````
 finetune.py

This script takes a pre-trained model (like GPT-2) and further trains it
on a specific dataset for a particular task. It's like teaching a general
knowledge expert to become a specialist in your field. This process
allows the model to learn the nuances and specifics of your data,
improving its performance on your particular use case.
````
```
import os
import torch
print("CUDA Available:", torch.cuda.is_available())
print("Current Device:", torch.cuda.current_device() if
torch.cuda.is_available() else "No GPU found")
print("Device Name:", torch.cuda.get_device_name(0) if
torch.cuda.is_available() else "No GPU found")
import json
import logging
from transformers import AutoModelForCausalLM, AutoTokenizer,
TrainingArguments, Trainer, BitsAndBytesConfig
from torch.utils.data import Dataset
import PyPDF2
from docx import Document
from pptx import Presentation
from peft import get_peft_model, LoraConfig, TaskType
import configparser
from dotenv import load_dotenv
import tkinter as tk
from tkinter import ttk, scrolledtext
import datetime
import glob
from transformers import GPT2LMHeadModel, GPT2Tokenizer, pipeline
import warnings
from pathlib import Path

print("Starting GPT-2 Medium QA System...")

# Set up environment variable for CUDA allocation
os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True'
torch.cuda.empty_cache()

# Configure logging
home_dir = str(Path.home())
log_dir = r"C:\ Local_LLM\GPT2-Medium\GPT2-Medium-Logs"
os.makedirs(log_dir, exist_ok=True)
current_datetime = datetime.datetime.now()
date_str = current_datetime.strftime("%Y%m%d")
existing_logs = glob.glob(os.path.join(log_dir,
f'yaghi_{date_str}_*.log'))

# Determine the next log number
if existing_logs:
    next_number = max([int(log.split('_')[-1].split('.')[0]) for log in
existing_logs]) + 1
```

```
else:
    next_number = 1

log_filename = f'yaghi_{date_str}_{next_number:03d}.log'
log_file_path = os.path.join(log_dir, log_filename)

# Set up logging
logging.basicConfig(
    filename=log_file_path,
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
logger.info("Logging setup complete. Starting the application.")

# Set up logging
logging.basicConfig(
    filename=log_file_path,
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logging.info("Logging setup complete. Starting the application.")

# Function to read DOCX files
def read_docx(file_path):
    try:
        doc = Document(file_path)
        return "\n".join([para.text for para in doc.paragraphs])
    except Exception as e:
        logger.error(f"Error reading {file_path}: {e}")
        return ""

# Function to read PPTX files
def read_pptx(file_path):
    try:
        prs = Presentation(file_path)
        text = ""
        for slide in prs.slides:
            for shape in slide.shapes:
                if hasattr(shape, "text"):
                    text += shape.text + "\n"
        return text
    except Exception as e:
        logger.error(f"Error reading {file_path}: {e}")
        return ""

# Function to read PDF files
def read_pdf(file_path):
    try:
        with open(file_path, 'rb') as file:
            pdf_reader = PyPDF2.PdfReader(file)
            text = ""
            for page in pdf_reader.pages:
                text += page.extract_text() + "\n"
```

```
        return text
    except Exception as e:
        logger.error(f"Error reading {file_path}: {e}")
        return ""

# Function to process all files in a directory
def process_directory(directory, processed_files):
    all_text = ""
    file_count = 0
    newly_processed_files = {}

    for filename in os.listdir(directory):
        file_path = os.path.join(directory, filename)

        # Check if file has been processed before
        if filename in processed_files:
            modification_time = os.path.getmtime(file_path)
            if modification_time <= processed_files[filename]:
                logger.info(f"Skipping {filename} as it has not been
modified")
                continue

        if filename.lower().endswith('.pdf'):
            text = read_pdf(file_path)
        elif filename.lower().endswith('.docx'):
            text = read_docx(file_path)
        elif filename.lower().endswith('.pptx'):
            text = read_pptx(file_path)
        else:
            continue

        all_text += text
        file_count += 1
        newly_processed_files[filename] = os.path.getmtime(file_path)
        logger.info(f"Processed {filename}, added {len(text)}
characters")

    logger.info(f"Total new files processed: {file_count}")
    return all_text, newly_processed_files

class GPT2Dataset(Dataset):
    def __init__(self, text, tokenizer, max_length):
        self.tokenizer = tokenizer
        self.scaler = torch.cuda.amp.GradScaler()
        self.text = text
        self.max_length = max_length

    def __len__(self):
        return len(self.text) // self.max_length

    def __getitem__(self, idx):
        start_idx = idx * self.max_length
        end_idx = start_idx + self.max_length
        chunk = self.text[start_idx:end_idx]
```

```
        inputs = self.tokenizer(chunk, truncation=True,
max_length=self.max_length, return_tensors="pt")
        inputs = {key: value.squeeze(0) for key, value in inputs.items()}

        if inputs['input_ids'].size(0) < self.max_length:
            padding_length = self.max_length -
inputs['input_ids'].size(0)
            inputs['input_ids'] = torch.cat([inputs['input_ids'],
torch.full((padding_length,), self.tokenizer.eos_token_id,
dtype=torch.long)])
            inputs['attention_mask'] =
torch.cat([inputs['attention_mask'], torch.zeros(padding_length,
dtype=torch.long)])
        inputs["labels"] = inputs["input_ids"].clone()
        return inputs

def main():
    # Set up directories
config = configparser.ConfigParser()
config.read('config.ini')
dataset_path = "C:\\Local_LLM\\yaghiDataSet"
context_size = config.getint('DEFAULT', 'context_size', fallback=1000)
fine_tuned_model_path = config.get('DEFAULT', 'fine_tuned_model_path',
fallback="gpt2-medium")

    # Ensure processed_data directory exists
    os.makedirs(processed_dir, exist_ok=True)

    output_file = os.path.join(processed_dir, "processed_dataset.txt")
    processed_files_json = os.path.join(processed_dir,
"processed_files.json")

    # Load previously processed files
    if os.path.exists(processed_files_json):
        try:
            with open(processed_files_json, 'r') as f:
                content = f.read()
                if content.strip():  # Check if the file is not empty
                    processed_files = json.loads(content)
                    logger.info(f"Loaded information about
{len(processed_files)} previously processed files.")
                else:
                    processed_files = {}
                    logger.info("Processed files JSON exists but is
empty. Starting fresh.")
        except json.JSONDecodeError:
            processed_files = {}
            logger.warning("Error decoding JSON in processed files.
Starting fresh.")
    else:
        processed_files = {}
        logger.info("No previously processed files found. Starting
fresh.")
```

```
    processed_text, newly_processed_files =
process_directory(dataset_dir, processed_files)
    logger.info(f"Total processed text length: {len(processed_text)}")

    if len(processed_text) == 0:
        logger.info("No new text was processed. Exiting.")
        return

    # Append new text to existing processed dataset
    with open(output_file, 'a', encoding='utf-8') as f:
        f.write(processed_text)

    # Update processed files list
    processed_files.update(newly_processed_files)
    with open(processed_files_json, 'w') as f:
        json.dump(processed_files, f)
    logger.info(f"Updated processed_files.json with
{len(newly_processed_files)} new or modified files.")

    # Set up device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    logger.info(f"Using device: {device}")

    if device.type == 'cpu':
        logger.warning("Training on CPU. This will be very slow. Consider
using a GPU.")

    # Load model and tokenizer
    model_name = "gpt2-medium"
    model = GPT2LMHeadModel.from_pretrained(model_name).to(device)
    tokenizer = AutoTokenizer.from_pretrained("gpt2",
clean_up_tokenization_spaces=False)

    # Load all processed text
    with open(output_file, 'r', encoding='utf-8') as f:
        all_processed_text = f.read()

    # Prepare the dataset
    max_length = 256
    train_dataset = GPT2Dataset(all_processed_text, tokenizer,
max_length)

    # Split the dataset into training and evaluation sets
    train_size = int(0.9 * len(train_dataset))
    eval_size = len(train_dataset) - train_size
    train_dataset, eval_dataset =
torch.utils.data.random_split(train_dataset, [train_size, eval_size])

    if len(train_dataset) == 0:
        logger.error("Empty training dataset. Cannot proceed with
training.")
        return

    # Training arguments
```

```
    training_args = TrainingArguments(
        output_dir=os.path.join(base_dir, "results"),
        overwrite_output_dir=True,
        num_train_epochs=5,
        per_device_train_batch_size=4,
        gradient_accumulation_steps=4,
        save_steps=10_000,
        save_total_limit=2,
        learning_rate=1e-4,
        warmup_steps=500,
        fp16=torch.cuda.is_available(),
        evaluation_strategy="steps",
        eval_steps=500,
        logging_steps=100,
    )

    # Trainer initialization
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset,
        data_collator=lambda data: {
            'input_ids': torch.stack([f['input_ids'] for f in data]),
            'attention_mask': torch.stack([f['attention_mask'] for f in
data]),
            'labels': torch.stack([f['labels'] for f in data])
        },
    )

    # Start training
    trainer.train()

    # Save the fine-tuned model
    model.save_pretrained(os.path.join(base_dir, "fine_tuned_gpt2"))
    tokenizer.save_pretrained(os.path.join(base_dir, "fine_tuned_gpt2"))

if __name__ == "__main__":
    main()
```