversation through text or voice. They play a crucial role in customer service, information retrieval, and automation, enhancing user engagement and operational efficiency.

Pipeline Configuration

PipelineConfig defines the configuration settings for building and optimizing AI models. MixedPrecisionConfig within it enhances performance by utilizing lower precision arithmetic, optimizing both speed and efficiency in model computations.

Building the Chatbot

Using build_chatbot in Intel Developer Cloud involves:

- **Importing Libraries**: Include necessary modules.
- **Configuring Pipeline**: Define settings like PipelineConfig.
- **Building the Chatbot**: Use build_chatbot to create and configure the chatbot instance.

```
from intel_extension_for_transformers.neural_chat import build_chatbot, PipelineConfig

from intel_extension_for_transformers.transformers import MixedPrecisionConfig

config = PipelineConfig(optimization_config=MixedPrecisionConfig())
```

```
chatbot = build_chatbot(config)

response = chatbot.predict(query="what is intel")

print(response)
```

Chatbot Prediction

Use the `predict` method of the chatbot instance.

Example:

```
response = chatbot.predict(query="example query")

print(response)
```

5. SINGLE-NODE FINETUNING

Overview

Fine-tuning optimizes pre-trained models for specific tasks like NLP and chatbots. It enhances model performance by adapting it to new data or tasks efficiently.

Setting Up the Environment

- **Install required libraries and dependencies.**
- **Configure hardware resources for fine-tuning tasks.**
- **Ensure compatibility with Intel's optimized tools for efficient processing.**

Fine-Tuning Process

- **Dataset Preparation**: Preparing dataset for training.
- **Model Configuration**: Setting up model architecture and parameters.
- **Training**: Executing the fine-tuning process on a single node.
- **Evaluation**: Assessing the model's performance with validation metrics.

Code

Fine-tune the model on Alpaca-format dataset to conduct text generation:

```
from transformers import TrainingArguments

from intel_extension_for_transformers.neural_chat.config import (

    ModelArguments,

    DataArguments,

    FinetuningArguments,
```

```
    TextGenerationFinetuningConfig,
)
from intel_extension_for_transformers.neural_chat.chatbot import finetune_model
model_args = ModelArguments(model_name_or_path="meta-llama/Llama-2-7b-chat-hf")
data_args = DataArguments(train_file="alpaca_data.json", validation_split_percentage=1)
training_args = TrainingArguments(
    output_dir='./tmp',
    do_train=True,
    do_eval=True,
    num_train_epochs=3,
    overwrite_output_dir=True,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    gradient_accumulation_steps=2,
    save_strategy="no",
    log_level="info",
    save_total_limit=2,
    bf16=True,
)
finetune_args = FinetuningArguments()
finetune_cfg = TextGenerationFinetuningConfig(
        model_args=model_args,
        data_args=data_args,
        training_args=training_args,
```

```
            finetune_args=finetune_args,
        )
finetune_model(finetune_cfg)
```

Fine-tune the model on the summarization task:

```
from transformers import TrainingArguments
from intel_extension_for_transformers.neural_chat.config import (
    ModelArguments,
    DataArguments,
    FinetuningArguments,
    TextGenerationFinetuningConfig,
)
from intel_extension_for_transformers.neural_chat.chatbot import finetune_model


model_args = ModelArguments(model_name_or_path="meta-llama/Llama-2-7b-chat-hf")
data_args = DataArguments(dataset_name="cnn_dailymail", dataset_config_name="3.0.0")
training_args = TrainingArguments(
    output_dir='./tmp',
    do_train=True,
    do_eval=True,
    num_train_epochs=3,
    overwrite_output_dir=True,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    gradient_accumulation_steps=2,
```

```
    save_strategy="no",
    log_level="info",
    save_total_limit=2,
    bf16=True
)
finetune_args = FinetuningArguments(task='summarization')
finetune_cfg = TextGenerationFinetuningConfig(
        model_args=model_args,
        data_args=data_args,
        training_args=training_args,
        finetune_args=finetune_args,
    )
finetune_model(finetune_cfg)
```

Results and Analysis

Evaluate using standard metrics like accuracy or F1-score. Analyze performance based on speed and resource utilization. Key observations highlight improvements or areas for optimization in model performance.

6. PRACTICAL APPLICATIONS

Use Cases of the Developed Chatbot

Real-world Applications: Enhance customer support, automate FAQ handling, and streamline information retrieval across industries like retail, healthcare, and finance.

Potential Integrations and Deployments

Integration Strategies: Integrate with existing CRM systems, websites, and mobile apps for seamless user interaction. **Deployment**: Deploy on cloud platforms or edge devices for scalable and efficient performance.

7. CONCLUSION

Summary of Key Points

The report demonstrated the creation and fine-tuning of chatbots using Intel Developer Cloud's AI tools. Key aspects covered include chatbot development, fine-tuning techniques, and practical applications.

Future Work and Improvements

Future research could focus on enhancing chatbot capabilities through more complex AI models. Improvements in scalability and integration with diverse platforms could further enhance usability and performance. Suggest potential areas for further research and development.

MODEL 2:

CUSTOMER SERVICE CHATBOT

What sets our project apart is the innovative approach to chatbot development. Not only have we created a customer-facing chatbot that provides answers and solutions, but we've also designed a parallel chatbot that gathers organizational information and data from the business itself. This dual-chatbot system enables the customer-facing chatbot to provide highly accurate and personalized responses to customer queries, leveraging the rich data and insights gathered from the organization. This unique approach empowers small businesses to deliver exceptional customer service, setting them apart in a competitive market.

Data Retrieval About the Organization

1. Libraries Installation and Imports

To get started with our project, it's essential to install the necessary libraries. The following commands install the required packages:

```
pip install transformers gradio
```

- **Transformers**: This library by Hugging Face provides pre-trained models and tools to work with them, which is crucial for our chatbot's natural language processing (NLP) capabilities.
- **Gradio**: This library helps in creating an interactive user interface for our chatbot, making it easier for users to interact with and provide inputs.

In the script, these libraries are imported as follows:

```
import gradio as gr

from transformers import pipeline
```

2. Global Variables and Questions

Global variables are used to store the user's responses and the questions the chatbot will ask. This helps in maintaining the state throughout the interaction. The initialization looks like this:

```
responses = []

questions = [

    "What is the name of your organization?",

    "What services do you offer?",

    "Who are your target customers?",

    "What is your contact email?",

    "What are your business hours?"

]
```

3. Chatbot Response Function

The chatbot response function handles user inputs and navigates through the questions. Here's the function definition:

```
def chatbot_response(user_input, state):

    state['responses'].append(user_input)

    state['index'] += 1

    if state['index'] < len(state['questions']):

        return state['questions'][state['index']], state

    else:

        return "Thank you for your responses!", state
```

This function:

- Appends the user's input to the `responses` list.
- Increments the question index.
- Checks if there are more questions to ask. If so, it returns the next question; otherwise, it thanks the user.

4. Navigation Functions

Several functions handle navigation and data submission:

Previous Question

```
def previous_question(state):
    state['index'] = max(0, state['index'] - 1)
    return state['questions'][state['index']], state
```

This function decrements the question index to allow the user to go back to the previous question.

Submit Data

```
def submit_data(state):
    save_responses_to_file(state['responses'])
    return "Your data has been submitted successfully!", state
```

This function saves the responses and confirms submission.

Save Responses to File

```
def save_responses_to_file(data):
    with open('responses.txt', 'w') as file:
        for response in data:
            file.write(response + '\n')
```

This function writes the collected responses to a text file.

Reset Chatbot

```
def reset_chatbot():
    return {"responses": [], "index": 0}
```

This function resets the chatbot state for a fresh start.

5. Gradio Interface Setup

Layout

The layout is designed using Gradio to facilitate user interaction:

```
chatbot_output = gr.Textbox(label="Chatbot", interactive=False)
user_input = gr.Textbox(label="Your Response")
```

next_button = gr.Button("Next")

prev_button = gr.Button("Previous")

submit_data_button = gr.Button("Submit Data")

restart_button = gr.Button("Restart")

Button Click Functions

Button click functions handle the navigation and data submission:

next_button.click(chatbot_response, inputs=[user_input, state], outputs=[chatbot_output, state])

prev_button.click(previous_question, inputs=state, outputs=[chatbot_output, state])

submit_data_button.click(submit_data, inputs=state, outputs=[chatbot_output, state])

restart_button.click(reset_chatbot, outputs=state)

Initial Load

The initial state is loaded as follows:

demo.load(lambda: ("Welcome! What is the name of your organization?", {"responses": [], "index": 0}), outputs=[chatbot_output, state])

6. Launching the App

The Gradio app is launched with:

demo.launch()

Explanation

Starting the Chatbot

When you first open the chatbot, it greets you and asks for the name of your organization. It's designed to help you set up a customized customer service bot by gathering some essential information about your business.

Answering Questions

The chatbot will ask you a series of questions one by one. You just type your response in the provided textbox and click "Next" to move to the next question.

Navigating Through Questions

If you make a mistake or want to change an answer, you can click "Previous" to go back to the last question and update your response. You can keep moving back and forth between questions using the "Next" and "Previous" buttons until you're satisfied with all your answers.

Submitting Your Responses

Once you reach the last question and provide your answer, a "Submit Data" button will appear. Clicking this button will save all your responses to a file and show a message thanking you for completing the questionnaire.

Restarting the Process

If you need to start over for any reason, you can click the "Restart" button to reset the chatbot to its initial state and begin answering the questions again.

Completion

After submission, the chatbot saves your responses, indicating that the customization process for your customer service bot is complete.

Gradio Interface to Read and Display Responses File Content

Function to Read File

To display the content of the saved responses, we define a function to read the text file:

```
def read_file():
    try:
        with open('responses.txt', 'r') as file:
            content = file.read()
            line_count = content.count('\n') + 1
        return content, line_count
    except FileNotFoundError:
        return "File not found.", 0
```

This function:

- Opens the file and reads its content.

- Counts the number of lines.
- Returns the content and line count, or an error message if the file is not found.

Creating the Interface

The interface for displaying the file content is set up using Gradio:

```
python
Copy code
with gr.Blocks() as demo:
    with gr.Column():
        file_content = gr.Textbox(label="File Content", interactive=False, lines=20)
        line_count = gr.Textbox(label="Line Count", interactive=False, lines=1)

    demo.load(read_file, outputs=[file_content, line_count])
    demo.launch()
```

Launching the App

The app is launched using:

```
python
Copy code
demo.launch()
```

This interface allows users to see the content of the responses file and the number of lines it contains.

PDF Upload and Text Extraction Interface

Function to Convert PDF to Text

To process PDFs, we use the pdfminer library for text extraction:

```
python
Copy code
from pdfminer.high_level import extract_text

def pdf_to_text(pdf_file):
    try:
        text = extract_text(pdf_file)
        lines = text.split('\n')
        line_count = len(lines)
        return text, line_count
    except Exception as e:
        return str(e), 0
```

This function:

- Extracts text from the uploaded PDF.
- Counts the number of lines.
- Returns the extracted text and line count, or an error message if extraction fails.

Function to Handle File Upload and Conversion

To handle file uploads and convert PDFs to text:

```
python
Copy code
def handle_file_upload(pdf_file):
    if pdf_file is None:
        return "Please upload a PDF file.", "", "", False
    text, line_count = pdf_to_text(pdf_file)
    with open('pdf_text.txt', 'w') as file:
        file.write(text)
    response_message = f"File '{pdf_file.name}' uploaded and converted successfully.\nNumber of lines:
{line_count}\nText saved to 'pdf_text.txt'."
    return response_message, 'pdf_text.txt', text, True
```

This function:

- Checks for file upload.
- Converts the PDF to text.
- Saves the extracted text to a file.
- Returns the response message, saved file name, text content, and a success flag.

Gradio Interface Setup

Title and Description

```
python
Copy code
title = "PDF UPLOAD"
description = "Upload your license terms, conditions policies, and any other related information to
finalize the questionnaire."
```

Creating the Interface

The interface for PDF upload and text extraction is created as follows:

```
python
Copy code
gr.Interface(
    fn=handle_file_upload,
    inputs="file",
    outputs=["text", "text", "text"],
    title=title,
    description=description
).launch()
```

Launching the App

The app is launched using:

```
python
Copy code
demo.launch()
```

This Gradio interface enables users to upload a PDF file, converts the PDF content to text, counts the number of lines, saves the text to a file, and displays the results.

File Content Appender

Function Definition

To append content from one file to another:

```
python
Copy code
def append_file(file1_path, file2_path):
    try:
        with open(file2_path, 'r', encoding='utf-8', errors='replace') as file:
            content = file.read()
        return content
    except FileNotFoundError:
        return "File not found."
    except Exception as e:
        return f"An error occurred: {e}"
```

This function:

- Reads the content with errors='replace' to handle problematic characters.
- Returns the content or an error message.

Example Usage

Reading the content of responses.txt:

```
python
Copy code
file_path = 'responses.txt'
content = read_file(file_path)
print(content)
```

This will print the content of the responses file or an error message if something goes wrong.

AI-Powered Customer Service Chatbot with Contextual Answering

This code sets up an AI-powered chatbot using a pre-trained BERT model for question answering. The chatbot reads context from a text file and answers user queries based on that context. If it cannot find an answer, it provides customer care contact numbers extracted from the text. Here's a detailed explanation:

1. Installation and Imports

The required libraries are installed using pip:

sh
Copy code
pip install transformers torch

The necessary modules are imported as follows:

python
Copy code
from transformers import AutoTokenizer, AutoModelForQuestionAnswering
import torch
import re

2. Model and Tokenizer Setup

The model and tokenizer used for question answering are loaded:

python
Copy code
model_name = "bert-large-uncased-whole-word-masking-finetuned-squad"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForQuestionAnswering.from_pretrained(model_name)

3. Read Text File

A function to read the content of a text file is defined:

python
Copy code
def read_text_file(file_path):
    with open(file_path, 'r', encoding='latin-1') as file:
        return file.read()

4. Extract Customer Care Numbers

A function to extract customer care numbers using regular expressions is defined:

python
Copy code
def extract_customer_care_numbers(context):
    pattern = r"\b[A-Z][a-zA-Z]+\b.*?(\d{3}[-\s]?\d{3}[-\s]?\d{4})"
    matches = re.findall(pattern, context)
    return matches

5. Generate Response

The main function to generate responses based on the context is defined:

python
Copy code
def generate_response(context, question):
    inputs = tokenizer.encode_plus(question, context, add_special_tokens=True, return_tensors="pt",
truncation=True)
    input_ids = inputs["input_ids"].tolist()[0]
    text_tokens = tokenizer.convert_ids_to_tokens(input_ids)
    output = model(**inputs)

```
    answer_start_scores = output.start_logits
    answer_end_scores = output.end_logits
    answer_start = torch.argmax(answer_start_scores)
    answer_end = torch.argmax(answer_end_scores) + 1
    answer = tokenizer.convert_tokens_to_string(tokenizer.convert_ids_to_tokens(input_ids[answer_start:answer_end]))
    if answer.strip() == "":
        customer_care_numbers = extract_customer_care_numbers(context)
        return f"Sorry, I couldn't find an answer. Please contact customer care: {customer_care_numbers}"
    else:
        return answer
```

6. Load Text File Content

The content of the responses file is loaded:

```
python
Copy code
context = read_text_file('responses.txt')
```

7. Chatbot Loop

A function to run an interactive chatbot loop is defined:

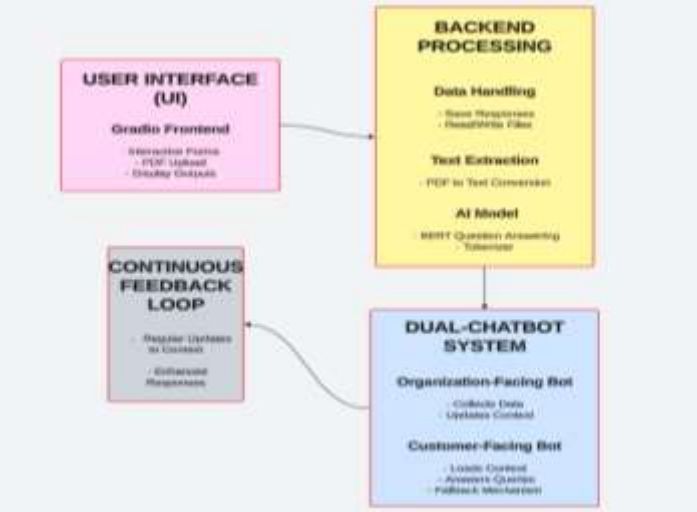
```
python
Copy code
def chatbot():
    print("Hello! How can I help you today? (Type 'exit' to stop)")
    while True:
        question = input("\nYour question: ")
        if question.lower() == 'exit':
            break
        response = generate_response(context, question)
        print("\nAnswer:", response)
```

8. Run the Chatbot

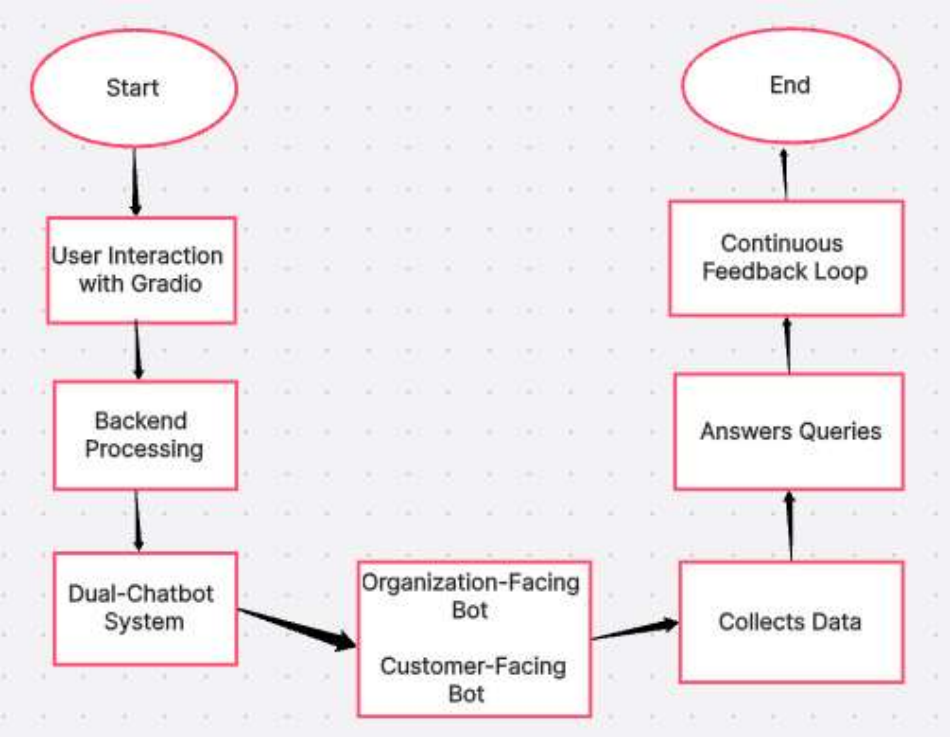
The chatbot function is called to start the chatbot:

```
python
Copy code
chatbot()
```

Architecture Diagram



Flow Chart



Summary

This script sets up a chatbot that leverages a pre-trained BERT model for question answering. It reads a text file to gather context, answers user queries based on that context, and provides customer care numbers if the answer is not found. The chatbot interacts with users in a loop, offering an engaging and informative experience.

Conclusion

The project showcases an innovative approach to chatbot development through the creation of a dual-chatbot system. The integration of a customer-

facing chatbot, capable of providing solutions and answers to customer queries, and a parallel chatbot, designed to gather organizational data, enables a highly personalized and accurate customer service experience. This system leverages the powerful capabilities of pre-trained BERT models for natural language processing and employs Gradio to create an intuitive user interface for interaction.

By incorporating advanced NLP techniques and ensuring robust data handling through comprehensive file management functions, the project exemplifies how small businesses can utilize AI to enhance customer engagement. The detailed explanation of the chatbot's functions, from question handling to PDF processing and response generation, provides a clear understanding of its implementation and capabilities.

This project not only demonstrates the potential of AI-powered solutions in customer service but also offers a practical and scalable approach for small businesses to differentiate themselves in a competitive market.