AI Chatbot Fine-Tuning Using WhatsApp Data

A comprehensive workflow for building a custom chatbot using WhatsApp conversations, instruction fine-tuning, and open-source models.

WhatsApp Data

Fine-Tuning

Custom Chatbot

Agenda

Overview of Chatbot Fine-Tuning Workflow
End-to-end process from data collection to deployment

Data Preparation & Preprocessing
Converting WhatsApp chats to trainable format

Model Loading and Training
Fine-tuning with LoRA and parameter-efficient techniques

Chat Interface and Usage
Deploying and interacting with your personalized chatbot

Workflow Overview

1. Setup & Installation
2. Data Acquisition
3. Data Transformation (Alpaca format)
4. Optional WhatsApp Parser
5. Cleaning & Preprocessing
6. Model Training Libraries
7. Model Loading & LoRA Adapter
8. Dataset Preparation
9. Training & Callbacks
10. Deploying and Interacting

This end-to-end workflow transforms raw WhatsApp conversations into a personalized AI chatbot using parameter-efficient fine-tuning techniques.

Step 1: Setup & Installation

First, install the required Python packages for your chatbot project:

whatstk
Python toolkit for WhatsApp chat analysis and parsing

einops
Simplified tensor operations for deep learning models

Code Example:

```
!pip install whatstk einops
```

Additional libraries for model training (bitsandbytes, transformers, peft, etc.) will be installed in later steps.

Step 2: Data Acquisition

Upload your dialog dataset in tab-separated format for training:

dialogs.txt Format:

instruction	response
How are you today?	I'm doing well, thanks for asking!
What's the weather like?	I don't have real-time weather data, but I can help you find that information.

Two-column format with user messages and corresponding chatbot replies

Code Example:

```
from google.colab import files
uploaded = files.upload()
df = pd.read_csv("dialogs.txt", sep="\t", header=None,
                 names=["instruction", "response"])
```

For best results, ensure your dialog dataset is clean, relevant, and contains diverse examples of the conversation style you want your chatbot to learn.

Step 3: Format Data for Alpaca

Transform your chat data pairs into Alpaca format, which is a standard for instruction fine-tuning language models:

```
instruction: "How are you doing today?"
response: "I'm doing great, thanks for asking!"
```

→

```
{
  "instruction": "How are you doing today?", "input": "",
  "output": "I'm doing great, thanks for asking!"

}
```

Code Example:

```
alpaca_data = []
for _, row in df.iterrows():
alpaca_data.append({ "instruction":
row["instruction"],
"input": "", # Optional context field
"output": row["response"]
})
# Save as JSONL for training
with open("alpaca_data.jsonl", "w") as f: for
item in alpaca_data: f.write(json.dumps(item) +
```

Step 4: WhatsApp Chat Parser (Optional)

If starting with raw WhatsApp chat exports, use the WhatsApp Chat Parser to extract structured data:

Reading Chat Exports

Parse .txt files exported directly from WhatsApp conversations

Data Extraction

Extract structured data including date/time, sender name, and message text

10:42 AM

How's the project coming along?

User 1

10:45 AM

Almost done! Just finishing the documentation.

User 2

Code Example:

```
chat = WhatsAppChat.from_source("chat_export.txt") df =
```

Step 5: Cleaning & Preprocessing

Preparing WhatsApp data for effective training requires several key preprocessing steps:

Remove Media Placeholders

Filter out non-text content such as `GIF omitted`, `image omitted`, and other media references that don't contribute to the conversation content.

Calculate Time Deltas

Measure time gaps between consecutive messages to identify natural conversation breaks. Messages within a short timeframe (e.g., 5 minutes) are considered part of the same conversation thread.

Group Into Conversation Blocks

Concatenate related messages from the same person into coherent blocks, creating meaningful query-response pairs for training the chatbot.

Raw WhatsApp Export → Cleaning → Conversation Grouping → Training Ready Data

Step 6: Model Training Preparation

Install Libraries for Training

```
!pip install bitsandbytes transformers peft accelerate datasets
```

bitsandbytes
4-bit quantization to reduce memory usage

transformers
HuggingFace models and training utilities

peft
Parameter-Efficient Fine-Tuning adapters

accelerate
Distributed training support

Load Pre-trained Model

```
model_id = "TinyLlama/TinyLlama-1.1B-Chat-v1.0" tokenizer = AutoTokenizer.from_pretrained(model_id) model =
AutoModelForCausalLM.from_pretrained( model_id, load_in_4bit=True, device_map="auto" )
```

Set up LoRA Adapter

```
model = prepare_model_for_kbit_training(model) config = LoraConfig( r=16, # Rank of LoRA adapters lora_alpha=32, # Scaling factor
target_modules=["q_proj", "v_proj"], # Target layers lora_dropout=0.05, # Dropout probability bias="none", task_type="CAUSAL_LM"
) model = get_peft_model(model, config)
```

Step 7: Dataset Preparation & Training

Format Data for Instruction-Tuning

1 **Format Text Template**

Structure conversations with clear instruction/response format

```
data_text = df.apply(lambda row: f"Instruction:
{row['instruction']}\nResponse:  {row['response']}", axis=1)
```

2 **Tokenization**

Convert text to tokens for model processing

```
tokenizer.pad_token = tokenizer.eos_token
train_dataset =
Dataset.from_pandas(pd.DataFrame(data_text)) def
tokenize(batch):
  return tokenizer(batch["text"],
padding="max_length", truncation=True,
max_length=512)
```

Training Configuration

3 **Training Parameters**

Using 4-bit quantized TinyLlama for efficiency

- Model: TinyLlama-1.1B-Chat-v1.0
- Quantization: 4-bit
- Batch size: 8
- Training epochs: 1
- LoRA adapter for parameter-efficient fine-tuning

4 **Trainer Setup**

HuggingFace Trainer with optimized settings

```
trainer = transformers.Trainer(
  model=model,
  train_dataset=tokenized_dataset,
  args=training_args,
  data_collator=DataCollatorForLanguageModeling(
    tokenizer=tokenizer, mlm=False
  )
```

Step 8: Model Deployment & Chat Interface

Deploy your chatbot with two different approaches:

Option 1: Use Your Fine-tuned TinyLlama

Load and use your personalized model trained on WhatsApp data

```
# Load your fine-tuned model
from transformers import AutoModelForCausalLM, AutoTokenizer
model_path = "fine_tuned_model" # Your saved model path model =
AutoModelForCausalLM.from_pretrained(model_path) tokenizer =
AutoTokenizer.from_pretrained(model_path)
```

Option 2: Use Zephyr 7B for Demo

Implement CLI chat interface with open-source Zephyr model

```
model_name = "HuggingFaceH4/zephyr-7b-alpha" tokenizer =
AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name) def
chat(role, nonrole):
   # Implements chat interface with role assignment #
```

Step 9: Summary & Best Practices

End-to-End Pipeline Summary

- Complete WhatsApp to chatbot training pipeline
- Fine-tuning benefits: personalized responses that reflect conversation style and domain knowledge

Best Practices

Clean Data Thoroughly

Remove media placeholders, filter irrelevant messages, and group conversations properly for quality training data.

Use LoRA Adapters

Maximize resource efficiency with parameter-efficient fine-tuning techniques instead of full model training.

Test Interactively

Run periodic inference tests during and after training to ensure the model generates appropriate responses.

Pro tip: Start with smaller models like TinyLlama before scaling to larger models like Zephyr-7B for faster iteration and testing.