# Language Modelling and Agentic AI Systems

Complete Laboratory Manual (Revised 2026)


**Prepared by:**

Nilesh Sarkar

5th Semester Student

Artificial Intelligence and Robotics Engineering

Dayananda Sagar University, Bangalore

**Under the Supervision of:**

Dr. Pramod Kumar Naik

Chairperson, Dept of Artificial Intelligence and Robotics Engineering

Dayananda Sagar University, Bangalore

# Revised Laboratory Curriculum

| | |
|---|---|
| Week 1 | Foundations: Robust Scraping, Error Handling & Cosine Similarity |
| Week 2 | Conversational AI: Memory, Tools & Multi-Modal Assistant Architecture |
| Week 3 | Architecture: Deep Dive into BPE Tokenization & Structured UI |
| Week 4 | Prompt Engineering: Advanced Steering & AI Safety (Anti-Injection) |
| Week 5 | RAG Systems: Chunking Strategies & Persistent Vector DBs (ChromaDB) |
| Week 6 | Local LLMs: Performance Benchmarking, Quantization & Modelfiles |
| Week 7 | Agentic AI: Self-Reflective Loops with LangGraph & Reasoning |
| Week 8 | Evaluation: RAG Triad & Performance Telemetry (Cost/Latency) |
| Week 9 | Production: FastAPI Streaming & Human-in-the-Loop (HITL) Capstone |

# Week 1 Lab Manual

## Foundations of Deep Learning & AI Functionality

**Instructor Note**: This lab manual provides the aim, code, and explanation for each practical task. Focus on the architectural patterns and the transition from theoretical concepts to functional AI implementations.

---

# Week 1: Foundations of Language Modelling & Setup

## The Journey from NLP to Modern Transformers

### Weekly Table of Contents



- Environment Configuration
- Website Scraping Logic
- Summarization Logic
- Local Model Integration (Ollama)
- LangChain Re-implementation

### Learning Objectives

Welcome to the Language Modelling curriculum! This week, we bridge the gap between traditional NLP and modern GenAI. You will learn:

1. **Fundamental Concepts**: A look at NLP, Deep Learning, and the Transformer architecture.
2. **Environment Setup**: Configuring Google Gemini 1.5 Flash and Ollama.
3. **Prompt & Response**: Understanding how to talk to models (Cloud vs Local).
4. **Basics of NLP**: Tokenization, Embeddings, and why "Context" matters.
5. **Hands-on Project**: Building a "Web Research Assistant" using Gemini.

---

## 1.1 Basics of NLP & Deep Learning

#### What is NLP?

**Natural Language Processing (NLP)** is a branch of artificial intelligence that helps computers understand, interpret, and manipulate human language.

- **Tokenization**: The process of converting a sequence of characters into a sequence of tokens. For example, the sentence "I love coding" becomes `["I", "love", "coding"]`.
- **Embeddings**: Words aren't numbers, but computers need numbers. Embeddings represent words as dense vectors (lists of numbers) in a high-dimensional space where words with similar meanings are closer together.
- **Stop Words**: Common words (like "the", "a", "is") that are often removed in traditional NLP to focus on "meaningful" words.

---

# Lab 1.1: Basic Tokenization

**Aim**: To understand the fundamental concept of tokenization by comparing simple whitespace splitting with regular expression-based word boundary detection.

**Explanation**: This lab demonstrates two primary methods of tokenization:

1. **Simple Split**: Uses Python's `split()` method, which separates text based on whitespace. This often keeps punctuation attached to words (e.g., "token.").
2. **Regex Split**: Uses the pattern `\w+|[^\w\s]` to extract words ( `\w+` ) or individual punctuation characters ( `[^\w\s]` ), providing a much cleaner set of tokens for natural language processing tasks.

*Insight: Modern LLMs use 'Subword Tokenization' which we will explore in Week 3!*

```
# --- Lab 1.1: Basic Tokenization ---
import re

text = "Language Modelling is the art of predicting the next token. Isn't it fascinating?"

# 1. Simple Word Tokenization (Split by space)
tokens_simple = text.split()
print(f"Simple Split ({len(tokens_simple)}): {tokens_simple}")

# 2. Regex Tokenization (Handling punctuation)
tokens_regex = re.findall(r"\w+|[^\w\s]", text)
print(f"Regex Split ({len(tokens_regex)}): {tokens_regex}")

# Insight: Modern LLMs use 'Subword Tokenization' which we will explore in Week 3!
```

# Lab 1.2: Building a Website Summarizer

**Aim**: To build a production-ready web scraping and summarization tool that utilizes Gemini 1.5 Flash to process large-scale text content from live URLs.

**Explanation**: This project implements a complete pipeline for AI-driven web research:

1. **Configuration**: Uses `dotenv` to securely manage API keys.
2. **Scraping**: Leveraging `BeautifulSoup` and `requests` to extract clean text while ignoring boilerplate elements like scripts and navigation bars.
3. **Prompt Engineering**: A structured system prompt guides the model to act as a research assistant, ensuring concise markdown output.
4. **Integration**: Successfully connects to **Gemini 1.5 Flash** for high-performance cloud processing and **Ollama** for local execution, providing a hybrid deployment model.

## Import Required Libraries

We'll import all the necessary libraries for web scraping (requests, BeautifulSoup), environment variables (dotenv), Gemini AI (google.generativeai), and display formatting (IPython.display).

```
# 📦 WEEK 1 INITIALIZATION
import os
import requests
from bs4 import BeautifulSoup
from dotenv import load_dotenv
import google.generativeai as genai
import ollama
from IPython.display import Markdown, display

# --- CONFIGURATION ---
load_dotenv(override=True)

# GEMINI CLOUD SETUP
GEMINI_API_KEY = os.getenv("GOOGLE_API_KEY") or os.getenv("GEMINI_API_KEY")
if GEMINI_API_KEY:
    genai.configure(api_key=GEMINI_API_KEY)
else:
    print("❌ ERROR: GOOGLE_API_KEY not found in environment.")

# MODEL CONFIGURATION
CLOUD_MODEL = "gemini-1.5-flash"
LOCAL_MODEL = "gemma2:2b"

# Initialize models
model = genai.GenerativeModel(CLOUD_MODEL)

# Verify Ollama status
try:
    ollama.list()
    print("✅ Ollama local server is active.")
except Exception:
    print("⚠️ Warning: Ollama server not detected. Local model features will be unavailable."

print(f"✅ Cloud Model Configured: {CLOUD_MODEL}")
print(f"✅ Local Model Configured: {LOCAL_MODEL}")
```

```python
# 🌐 WEB SCRAPING INFRASTRUCTURE
# Robust website content extraction using BeautifulSoup

class Website:
    """
    A utility class for fetching and cleaning webpage content for LLM consumption.
    """
    def __init__(self, url: str):
        self.url = url
        self.headers = {
            "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML
        }
        try:
            response = requests.get(url, headers=self.headers, timeout=10)
            response.raise_for_status()
            soup = BeautifulSoup(response.content, 'html.parser')

            self.title = soup.title.string if soup.title else "Untitled Page"

            # Clean unwanted elements
            if soup.body:
                for element in soup.body(["script", "style", "img", "input", "nav", "footer"])
                    element.decompose()
                self.text = soup.body.get_text(separator="\n", strip=True)
            else:
                self.text = "No content found in body."

        except Exception as e:
            self.title = "Error"
            self.text = f"Failed to fetch {url}: {str(e)}"

    def get_contents(self) -> str:
        return f"Page Title: {self.title}\n\nContent:\n{self.text[:15000]}" # Limit context wi

# Test the infrastructure
test_site = Website("https://blog.google/technology/ai/")
print(f"✅ Website Scraping Test: {test_site.title}")
print(f"Content length: {len(test_site.text)} chars")
```

```python
# ✍️ SUMMARIZATION LOGIC
# Defining the system prompt and the summarization pipeline

system_prompt = (
    "You are an expert technical assistant. Your task is to analyze the content of a website "
    "and provide a concise, professional summary in Markdown format. Focus on core features, "
    "announcements, and key takeaways."
)

def summarize_website(url: str):
    """
    Fetches website content and generates a summary using Gemini.
    """
    site = Website(url)

    # Construct user message
```

```
    user_message = (
        f"Analyze the following website titled '{site.title}'.\n\n"
        f"Content:\n{site.text[:10000]}"
    )

    # Call Gemini (initialized in the first block)
    # Using the standardized 'model' instance
    response = model.generate_content([system_prompt, user_message])
    return response.text

def display_summary(url: str):
    """Utility to display the markdown summary in the notebook"""
    print(f"Summarizing: {url}...")
    summary = summarize_website(url)
    display(Markdown(summary))

# Execution
display_summary("https://openai.com/news/")
```

```
# 🏠 LOCAL MODELS WITH OLLAMA
# Running open-source models (Gemma 2:2b) locally for privacy and cost-efficiency.

def call_local_gemma(prompt: str):
    """
    Interacts with the local Gemma 2:2b model via Ollama.
    """
    try:
        response = ollama.chat(model=LOCAL_MODEL, messages=[
            {
                'role': 'user',
                'content': prompt,
            },
        ])
        return response['message']['content']
    except Exception as e:
        return f"Error calling Ollama: {str(e)}"

# Example Usage
print(f"Testing Local Model ({LOCAL_MODEL}):")
print(call_local_gemma("Explain the concept of 'Tokenization' in NLP in one professional sente
```

```
# 🏠 LOCAL SUMMARIZATION
# Adapting the summarization pipeline for local execution using Ollama.

def summarize_local(url, model_name=LOCAL_MODEL):
    """Local summarize function using Ollama"""
    website = Website(url)
    messages = [
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": f"Summarize this website titled '{website.title}':\n\n{web
    ]
    response = ollama.chat(model=model_name, messages=messages)
    return response['message']['content']

def display_summary_local(url):
```

```
    print(f"Summarizing Locally ({LOCAL_MODEL}): {url}")
    summary = summarize_local(url)
    display(Markdown(summary))

# Test local summary
display_summary_local("https://blog.google/technology/ai/")
```

# Company Brochure Generator

## Enhanced Website Class with Link Extraction

This extended version of our Website class also extracts all links from the webpage, which we'll use to find relevant company pages.

```
# 💼 EXTENDED APPLICATION: THE COMPANY BROCHURE GENERATOR
# Combining link extraction, AI filtering, and multi-page summarization.

from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import JsonOutputParser
from pydantic import BaseModel, Field

class LinkList(BaseModel):
    links: List[dict] = Field(description="A list of relevant links with 'type' (e.g., About,

class EnhancedWebsite(Website):
    """Extended Website class that extract and filters links"""
    def __init__(self, url):
        super().__init__(url)
        try:
            response = requests.get(url, headers={"User-Agent": "Mozilla/5.0"}, timeout=10)
            soup = BeautifulSoup(response.content, 'html.parser')
            links = [link.get('href') for link in soup.find_all('a') if link.get('href')]
            self.links = list(set([url.rstrip('/') + l if l.startswith('/') else l for l in li
        except: self.links = []

def get_relevant_links(url):
    website = EnhancedWebsite(url)
    links_text = "\n".join(website.links[:30])
    prompt = ChatPromptTemplate.from_template("Identify top 3 links (About, Careers, Products)
    chain = ChatGoogleGenerativeAI(model=MODEL, temperature=0) | JsonOutputParser()
    return chain.invoke({"links": links_text})

def create_brochure(company_name, url):
    print(f"Creating brochure for {company_name}...")
    details = EnhancedWebsite(url).get_contents()
    links = get_relevant_links(url)

    for l in links.get('links', []):
        try: details += f"\n\n-- {l['type']} --\n" + Website(l['url']).get_contents()
        except: pass

    prompt = f"Create a markdown brochure for {company_name} using this info:\n{details[:8000]
```

```
    return model.generate_content(prompt).text

# Example execution
# brochure = create_brochure("HuggingFace", "https://huggingface.co")
# display(Markdown(brochure))
print("✅ Brochure Generation logic defined.")
```

---

## Lab 1.3: Intro to LangChain

**Aim**: To recreate our website summarizer using **LangChain**, the industry-standard framework for building LLM-powered applications.

**Explanation**: This lab introduces the LangChain framework to rebuild our summarization pipeline. It highlights key advantages:

- **Composability**: Chain different components together using LCEL.
- **Model Agnostic**: Swap Gemini with other models (like local Llama/Gemma) easily.
- **Rich Eco-system**: Built-in parsers, prompt templates, and output handling.

```
# 🦜🔗 LANGCHAIN RE-IMPLEMENTATION
# Using ChatGoogleGenerativeAI to wrap our Gemini model
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

chat_model = ChatGoogleGenerativeAI(
    model="gemini-1.5-flash",
    google_api_key=GEMINI_API_KEY,
    temperature=0
)

# 1. Define the Prompt Template
summary_template = """
You are a professional technical researcher.
Analyze the following website content and provide a concise, bulleted summary in Markdown.
Focus on core value propositions and key announcements.

Website Title: {title}
Website Content: {content}
"""

summary_prompt = ChatPromptTemplate.from_template(summary_template)

# 2. Build the LCEL Chain (LangChain Expression Language)
summarize_chain = summary_prompt | chat_model | StrOutputParser()

# 3. Execute for a website
def langchain_summarize(url):
    ws = Website(url)
    result = summarize_chain.invoke({
        "title": ws.title,
```

```
        "content": ws.get_contents()
    })
    display(Markdown(result))

# Test LangChain implementation
print("Summarizing via LangChain...")
langchain_summarize("https://www.deepmind.com")
```

## Summary and Learning Outcomes - Week 1

- **Model Mastery**: You've used both Google's state-of-the-art **Gemini 1.5 Flash** and locally hosted **Gemma 2** via Ollama.
- **Workflow Automation**: Built a complete pipeline from raw URL to professional summary.
- **Modern Paradigms**: Introduced **LangChain** and **LCEL** (LangChain Expression Language) for building robust AI pipelines.

**Next Week**: We dive deeper into Conversational AI and advanced UI development with Gradio.

---

# Instructor's Evaluation & Lab Summary

## Assessment Criteria

1. **Technical Implementation**: Adherence to the lab objectives and code functionality.
2. **Logic & Reasoning**: Clarity in the explanation of the underlying AI principles.
3. **Best Practices**: Use of secure environment variables and structured prompts.

**Lab Completion Status: Verified Focus Area**: Language Modelling & Deep Learning Systems.

# Week 2 Lab Manual

## Foundations of Deep Learning & AI Functionality

**Instructor Note**: This lab manual provides the aim, code, and explanation for each practical task. Focus on the architectural patterns and the transition from theoretical concepts to functional AI implementations.

---

# Week 2: Deep Learning for NLP & Conversational AI

# From RNNs to Memory Systems

## Weekly Table of Contents



## Learning Objectives

Last week we looked at basic NLP. This week, we understand the *Deep Learning* that powers it and how to build chat systems. You will learn:

1. **RNNs & LSTMs**: The predecessors to Transformers and why they were replaced.
2. **Stateful Interactions**: Understanding the difference between stateless and stateful chat.
3. **LangChain Memory**: Using buffer and window memory with Gemini.
4. **Prompt Personas**: Designing system instructions to give your AI a specific "personality."
5. **Gradio Dashboards**: Creating a professional web interface for your LLM.

---

## 2.1 The Evolution: RNNs to Transformers

### Sequential vs. Parallel Processing

- **RNN (Recurrent Neural Network)**: Processes text one word at a time. It's slow and forgets the beginning of long sentences (The "Vanishing Gradient" problem).
- **Transformer (Parallel Processing)**: Processes all words at once. It's fast and uses "Attention" to link related words regardless of distance.

### Chat vs. Completion

A standard model completes text. A **Chat Model** is trained specifically for dialogue, using special system frames (messages) to distinguish between User, AI, and System.

---

# Lab 2.1: Chat Interface with Gemini & Gemma

**Aim**: To build a stateful, streaming chat interface that uses Gemini 1.5 Flash for high-quality interactions and showcases the transition from sequential processing to parallel transformer models.

**Explanation**: This lab establishes the foundation for conversational AI:

1. **Preprocessing Logic**: Visualizing the difference between sequential RNN-style and parallel Transformer-style processing.
2. **Model Orchestration**: Creating uniform wrappers for both Cloud (Gemini) and Local (Gemma) model calls.
3. **Real-Time UX**: Implementing streaming interfaces with Gradio to provide immediate feedback to users, mirroring modern AI application standards.

```
# 📦 WEEK 2 INITIALIZATION
import os
import google.generativeai as genai
import ollama
from dotenv import load_dotenv

# --- CONFIGURATION ---
load_dotenv(override=True)
MODEL = "gemini-1.5-flash"
LOCAL_MODEL = "gemma2:2b"

# Ensure API Key is available
GEMINI_API_KEY = os.getenv("GOOGLE_API_KEY") or os.getenv("GEMINI_API_KEY")
if GEMINI_API_KEY:
    genai.configure(api_key=GEMINI_API_KEY)
else:
    print("⚠️ GEMINI_API_KEY not found. Fallback to local model enabled.")

# Simulation of RNN vs Parallel (Theory and Logic)
sentence = "Deep Learning is powerful"

print("--- Sequential (RNN-style) Processing ---")
hidden_state = "init"
for word in sentence.split():
    hidden_state = f"State({hidden_state} + {word})"
    print(f" -> {hidden_state}")

print("\n--- Parallel (Transformer-style) Processing ---")
print(f" -> Tokens: {sentence.split()} processed simultaneously via Self-Attention matrix")
```

```
# 🤖 CONVERSATIONAL MODEL WRAPPERS
# Standardized interfaces for both Cloud (Gemini) and Local (Gemma) models.
```

```python
def message_gemini(prompt, system_instruction="You are a helpful AI assistant."):
    """Synchronous Gemini call"""
    try:
        model = genai.GenerativeModel(model_name=MODEL, system_instruction=system_instruction)
        response = model.generate_content(prompt)
        return response.text
    except Exception as e:
        return f"Gemini Error: {e}"

def message_gemma(prompt, system_instruction="You are a helpful AI assistant."):
    """Synchronous Gemma 2 (Local) call"""
    try:
        full_prompt = f"{system_instruction}\n\nUser: {prompt}\nAssistant:"
        response = ollama.generate(model=LOCAL_MODEL, prompt=full_prompt)
        return response['response']
    except Exception as e:
        return f"Ollama Error: {e}"

# Streaming helper for Gemini
def stream_gemini(prompt, system_instruction="You are a helpful AI assistant."):
    model = genai.GenerativeModel(model_name=MODEL, system_instruction=system_instruction)
    response = model.generate_content(prompt, stream=True)
    for chunk in response:
        if chunk.text: yield chunk.text

# Test the wrappers
test_prompt = "Tell a light-hearted joke for Data Scientists."
print("Gemini:", message_gemini(test_prompt))
# print("Gemma:", message_gemma(test_prompt)) # Uncomment for local test
```

## Streaming Test

Testing streaming functionality from both models.

```python
# Test streaming from both models
test_prompt = 'Explain when to use an LLM for a business problem in a short bullet list'

print('--- Gemini 1.5 Flash streaming test ---')
for chunk in stream_gemini(test_prompt):
    print(chunk)

print('\n--- Gemma 2:2B streaming test ---')
for chunk in stream_gemma(test_prompt):
    print(chunk)
print('--- end streaming test ---')
```

```python
# Streaming Gradio interface (guarded)
if not ALLOW_GRADIO:
    print('Skipping streaming Gradio UI because ALLOW_GRADIO is not set to true')
else:
    import gradio as gr
    def stream_gemini_ui(prompt):
        # Use the streaming helper defined earlier
        for chunk in stream_gemini(prompt):
```

```
            yield chunk
    view = gr.Interface(
        fn=stream_gemini_ui,
        inputs=[gr.Textbox(label="Your message:")],
        outputs=[gr.Markdown(label="Response:")],
        flagging_mode="never",
    )
    view.launch()
```

```
# Claude streaming function (placeholder - Claude not configured in this setup)
claude = None  # Claude not available in this setup

def stream_claude(prompt):
    if not claude:
        yield "Claude API not configured - using local Gemma instead"
        # Fallback to local model
        for chunk in stream_gemma(prompt):
            yield chunk
        return
```

```
# Multi-model interface with Gemini and Gemma (guarded)
if not ALLOW_GRADIO:
    print('Skipping multi-model Gradio UI because ALLOW_GRADIO is not set to true')
else:
    import gradio as gr
    def stream_model(prompt, model):
        if model == 'Gemini':
            for chunk in stream_gemini(prompt):
                yield chunk
        elif model == 'Gemma':
            for chunk in stream_gemma(prompt):
                yield chunk
        else:
            yield 'Unknown model selected'
            return

    view = gr.Interface(
        fn=stream_model,
        inputs=[
            gr.Textbox(label='Your message:'),
            gr.Dropdown(['Gemini', 'Gemma'], label='Select model', value='Gemini')
        ],
        outputs=[gr.Markdown(label='Response:')],
        flagging_mode='never',
    )
    view.launch()
```

```
# Enhanced chat function with fallback logic
def chat_gemini(message, history):
    # Convert history to Gemini format
    gemini_history = []
    for item in history:
        role = "user" if item["role"] == "user" else "model"
        gemini_history.append({"role": role, "parts": [item["content"]]})
```

```
    # Try Gemini first
    if GEMINI_API_KEY:
        try:
            model = genai.GenerativeModel(model_name=MODEL_NAME)
            chat = model.start_chat(history=gemini_history[:-1]) # Don't include the current m
            response = chat.send_message(message, stream=True)

            result = ""
            for chunk in response:
                if hasattr(chunk, 'text') and chunk.text:
                    result += chunk.text
                    yield result
            return # Exit after successful Gemini response
        except Exception as e:
            print(f"Error initializing Gemini: {e}")

    # Fallback to local Gemma model via Ollama
    try:
        # Prepare simple chat prompt for Gemma
        full_prompt = "You are a helpful assistant.\n\n"
        for item in history[:-1]:
            full_prompt += f"{item['role'].capitalize()}: {item['content']}\n"
        full_prompt += f"User: {message}\nAssistant:"

        response = ollama.generate(model=LOCAL_MODEL, prompt=full_prompt, stream=True)
        result = ""
        for chunk in response:
            if 'response' in chunk:
                result += chunk['response']
                yield result
    except Exception as e:
        yield f'❌ Error: Both Gemini and Local {LOCAL_MODEL} failed. {e}'

# Guarded Gradio launch
if not ALLOW_GRADIO:
    print('Skipping Gradio ChatInterface because ALLOW_GRADIO is not true')
else:
    import gradio as gr
    # Using the newer messages format for Gradio 4.x+
    gr.ChatInterface(fn=chat_gemini, type="messages").launch()
```

## Advanced: LangChain Conversational Agent

Moving from manual history management to standard frameworks. LangChain's `ConversationChain` handles history automatically using `ConversationBufferMemory`.

```
# 🔗 LANGCHAIN CONVERSATIONAL AGENT
# Building a memory-aware agent using LangChain and Gemini
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain

def build_langchain_agent():
```

```
    # Model
    llm = ChatGoogleGenerativeAI(model=MODEL_NAME, google_api_key=GEMINI_API_KEY, temperature=

    # Prompt with memory placeholder
    prompt = ChatPromptTemplate.from_messages([
        ("system", "You are a professional technical mentor. Be concise and accurate."),
        MessagesPlaceholder(variable_name="history"),
        ("human", "{input}"),
    ])

    # Memory
    memory = ConversationBufferMemory(return_messages=True)

    # Chain (Legacy syntax for clarity in early weeks, move to LCEL later)
    # Note: ConversationChain expects specific memory key
    conversation = ConversationChain(
        memory=memory,
        prompt=prompt,
        llm=llm
    )
    return conversation

# Initialize agent
try:
    agent = build_langchain_agent()

    def chat_with_agent(user_input):
        response = agent.predict(input=user_input)
        print(f"\nAI: {response}")

    # Test the memory
    print("Test 1: 'Hi, my name is Alex.'")
    chat_with_agent("Hi, my name is Alex.")
    print("\nTest 2: 'What is my name?'")
    chat_with_agent("What is my name?")
except Exception as e:
    print(f"Skipping LangChain agent test: {e}")
```

# Lab 2.2: Airline Assistant with Function Calling (Tools)

**Aim**: To implement a functional "Airline Assistant" using Gemini's native function calling capabilities to retrieve destination pricing from a simulated database.

**Explanation**: This lab demonstrates the power of **Function Calling** (Tools). Instead of just generating text, the model can:

1. **Detect Intent**: Recognize when a user is asking for specific data (like ticket prices).
2. **Generate Structured Calls**: Output the exact function name and arguments needed.
3. **Process Tool Output**: Take the results from the Python function and incorporate them into a natural language response.

```python
# --- Lab 2.2: Airline Assistant with Function Calling ---

# 1. Define Tools
ticket_prices = {'london': '$799', 'paris': '$899', 'tokyo': '$1400', 'berlin': '$499'}

def get_ticket_price(destination_city: str):
    """Returns the ticket price for a given destination city."""
    city = destination_city.lower().strip()
    return {"price": ticket_prices.get(city, "not found (please ask for a valid city: London,

# 2. Integrate with Gemini
def airline_assistant(user_input: str):
    # Initialize model with tools
    model = genai.GenerativeModel(
        model_name=MODEL,
        tools=[get_ticket_price],
        system_instruction="You are a helpful assistant for FlightAI. Use the 'get_ticket_pric
    )

    # Start chat session
    chat = model.start_chat()
    response = chat.send_message(user_input)

    # Handle function calls (Simplistic for Lab 2.2)
    for part in response.candidates[0].content.parts:
        if part.function_call:
            fn = part.function_call
            if fn.name == "get_ticket_price":
                # Execute tool
                result = get_ticket_price(**fn.args)
                # Send result back to model
                response = chat.send_message(
                    genai.protos.Content(
                        parts=[genai.protos.Part(
                            function_response=genai.protos.FunctionResponse(
                                name="get_ticket_price",
                                response=result
                            )
                        )]
                    )
                )

    return response.text

# Test the assistant
print("Assistant:", airline_assistant("How much is a ticket to London?"))
```

# Lab 2.3: Building a Multi-Modal Assistant Architecture

**Aim**: To design a multi-modal assistant architecture capable of processing both text instructions and image inputs for complex reasoning tasks.

**Explanation**: This lab covers the integration of visual and textual data into a single AI workflow:

1. **Image Simulation**: Using the `Pillow` library to simulate an image generation tool (The "Artist").
2. **Unified Input**: Passing both strings and Image objects to Gemini 1.5 Flash.
3. **Contextual Reasoning**: Allowing the AI to "see" and "read" simultaneously to provide travel advice or research insights.

```
# --- Lab 2.3: Building a Multi-Modal Assistant Architecture ---
import PIL.Image
import PIL.ImageDraw
import PIL.ImageFont
import random

# 1. Image Generation Simulation (The "Artist" Tool)
def generate_destination_preview(city: str):
    """Simulates an image generator for travel destinations."""
    img = PIL.Image.new('RGB', (512, 512), color=(random.randint(100,255), random.randint(100,
    draw = PIL.ImageDraw.Draw(img)
    draw.text((150, 250), f"PREVIEW: {city.upper()}", fill=(0,0,0))
    return img

# 2. Multi-Modal Reasoning
def multimodal_researcher(message: str, image_file=None):
    """
    Combines text and image inputs (if provided) to provide travel advice.
    """
    model = genai.GenerativeModel(model_name=MODEL)
    inputs = [message]
    if image_file:
        img = PIL.Image.open(image_file)
        inputs.append(img)

    response = model.generate_content(inputs)
    return response.text

# 3. Final Multi-Modal Dashboard (Simulated logic)
print("✅ Multi-modal logic defined.")
# Note: In a real lab, students would use gr.Image and gr.ChatInterface to combine these.
```

# Summary

This notebook demonstrates a complete multi-modal AI application with:

- **Online AI**: Gemini 1.5 Flash for high-quality responses
- **Local AI**: Gemma 2:2B via Ollama for privacy and offline use
- **Image Generation**: Simple PIL-based destination images
- **Text-to-Speech**: Cross-platform TTS functionality
- **Function Calling**: Intelligent tool usage for ticket prices
- **Multiple Interfaces**: Various Gradio UIs for different use cases

To enable Gradio interfaces, set `ALLOW_GRADIO=true` in your `.env` file. All functions include fallbacks to ensure the notebook works even without API keys or local models.

---

# Instructor's Evaluation & Lab Summary

## Assessment Criteria

1. **Technical Implementation**: Adherence to the lab objectives and code functionality.
2. **Logic & Reasoning**: Clarity in the explanation of the underlying AI principles.
3. **Best Practices**: Use of secure environment variables and structured prompts.

**Lab Completion Status: Verified Focus Area**: Language Modelling & Deep Learning Systems.

# Week 3 Lab Manual

## Foundations of Deep Learning & AI Functionality

**Instructor Note**: This lab manual provides the aim, code, and explanation for each practical task. Focus on the architectural patterns and the transition from theoretical concepts to functional AI implementations.

---

# Week 3: Attention Mechanisms & Structured Output

## Turn Unstructured Text into Actionable Data

### Weekly Table of Contents



### Learning Objectives

LLMs are great at chatting, but they are *amazing* at data extraction. This week we master:

1. **Transformers Core**: Understanding the "Attention Mechanism" that powers Gemini.
2. **Structured Output**: Using Pydantic models to force LLMs to return JSON.
3. **The Parser Pattern**: Transitioning from string responses to Python objects.
4. **Hands-on Project**: Designing an automated **Meeting Minutes Extractor** using Gemini 1.5 Flash.

---

## 3.1 Deep Learning Deep Dive: Attention

### What is Self-Attention?

In a sentence like "*The animal didn't cross the street because **it** was too tired*", how do we know **"it"** refers to the animal and not the street?

- **Self-Attention**: The model assigns "weights" to every other word while processing a specific word. It "attends" to the animal.
- **Multi-Head Attention**: The model looks at the sentence through multiple "lenses" at onceone focused on grammar, one on logic, one on pronouns, etc.

### Tokenization: The Gatekeeper

Modern models like **Gemini** and **Gemma** use subword tokenization (SentencePiece). This allows the model to understand rare words by breaking them into smaller meaningful chunks.

---

# Setup: Gemini 1.5 Flash Configuration

Configures Google's Gemini 1.5 Flash API for online AI operations.

```
# 📦 WEEK 3 INITIALIZATION
import os
import json
import torch
import requests
import pandas as pd
from datetime import datetime
from dotenv import load_dotenv
import google.generativeai as genai
import ollama
from IPython.display import Markdown, display

# HUGGINGFACE & LANGCHAIN
from transformers import pipeline, AutoTokenizer, AutoModel, AutoConfig
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import JsonOutputParser, StrOutputParser
from pydantic import BaseModel, Field
from typing import List, Optional

# --- CONFIGURATION ---
load_dotenv(override=True)
MODEL = 'gemini-1.5-flash'
LOCAL_MODEL = 'gemma2:2b'

GEMINI_API_KEY = os.getenv('GEMINI_API_KEY') or os.getenv('GOOGLE_API_KEY')
if GEMINI_API_KEY:
    genai.configure(api_key=GEMINI_API_KEY)
else:
    print("⚠️ WARNING: GEMINI_API_KEY not found.")

# HuggingFace Setup
device = 0 if torch.cuda.is_available() else -1
classifier_hf = pipeline("sentiment-analysis", model="distilbert-base-uncased-finetuned-sst-2-

print(f"✅ Week 3 Ready. Cloud Model: {MODEL} | Local Model: {LOCAL_MODEL}")
```

# Lab 3.1: Comparing Text Generation (HuggingFace vs Gemini vs Ollama)

**Aim**: To contrast high-level results across three distinct environments: Local Transformers (GPT-2), Cloud APIs (Gemini 1.5 Flash), and Local LLM runners (Ollama/Gemma 2:2b).

**Explanation**: This lab evaluates both **Creative Generation** and **Sentiment Analysis** to see how reasoning depth varies by model size and hosting style. It highlights the differences between:

1. **HuggingFace**: Running open-source models (like GPT-2) directly in your memory.
2. **Gemini Cloud**: High-performance, large-scale reasoning via API.
3. **Local Runners**: Managing local models via Ollama for privacy and offline use.

```
# 1. Initialize HuggingFace Pipelines
print("Initializing HuggingFace pipelines...")
generator_hf = pipeline("text-generation", model="gpt2", device=device)
classifier_hf = pipeline("text-classification", model="cardiffnlp/twitter-roberta-base-sentime

# 2. Text Generation Comparison
def compare_text_generation():
    prompt = "The most important habit for an engineer to develop is"
    print(f"=== Text Generation Comparison ===\nPrompt: {prompt}\n")

    # HuggingFace GPT-2
    print("--- HuggingFace GPT-2 ---")
    try:
        res = generator_hf(prompt, max_length=100, num_return_sequences=1)
        print(res[0]['generated_text'])
    except Exception as e: print(f"Error: {e}")

    # Gemini 1.5 Flash
    print("\n--- Gemini 1.5 Flash ---")
    if gemini_model:
        try:
            res = gemini_model.generate_content(prompt)
            print(res.text)
        except Exception as e: print(f"Error: {e}")

    # Ollama gemma2:2b
    print(f"\n--- Ollama {LOCAL_MODEL} ---")
    if ollama_client.is_running():
        try:
            res = ollama_client.generate(LOCAL_MODEL, prompt)
            print(res.get('response', 'No response'))
        except Exception as e: print(f"Error: {e}")

# 3. Sentiment Analysis Comparison
def compare_sentiment(text):
    print(f"--- Sentiment Test: '{text}' ---")

    # HF
    hf_res = classifier_hf(text)[0]
```

```
    print(f"HF: {hf_res['label']} ({hf_res['score']:.2f})")

    # Gemini
    if gemini_model:
        prompt = f'Identify sentiment of: "{text}". Reply with just POSITIVE, NEGATIVE, or NEU
        try:
            print(f"Gemini: {gemini_model.generate_content(prompt).text.strip()}")
        except: pass

    # Ollama
    if ollama_client.is_running():
        prompt = f'Identify sentiment of: "{text}". Reply with just POSITIVE/NEGATIVE/NEUTRAL.
        try:
            print(f"Ollama: {ollama_client.generate(LOCAL_MODEL, prompt).get('response', '').s
        except: pass

# Run Comparisons
compare_text_generation()
print("\n" + "="*50 + "\n")
for t in ["I love working with LLMs!", "This error message is terrible.", "The sky is blue toc
    compare_sentiment(t)
    print("-" * 30)
```

# Lab 3.2: Tokenizer Comparison & Model Performance

**Aim**: To understand how different sub-word tokenization strategies (BPE, WordPiece, SentencePiece) impact model performance and how model architecture (Total Parameters vs Layers) correlates with reasoning capabilities.

**Explanation**: This lab involves:

1. **Tokenizer Inspection**: Comparing how different models split text into numeric tokens.
2. **Architecture Analysis**: Examining model configuratons (hidden layers, attention heads) to understand the "brain" of the LLM.
3. **Performance Trade-offs**: Evaluating why some models are better at logic while others are better at speed.

```
# --- Lab 3.2: Tokenizer Comparison & Analysis ---

def compare_tokenizers(text):
    tokenizer_models = ["gpt2", "bert-base-uncased", "t5-small"]
    results = []

    print(f"Analyzing text: '{text}'\n")
    for model_name in tokenizer_models:
        try:
            tokenizer = AutoTokenizer.from_pretrained(model_name)
            tokens = tokenizer.tokenize(text)
            token_ids = tokenizer.encode(text)
            results.append({
                'Model': model_name,
                'Tokens': len(tokens),
```

```
                'Sample': tokens[:10]
            })
            print(f"{model_name} ({len(tokens)} tokens): {tokens}")
        except Exception as e:
            print(f"Error loading {model_name}: {e}")

    return pd.DataFrame(results)

# Run Comparison
test_text = "Language Modelling with Transformers is revolutionary! Isn't it?"
df_tok = compare_tokenizers(test_text)
display(df_tok)

# Gemini & Ollama Tokenization Insight
def ai_token_insight(text):
    print(f"\n--- AI Insights for: '{text}' ---")
    # Gemini
    try:
        model_gemini = genai.GenerativeModel(MODEL)
        res = model_gemini.generate_content(f"How do you tokenize the word '{text}'? Answer ir
        print(f"Gemini: {res.text.strip()}")
    except: pass

    # Ollama
    try:
        res = ollama.generate(model=LOCAL_MODEL, prompt=f"Explain your tokenization of '{text}
        print(f"Ollama: {res['response'].strip()}")
    except: pass

ai_token_insight("Transformers")
```

```
# Model Architecture Analysis
from transformers import AutoModel, AutoConfig
import torch

def explore_model_architecture(model_name):
    """Analyze model architecture and parameters"""

    print(f"=== {model_name} ===")

    try:
        # Load configuration
        config = AutoConfig.from_pretrained(model_name)

        print(f"Architecture: {config.model_type}")
        print(f"Hidden size: {config.hidden_size}")
        print(f"Layers: {config.num_hidden_layers}")
        print(f"Attention heads: {config.num_attention_heads}")
        print(f"Vocab size: {config.vocab_size}")

        # Load model for parameter count
        model = AutoModel.from_pretrained(model_name)
        total_params = sum(p.numel() for p in model.parameters())

        print(f"Parameters: {total_params:,}")
        print(f"Size (approx): {total_params * 4 / 1024**2:.1f} MB")
```

```
        # Model structure
        print("Layers:")
        for name, module in model.named_children():
            print(f"  {name}: {type(module).__name__}")

    except Exception as e:
        print(f"Error: {e}")

    print("-" * 40)

# Analyze different architectures
models = [
    "distilbert-base-uncased",    # Smaller BERT
    "gpt2",                       # GPT-2
    "t5-small",                   # T5 encoder-decoder
    "facebook/bart-base",         # BART encoder-decoder
]

for model_name in models:
    explore_model_architecture(model_name)
```

# Lab 3.3: Automated Meeting Minutes Creator (Mini-Project)

**Aim**: To extract structured, actionable data from unstructured meeting transcripts using Pydantic-based output parsing and the LangChain Expression Language (LCEL).

**Explanation**:

1. **Schema Definition**: Uses `Pydantic` to define the exact shape of the desired output (Summary, Decisions, Action Items).
2. **Output Parsing**: Employs `JsonOutputParser` to reliably convert the LLM's text output into a Python dictionary or object.
3. **Zero-Shot Extraction**: Leverage the massive context window of Gemini 1.5 Flash to process long transcripts without losing detail.
4. **Actionable Results**: The output is ready for direct integration into project management tools or calendar systems.

```
# Meeting Minutes Creator Class with Structured Output
import json
from datetime import datetime
from pydantic import BaseModel, Field
from typing import List

# Define the schema for meeting minutes
class ActionItem(BaseModel):
    owner: str = Field(description="The person responsible for the task")
    task: str = Field(description="The description of the task")
    deadline: Optional[str] = Field(description="The deadline for the task, if mentioned")
```

```python
class MeetingMinutes(BaseModel):
    summary: str = Field(description="A 2-3 sentence summary of the meeting")
    key_points: List[str] = Field(description="List of key discussion points")
    action_items: List[ActionItem] = Field(description="List of tasks assigned during the meet
    decisions: List[str] = Field(description="List of decisions made during the meeting")

class MeetingMinutesCreator:
    def __init__(self, gemini_model=None, ollama_client=None):
        self.gemini_model = gemini_model
        self.ollama_client = ollama_client
        # Setup LangChain JSON parser
        self.parser = JsonOutputParser(pydantic_object=MeetingMinutes)

    def analyze_with_gemini(self, transcript):
        """Analyze transcript using Gemini 1.5 Flash with structured output"""
        if not GEMINI_API_KEY:
            return "Gemini not configured"

        prompt = f"""
        Analyze this meeting transcript and create structured minutes in JSON format.
        {self.parser.get_format_instructions()}

        TRANSCRIPT: {transcript}
        """

        try:
            # We can use LangChain for structured output more easily
            chat_model = ChatGoogleGenerativeAI(model=MODEL, google_api_key=GEMINI_API_KEY, te
            chain = chat_model | self.parser
            return chain.invoke(prompt)
        except Exception as e:
            # Fallback to simple text if JSON fails
            print(f"Structured Gemini error: {e}")
            try:
                resp = self.gemini_model.generate_content("Summarize this meeting: " + transcr
                return {"summary": resp.text, "error": "Structured output failed"}
            except:
                return {"error": str(e)}

    def analyze_with_ollama(self, transcript):
        """Analyze transcript using Ollama local model"""
        if not self.ollama_client or not self.ollama_client.is_running():
            return "Ollama not available"

        prompt = f"""
        Create meeting minutes from this transcript. Respond only with a JSON object.
        Template: {{"summary": "...", "key_points": ["..."], "action_items": [{{"owner": "...'

        TRANSCRIPT:
        {transcript}
        """

        try:
            response = self.ollama_client.generate(LOCAL_MODEL, prompt)
            if response and 'response' in response:
                # Attempt to find JSON in response if it's not clean
```

```
                text = response['response']
                return text # Simple text return for local comparison
            return "No response"
        except Exception as e:
            return f"Ollama error: {e}"

    def create_minutes(self, transcript):
        """Create meeting minutes using both models"""

        print("=== Creating Meeting Minutes ===")
        print(f"Transcript length: {len(transcript)} characters")
        print("=" * 50)

        # Gemini analysis (Structured)
        print("\n--- Gemini 1.5 Flash (Structured) ---")
        gemini_result = self.analyze_with_gemini(transcript)
        print(json.dumps(gemini_result, indent=2) if isinstance(gemini_result, dict) else gemi

        # Ollama analysis (Text-based)
        print(f"\n--- OLLAMA {LOCAL_MODEL} ---")
        ollama_result = self.analyze_with_ollama(transcript)
        print(ollama_result)

        return {
            'gemini': gemini_result,
            'ollama': ollama_result,
            'timestamp': datetime.now().isoformat()
        }

# Initialize meeting minutes creator
minutes_creator = MeetingMinutesCreator(gemini_model, ollama_client)
print("✅ Meeting Minutes Creator ready!")
```

## Demo: Processing Sample Meeting

Demonstrates the meeting minutes creator with a sample project status meeting transcript.

```
# Sample meeting transcript for demonstration
sample_transcript = """
Good morning everyone. I'm Sarah, project manager. We have John from development,
Lisa from QA, and Mike from marketing.

John: We've completed 80% of the user authentication module. Login and registration
work well, but password reset has API integration issues that will delay us 2-3 days.

Lisa: I've tested completed features - found 3 minor bugs in login that are fixed.
Password reset is critical, so we should prioritize this.

Mike: Marketing materials are ready, but if there's a delay, we need to adjust
our launch timeline. Can we get a firm delivery date?

John: I'm confident we can finish by next Friday if we focus on password reset.
I'll work overtime if needed.
```

```
Action items: John prioritizes password reset fix, Lisa prepares for immediate testing,
Mike prepares for potential one-week campaign delay. Next meeting Thursday.
"""

print("=== SAMPLE MEETING TRANSCRIPT ===")
print(sample_transcript)
print("\n" + "="*60)

# Process the transcript
results = minutes_creator.create_minutes(sample_transcript)

# Save results
output_file = "meeting_minutes_results.json"
with open(output_file, 'w') as f:
    json.dump(results, f, indent=2)

print(f"\n✅ Results saved to {output_file}")
```

```
# Audio processing capability (requires audio file)
import speech_recognition as sr
import os

def process_audio_file(file_path):
    """Process audio file and create meeting minutes"""

    if not os.path.exists(file_path):
        print(f"Audio file not found: {file_path}")
        return None

    print(f"Processing: {file_path}")

    # Transcribe audio
    recognizer = sr.Recognizer()
    try:
        with sr.AudioFile(file_path) as source:
            recognizer.adjust_for_ambient_noise(source)
            audio = recognizer.record(source)

        transcript = recognizer.recognize_google(audio)
        print(f"Transcription complete: {len(transcript)} characters")

        # Create meeting minutes
        results = minutes_creator.create_minutes(transcript)
        return results

    except sr.UnknownValueError:
        print("Could not understand audio")
        return None
    except sr.RequestError as e:
        print(f"Speech recognition error: {e}")
        return None

print("Audio processing function ready!")
print("Usage: process_audio_file('path/to/your/audio.wav')")
```

```
# Example usage (uncomment when you have an audio file):
# results = process_audio_file("meeting_recording.wav")
```

---

# Instructor's Evaluation & Lab Summary

## Assessment Criteria

1. **Technical Implementation**: Adherence to the lab objectives and code functionality.
2. **Logic & Reasoning**: Clarity in the explanation of the underlying AI principles.
3. **Best Practices**: Use of secure environment variables and structured prompts.

**Lab Completion Status: Verified Focus Area**: Language Modelling & Deep Learning Systems.

# Week 4 Lab Manual

## Foundations of Deep Learning & AI Functionality

**Instructor Note**: This lab manual provides the aim, code, and explanation for each practical task. Focus on the architectural patterns and the transition from theoretical concepts to functional AI implementations.

---

# Week 4: Training Paradigms & Prompt Engineering

# Optimizing Model Performance with Gemini

## Weekly Table of Contents



- Cross-Platform Compilation Logic
- System Prompting for Performance
- Real-time Execution Benchmarking

## Learning Objectives

Effective prompting is the difference between a generic response and a state-of-the-art solution. This week, we explore advanced prompting techniques and the training history of LLMs. You will learn:

1. **Transfer Learning**: How models are pre-trained on the internet and fine-tuned for tasks.
2. **System Prompting**: Crafting personas and constraints for Gemini 1.5 Flash.
3. **Few-Shot CoT**: Using Chain-of-Thought prompting to solve complex logic.
4. **Structured Parsers**: Using LangChain to extract JSON and code.
5. **Sampling**: Understanding Temperature, Top-P, and Top-K.

---

```
# 📦 WEEK 4 INITIALIZATION
import os
import json
import subprocess
import platform
import io
import sys
import time
```

```python
from dotenv import load_dotenv
import google.generativeai as genai
import ollama
from IPython.display import Markdown, display
import gradio as gr

# LANGCHAIN PROMPTING & PARSING
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate, FewShotChatMessagePromptTemplate
from langchain_core.output_parsers import JsonOutputParser, CommaSeparatedListOutputParser
from pydantic import BaseModel, Field

# Load environment variables
load_dotenv(override=True)
GEMINI_API_KEY = os.getenv('GEMINI_API_KEY') or os.getenv('GOOGLE_API_KEY')
HF_TOKEN = os.getenv('HF_TOKEN')

# Configure Gemini
if GEMINI_API_KEY:
    genai.configure(api_key=GEMINI_API_KEY)
else:
    print("⚠️ WARNING: GEMINI_API_KEY not found.")

# Global Model Names
GEMINI_MODEL = 'gemini-1.5-flash'
LOCAL_MODEL = 'gemma2:2b'

# Define Sample Algorithms for the exercises
pi_calculation = """
import time
def calculate_pi(n_terms):
    start = time.time()
    pi = 0
    for i in range(n_terms):
        pi += 4 * ((-1)**i) / (2*i + 1)
    end = time.time()
    print(f"PI: {pi:.10f}")
    print(f"Time: {end - start:.6f}s")

calculate_pi(1000000)
"""

python_complex = """
import time
import random

def max_subarray_sum(n, seed, min_val, max_val):
    random.seed(seed)
    random_numbers = [random.randint(min_val, max_val) for _ in range(n)]

    max_sum = float('-inf')
    current_sum = 0
    for x in random_numbers:
        current_sum = max(x, current_sum + x)
        max_sum = max(max_sum, current_sum)
    return max_sum
```

```
def run_test():
    start = time.time()
    total = 0
    for i in range(20):
        total += max_subarray_sum(10000, 42 + i, -10, 10)
    end = time.time()
    print(f"Total Max Sum: {total}")
    print(f"Time: {end - start:.6f}s")

run_test()
"""

print(f"✅ Week 4 Ready: Cloud ({GEMINI_MODEL}) and Local ({LOCAL_MODEL})")
```

# Lab 4.1: LLM-Driven Code Optimization (C++ Performance)

**Aim**: To leverage Gemini 1.5 Flash's code generation capabilities to convert Python algorithms into high-performance, cross-platform C++ code and benchmark the execution speed improvement.

**Explanation**: This lab demonstrates a real-world application of LLMs in software engineering:

1. **Cross-Platform Logic**: The script automatically detects the host OS (Windows/Linux/Mac) and identifies available C++ compilers (cl.exe, g++, clang++).
2. **System Prompting**: We use a highly specific system prompt to force the model into "Optimizer Mode," prioritizing speed and exact numerical parity with Python.
3. **Dynamic Compilation**: The notebook compiles the generated C++ on-the-fly, runs it, and compares the output with the original Python version to ensure correctness.

```
# system message for code generation

system_message = "You are an assistant that reimplements Python code in high performance C++ f
system_message += "Respond only with C++ code; use comments sparingly and do not provide any e
system_message += "The C++ response needs to produce an identical output in the fastest possib
```

```
# prompt helper functions

def user_prompt_for(python):
    user_prompt = "Rewrite this Python code in C++ with the fastest possible implementation th
    user_prompt += "Respond only with C++ code; do not explain your work other than a few comm
    user_prompt += "Pay attention to number types to ensure no int overflows. Remember to #inc
    user_prompt += python
    return user_prompt

def messages_for(python):
    return [
        {"role": "system", "content": system_message},
        {"role": "user", "content": user_prompt_for(python)}
    ]
```

```python
# file output utility

def write_output(cpp, filename="optimized.cpp"):
    """Write C++ code to file, cleaning up code blocks"""
    code = cpp.replace("```cpp","").replace("```","")
    with open(filename, "w") as f:
        f.write(code)
```

```python
# Gemini optimization function

def optimize_gemini(python):
    """Generate C++ code using Gemini 1.5 Flash"""
    model = genai.GenerativeModel(GEMINI_MODEL)

    prompt = f"{system_message}\n\n{user_prompt_for(python)}"

    response = model.generate_content(
        prompt,
        generation_config=genai.types.GenerationConfig(
            temperature=0.1,
            max_output_tokens=2000,
        )
    )

    reply = response.text
    write_output(reply)
    return reply

def optimize_ollama_gemma(python):
    """Generate C++ code using local Gemma via Ollama"""
    print(f"Generating C++ with Ollama ({LOCAL_MODEL})...")
    prompt = f"{system_message}\n\n{user_prompt_for(python)}"
    try:
        response = ollama.generate(model=LOCAL_MODEL, prompt=prompt)
        reply = response['response']
        write_output(reply)
        print(reply)
        return reply
    except Exception as e:
        print(f"Ollama Error: {e}")
        return ""
```

```python
# Cross-platform C++ compilation utilities

VISUAL_STUDIO_2022_TOOLS = "C:\\Program Files\\Microsoft Visual Studio\\2022\\Community\\Commo
VISUAL_STUDIO_2019_TOOLS = "C:\\Program Files (x86)\\Microsoft Visual Studio\\2019\\BuildTools

simple_cpp_test = """
#include <iostream>

int main() {
    std::cout << "Hello";
    return 0;
}
"""
```

```
def run_cmd(command_to_run):
    """Execute a command and return its output"""
    try:
        run_result = subprocess.run(command_to_run, check=True, text=True, capture_output=True
        return run_result.stdout if run_result.stdout else "SUCCESS"
    except:
        return ""

def c_compiler_cmd(filename_base):
    """Auto-detect C++ compiler and return compilation command"""
    my_platform = platform.system()
    my_compiler = []

    try:
        with open("simple.cpp", "w") as f:
            f.write(simple_cpp_test)

        if my_platform == "Windows":
            # Try Visual Studio 2022
            if os.path.isfile(VISUAL_STUDIO_2022_TOOLS):
                if os.path.isfile("./simple.exe"):
                    os.remove("./simple.exe")
                compile_cmd = ["cmd", "/c", VISUAL_STUDIO_2022_TOOLS, "&", "cl", "simple.cpp"]
                if run_cmd(compile_cmd):
                    if run_cmd(["./simple.exe"]) == "Hello":
                        my_compiler = ["Windows", "Visual Studio 2022", ["cmd", "/c", VISUAL_S

            # Try Visual Studio 2019
            if not my_compiler:
                if os.path.isfile(VISUAL_STUDIO_2019_TOOLS):
                    if os.path.isfile("./simple.exe"):
                        os.remove("./simple.exe")
                    compile_cmd = ["cmd", "/c", VISUAL_STUDIO_2019_TOOLS, "&", "cl", "simple.c
                    if run_cmd(compile_cmd):
                        if run_cmd(["./simple.exe"]) == "Hello":
                            my_compiler = ["Windows", "Visual Studio 2019", ["cmd", "/c", VISU

            if not my_compiler:
                my_compiler=[my_platform, "Unavailable", []]

        elif my_platform == "Linux":
            # Try GCC
            if os.path.isfile("./simple"):
                os.remove("./simple")
            compile_cmd = ["g++", "simple.cpp", "-o", "simple"]
            if run_cmd(compile_cmd):
                if run_cmd(["./simple"]) == "Hello":
                    my_compiler = ["Linux", "GCC (g++)", ["g++", f"{filename_base}.cpp", "-o",

            # Try Clang
            if not my_compiler:
                if os.path.isfile("./simple"):
                    os.remove("./simple")
                compile_cmd = ["clang++", "simple.cpp", "-o", "simple"]
                if run_cmd(compile_cmd):
```

```
                    if run_cmd(["./simple"]) == "Hello":
                        my_compiler = ["Linux", "Clang++", ["clang++", f"{filename_base}.cpp",

            if not my_compiler:
                my_compiler=[my_platform, "Unavailable", []]

        elif my_platform == "Darwin":
            # Mac with optimized settings
            if os.path.isfile("./simple"):
                os.remove("./simple")
            compile_cmd = ["clang++", "-Ofast", "-std=c++17", "-march=armv8.5-a", "-mtune=appl
            if run_cmd(compile_cmd):
                if run_cmd(["./simple"]) == "Hello":
                    my_compiler = ["Macintosh", "Clang++", ["clang++", "-Ofast", "-std=c++17",

            if not my_compiler:
                my_compiler=[my_platform, "Unavailable", []]
    except:
        my_compiler=[my_platform, "Unavailable", []]

    if my_compiler:
        return my_compiler
    else:
        return ["Unknown", "Unavailable", []]
```

```
# Code execution functions

def execute_python(code):
    """Execute Python code and capture output"""
    try:
        output = io.StringIO()
        sys.stdout = output
        exec(code)
    finally:
        sys.stdout = sys.__stdout__
    return output.getvalue()

def execute_cpp(code):
    """Compile and execute C++ code"""
    write_output(code)
    compiler_info = c_compiler_cmd("optimized")

    if compiler_info[1] == "Unavailable":
        return "No C++ compiler available on this system"

    try:
        # Compile
        compile_result = subprocess.run(compiler_info[2], check=True, text=True, capture_outpu

        # Run
        if platform.system() == "Windows":
            run_cmd = ["./optimized.exe"]
        else:
            run_cmd = ["./optimized"]

        run_result = subprocess.run(run_cmd, check=True, text=True, capture_output=True)
```

```
        return run_result.stdout
    except subprocess.CalledProcessError as e:
        return f"Compilation/execution error:\n{e.stderr}"
```

```
# Streaming functions for real-time UI updates

def stream_gemini(python):
    """Stream Gemini responses for real-time UI"""
    model = genai.GenerativeModel(GEMINI_MODEL)
    prompt = f"{system_message}\n\n{user_prompt_for(python)}"

    response = model.generate_content(
        prompt,
        stream=True,
        generation_config=genai.types.GenerationConfig(
            temperature=0.1,
            max_output_tokens=2000,
        )
    )

    reply = ""
    for chunk in response:
        if chunk.text:
            reply += chunk.text
            yield reply.replace('```cpp\n','').replace('```','')

def stream_ollama(python, model_name):
    """Stream Ollama responses for real-time UI via direct request"""
    # Assuming OLLAMA_BASE_URL is defined as http://localhost:11434
    import requests
    url = "http://localhost:11434/api/generate"
    prompt = f"{system_message}\n\n{user_prompt_for(python)}"

    payload = {
        "model": model_name,
        "prompt": prompt,
        "stream": True,
        "options": {"temperature": 0.1}
    }

    reply = ""
    try:
        response = requests.post(url, json=payload, stream=True)
        for line in response.iter_lines():
            if line:
                chunk = json.loads(line)
                if 'response' in chunk:
                    reply += chunk['response']
                    yield reply.replace('```cpp\n','').replace('```','')
    except Exception as e:
        yield f"Ollama local error: {e}"
```

```
# Universal optimization function for all models

def optimize_code(python, model_choice):
```

```
    """Universal code optimization function supporting Gemini and Ollama"""
    if model_choice == "Gemini":
        result = stream_gemini(python)
    elif model_choice == "Ollama-Gemma":
        # Using the LOCAL_MODEL defined in initialization
        result = stream_ollama(python, LOCAL_MODEL)
    else:
        yield f"Unknown model: {model_choice}"
        return

    for stream_so_far in result:
        yield stream_so_far
```

```
# Sample program selector

def select_sample_program(sample_program):
    """Select pre-defined sample programs"""
    if sample_program == "pi":
        return pi_calculation
    elif sample_program == "complex":
        return python_complex
    else:
        return "# Type your Python program here\nprint('Hello World')"
```

```
# Auto-detect compiler and display system info

compiler_info = c_compiler_cmd("optimized")
print(f"Platform: {compiler_info[0]}")
print(f"Compiler: {compiler_info[1]}")
if compiler_info[2]:
    print(f"Compile command: {' '.join(compiler_info[2])}")
else:
    print("No C++ compiler detected")
```

```
# Basic Gradio interface

css = """
.python {background-color: #306998; color: white;}
.cpp {background-color: #005500; color: white;}
.info {background-color: #f0f0f0; padding: 10px; border-radius: 5px;}
"""

with gr.Blocks(css=css, title="Week 4 - Code Generator") as basic_ui:
    gr.Markdown("## 🪄 Lab 4.1: Python to C++ Code Generator")
    gr.Markdown("Convert Python code to optimized C++ using Gemini 1.5 Flash or Local Gemma 2'

    with gr.Row():
        python_input = gr.Textbox(
            label="Python code:",
            value=pi_calculation,
            lines=10,
            classes=["python"]
        )
        cpp_output = gr.Textbox(
            label="C++ code:",
```

```
            lines=10,
            classes=["cpp"]
        )

    with gr.Row():
        model_select = gr.Dropdown(
            ["Gemini", "Ollama-Gemma"],
            label="Select model",
            value="Gemini"
        )
        convert_btn = gr.Button("Convert to C++", variant="primary")

    with gr.Row():
        python_run = gr.Button("Run Python")
        cpp_run = gr.Button("Run C++", interactive=(compiler_info[1] != "Unavailable"))

    with gr.Row():
        python_result = gr.TextArea(label="Python result:", classes=["python"])
        cpp_result = gr.TextArea(label="C++ result:", classes=["cpp"])

    convert_btn.click(optimize_code, inputs=[python_input, model_select], outputs=[cpp_output]
    python_run.click(execute_python, inputs=[python_input], outputs=[python_result])
    cpp_run.click(execute_cpp, inputs=[cpp_output], outputs=[cpp_result])

print("Basic Gradio interface ready.")
```

```
# Launch basic interface

basic_ui.launch(inbrowser=True, share=False)
```

## Lab 4.2: Advanced Code Optimizer (System Prompting)

**Aim**: To build a sophisticated interface to optimize algorithms dynamically using specialized system prompts and cross-platform performance benchmarking.

**Explanation**: This lab expands on basic generation by incorporating:

1. **Cross-Platform Compilation Logic**: Automatic detection of system compilers.
2. **System Prompting for Performance**: Forcing the model to prioritize numerical parity and speed.
3. **Real-time Execution Benchmarking**: Comparing Python vs C++ execution times directly.

```
# Advanced Gradio interface with sample programs

with gr.Blocks(css=css, title="Week 4 - Advanced Code Optimizer") as advanced_ui:
    gr.Markdown("## 🚀 Lab 4.2: Advanced Code Optimizer")
    gr.Markdown("Full-featured interface with sample programs and performance benchmarking")

    with gr.Row():
        with gr.Column(scale=1):
            gr.Markdown("### System Information", classes=["info"])
            gr.Markdown(f"**Platform:** {compiler_info[0]}")
            gr.Markdown(f"**Compiler:** {compiler_info[1]}")
```

```
            sample_program = gr.Radio(
                ["pi", "complex", "custom"],
                label="Sample program",
                value="pi"
            )

            model_select = gr.Dropdown(
                ["Gemini", "Ollama-Gemma"],
                label="Select AI model",
                value="Gemini"
            )

        with gr.Column(scale=2):
            python_input = gr.Textbox(
                label="Python code:",
                value=pi_calculation,
                lines=15,
                classes=["python"]
            )

        with gr.Column(scale=2):
            cpp_output = gr.Textbox(
                label="Generated C++ code:",
                lines=15,
                classes=["cpp"]
            )

    with gr.Row():
        convert_btn = gr.Button("🔁 Convert to C++", variant="primary", size="lg")

    with gr.Row():
        python_run = gr.Button("🐍 Run Python")
        if compiler_info[1] != "Unavailable":
            cpp_run = gr.Button("⚡ Run C++")
        else:
            cpp_run = gr.Button("❌ No C++ Compiler", interactive=False)

    with gr.Row():
        python_result = gr.TextArea(
            label="Python execution result:",
            classes=["python"],
            max_lines=10
        )
        cpp_result = gr.TextArea(
            label="C++ execution result:",
            classes=["cpp"],
            max_lines=10
        )

    # Event handlers
    sample_program.change(select_sample_program, inputs=[sample_program], outputs=[python_inpu
    convert_btn.click(optimize_code, inputs=[python_input, model_select], outputs=[cpp_output]
    python_run.click(execute_python, inputs=[python_input], outputs=[python_result])
    cpp_run.click(execute_cpp, inputs=[cpp_output], outputs=[cpp_result])
```

```
print("Advanced Gradio interface ready.")
```

```
# Launch advanced interface

advanced_ui.launch(inbrowser=True, share=False)
```

# Performance Testing

Test and compare performance between Python and C++ implementations:

```
# Performance comparison function

def performance_test():
    """Run performance comparison between Python and C++ versions"""
    print("=== Performance Comparison ===")

    # Test Python performance
    print("\n1. Testing Python PI Calculation:")
    python_result = execute_python(pi_calculation)
    print(python_result)

    # Test Python complex algorithm
    print("\n2. Testing Python Complex Algorithm:")
    python_complex_result = execute_python(python_complex)
    print(python_complex_result)

    # Generate and test C++ version
    print("\n3. Generating C++ with Gemini:")
    try:
        cpp_code = optimize_gemini(python_complex)
        print("\n4. Testing C++ version:")
        cpp_result = execute_cpp(cpp_code)
        print(cpp_result)
    except Exception as e:
        print(f"Error in C++ generation/execution: {e}")

# Run performance test
# performance_test()  # Uncomment to run
print("Performance test function ready. Uncomment the last line to run it.")
```

---

# Reference: Generated C++ Examples

Below are examples of what the model typically generates for the benchmark algorithms.

## simple.cpp

```
#include <iostream>
```

```
int main() {
std::cout << "Hello";
return 0;
}
```

## optimized.cpp (Complex Algorithm)

```
#include <iostream>
#include <vector>
#include <chrono>
#include <limits>
#include <iomanip>

class LCG {
private:
uint64_t value;
const uint64_t a = 1664525;
const uint64_t c = 1013904223;
const uint64_t m = 1ULL << 32;

public:
LCG(uint64_t seed) : value(seed) {}

uint64_t next() {
value = (a * value + c) % m;
return value;
}
};

int64_t max_subarray_sum(int n, uint64_t seed, int min_val, int max_val) {
LCG lcg(seed);
std::vector<int> random_numbers(n);
for (int i = 0; i < n; ++i) {
random_numbers[i] = static_cast<int>(lcg.next() % (max_val - min_val + 1) +
min_val);
}

int64_t max_sum = std::numeric_limits<int64_t>::min();
int64_t current_sum = 0;
for (int i = 0; i < n; ++i) {
current_sum = std::max(static_cast<int64_t>(random_numbers[i]), current_sum +
random_numbers[i]);
max_sum = std::max(max_sum, current_sum);
}
return max_sum;
}

int64_t total_max_subarray_sum(int n, uint64_t initial_seed, int min_val, int
max_val) {
int64_t total_sum = 0;
LCG lcg(initial_seed);
for (int i = 0; i < 20; ++i) {
```

```
uint64_t seed = lcg.next();
total_sum += max_subarray_sum(n, seed, min_val, max_val);
}
return total_sum;
}

int main() {
int n = 10000;
uint64_t initial_seed = 42;
int min_val = -10;
int max_val = 10;

auto start_time = std::chrono::high_resolution_clock::now();
int64_t result = total_max_subarray_sum(n, initial_seed, min_val, max_val);
auto end_time = std::chrono::high_resolution_clock::now();

auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end_time -
start_time);

std::cout << "Total Maximum Subarray Sum (20 runs): " << result << std::endl;
std::cout << "Execution Time: " << std::fixed << std::setprecision(6) <<
duration.count() / 1e6 << " seconds" << std::endl;

return 0;
}
```

---

```
# Final summary and completion

print("="*60)
print("WEEK 4 COMPILED NOTEBOOK - SETUP COMPLETE")
print("="*60)
print()
print("✅ All code from day3.ipynb and day4.ipynb has been compiled")
print("✅ Updated to use Gemini 1.5 Flash")
print("✅ Added Ollama support with Gemma 2:2b model")
print("✅ Cross-platform C++ compilation support included")
print("✅ Both basic and advanced Gradio interfaces ready")
print("✅ Performance testing utilities included")
print("✅ All original C++ examples preserved")
print()
print("🚀 Ready to generate high-performance C++ code from Python!")
print("📊 Use the Gradio interfaces above to test the functionality")
print()
print("Models configured:")
print(f"  • Gemini: {GEMINI_MODEL}")
print(f"  • Ollama Gemma: {LOCAL_MODEL}")
print()
print(f"System: {compiler_info[0]} with {compiler_info[1]} compiler")
print("="*60)
```

---

# Instructor's Evaluation & Lab Summary

## Assessment Criteria

1. **Technical Implementation**: Adherence to the lab objectives and code functionality.
2. **Logic & Reasoning**: Clarity in the explanation of the underlying AI principles.
3. **Best Practices**: Use of secure environment variables and structured prompts.

**Lab Completion Status: Verified Focus Area**: Language Modelling & Deep Learning Systems.

# Week 5 Lab Manual

## Foundations of Deep Learning & AI Functionality

**Instructor Note**: This lab manual provides the aim, code, and explanation for each practical task. Focus on the architectural patterns and the transition from theoretical concepts to functional AI implementations.

---

# Week 5: RAG Systems & Semantic Search

Welcome to Week 5! This week we dive into **Retrieval Augmented Generation (RAG)**. Instead of just asking an LLM to "hallucinate" an answer, we provide it with specific, relevant facts from a database to ensure accuracy and reduce errors.

## Weekly Table of Contents



## Learning Objectives

- **Context Injection**: Learn how to manually feed data into LLMs.
- **Vector Embeddings**: Understand how text is converted into numbers (vectors).
- **Vector Databases**: Use **ChromaDB** and **FAISS** to store and retrieve information.
- **Retrieval Chains**: Build conversational systems that remember history and use external knowledge.
- **Advanced RAG**: Optimize chunking, metadata, and retrieval parameters.

---

## Configuration & API Setup

We'll start by ensuring our environment is ready. We use Gemini 1.5 Flash as our primary cloud model and Gemma 2:2b for local tasks.

# Lab 5.1: Basic Context Injection

**Aim**: To understand the fundamental logic of "Augmentation" by building a manual retrieval system that injects text from local files into a prompt.

**Explanation**: This lab breaks down the "R" in RAG:

1. **Retrieval**: We search a dictionary representing a knowledge base for keywords.
2. **Augmentation**: The retrieved text is prepended to the user query as "Context".
3. **Generation**: Gemini 1.5 Flash uses this context to answer questions it wasn't originally trained on.

```
# 📦 WEEK 5 INITIALIZATION
import os
import glob
import shutil
import google.generativeai as genai
import ollama
from dotenv import load_dotenv
from IPython.display import Markdown, display
import numpy as np
import plotly.graph_objects as go
from sklearn.manifold import TSNE
import gradio as gr

# RAG & LANGCHAIN LIBS
from langchain_google_genai import ChatGoogleGenerativeAI, GoogleGenerativeAIEmbeddings
from langchain_chroma import Chroma
from langchain_community.document_loaders import PyPDFLoader, DirectoryLoader, TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter, CharacterTextSplitter
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationalRetrievalChain
from langchain_core.callbacks import StdOutCallbackHandler

# --- CONFIGURATION ---
load_dotenv(override=True)
MODEL = "gemini-1.5-flash"
LOCAL_MODEL = "gemma2:2b"

# Ensure API Key is available
GEMINI_API_KEY = os.getenv("GOOGLE_API_KEY") or os.getenv("GEMINI_API_KEY")
if GEMINI_API_KEY:
    os.environ["GOOGLE_API_KEY"] = GEMINI_API_KEY
    genai.configure(api_key=GEMINI_API_KEY)
else:
    print("⚠️ WARNING: GEMINI_API_KEY not found.")

# Create folders for knowledge base
for folder in ["knowledge-base/products", "knowledge-base/employees", "knowledge-base/contract
    os.makedirs(folder, exist_ok=True)

print(f"✅ Week 5 Ready: Using {MODEL}")
```

```
# Define content for our documents
kb_files = [
    # Products
    ("knowledge-base/products/carllm.md", "# CarLLM\nCarLLM is our premium AI-powered auto ins
    ("knowledge-base/products/homellm.md", "# HomeLLM\nHomeLLM automates home insurance claims

    # Employees
    ("knowledge-base/employees/ceo.md", "# Alex Lancaster\nAlex Lancaster is the CEO and found
    ("knowledge-base/employees/cto.md", "# Sarah Avery\nSarah Avery is the CTO. She is a pione
```

```
    # Company Info
    ("knowledge-base/company/about.md", "# About Insurellm\nFounded in 2018, Insurellm is head
    ("knowledge-base/company/awards.md", "# Awards\nInsurellm won the 'Insurance Innovation of
]

# Write the files
for filepath, content in kb_files:
    with open(filepath, 'w', encoding='utf-8') as f:
        f.write(content)

print(f"✅ Created {len(kb_files)} knowledge base files in the 'knowledge-base/' directory.")
```

## Load Knowledge Base - Employees

Loading employee data from markdown files into a dictionary for simple keyword-based context retrieval.

```
# 1. PRE-DEFINED CONTEXT DICTIONARY
context = {
    'Lancaster': 'Alex Lancaster is the CEO of Insurellm. He founded the company in 2018 and h
    'Avery': 'Sarah Avery is the CTO of Insurellm. She leads the technical team and oversees a
    'CarLLM': "CarLLM is Insurellm's flagship AI-powered car insurance product. It won the pre
    'HomeLLM': 'HomeLLM is our home insurance AI assistant that helps customers understand cov
    'HealthLLM': "HealthLLM is a new project in the R&D stage, focused on personal health insu
    'PolicyLLM': "PolicyLLM helps users understand the jargon in their existing insurance poli
}

# 2. CONTEXT RETRIEVAL LOGIC
def get_relevant_context(message):
    """Simple keyword-based context retriever"""
    relevant_context = []
    message_lower = message.lower()
    for key, text in context.items():
        if key.lower() in message_lower:
            relevant_context.append(text)
    return relevant_context

def add_context(message):
    """Augments user message with retrieved context"""
    relevant_context = get_relevant_context(message)
    if relevant_context:
        augmented = "The following additional context might be relevant:\n\n"
        for fact in relevant_context:
            augmented += f"- {fact}\n"
        return f"{augmented}\nQuestion: {message}"
    return message

# 3. CHAT GENERATION
system_message = "You are an expert on Insurellm. Give brief, accurate answers based ONLY on p

def chat(message, history):
    prompt = add_context(message)
    model = genai.GenerativeModel(model_name=MODEL, system_instruction=system_message)
```

```
    response = model.generate_content(prompt)
    return response.text

print("✅ Manual RAG functions ready.")
```

## Gemini Chat Function

Main chat function that processes user messages, adds context, and generates responses using Gemini 1.5 Flash.

## Lab 5.1: Gradio Chat Interface

A simple RAG prototype using keyword-based context matching with Gemini 1.5 Flash.

```
# Launch Day 1 Gradio interface
view = gr.ChatInterface(chat, type="messages").launch()
```

# Lab 5.2: Automated Document Loading & Splitting

**Aim**: To automate the ingestion and transformation of raw document folders into manageable text chunks for RAG pipelines.

**Explanation**: Manual context injection doesn't scale. In this lab, we use LangChain to load a directory of markdown files and split them into chunks using the `RecursiveCharacterTextSplitter` to maintain semantic context within sub-documents.

## Document Loading with LangChain

Loading documents from our knowledge base using LangChain's DirectoryLoader with proper metadata.

```
# 1. DOCUMENT LOADING
loader = DirectoryLoader("knowledge-base", glob="**/*.md", loader_cls=TextLoader)
documents = loader.load()

# Assign metadata for categorization
for doc in documents:
    path_parts = doc.metadata['source'].split(os.sep)
    if len(path_parts) >= 2:
        doc.metadata['doc_type'] = path_parts[-2]
    else:
        doc.metadata['doc_type'] = 'general'

# 2. TEXT SPLITTING (RecursiveCharacterTextSplitter)
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=100,
    separators=["\n\n", "\n", " ", ""]
)
chunks = text_splitter.split_documents(documents)
```

```
print(f"✅ Loaded {len(documents)} documents.")
print(f"✅ Split into {len(chunks)} chunks.")
if chunks:
    print(f"Sample chunk types: {set(c.metadata['doc_type'] for c in chunks)}")
```

# Lab 5.3: Vector Databases (Chroma) & Semantic Search

**Aim**: To implement persistent vector storage and semantic retrieval using ChromaDB and Google Generative AI Embeddings.

**Explanation**: We store processed chunks in a vector database to perform semantic search instead of simple keyword matching. This allows the AI to find relevant context based on 'meaning' rather than just exact word matches.

```
# 1. EMBEDDINGS SETUP
embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")

# 2. VECTOR DATABASE (CHROMA)
db_path = "insurellm_vector_db"

if os.path.exists(db_path):
    shutil.rmtree(db_path)

vectorstore = Chroma.from_documents(
    documents=chunks,
    embedding=embeddings,
    persist_directory=db_path
)

print(f"✅ Vectorstore created with {len(chunks)} chunks using {embeddings.model}.")
```

```
# Get one vector and find how many dimensions it has
collection = vectorstore._collection
sample_embedding = collection.get(limit=1, include=["embeddings"])["embeddings"][0]
dimensions = len(sample_embedding)
print(f"The vectors have {dimensions:,} dimensions")
```

# Lab 5.4: Building a Conversational Retrieval Chain

**Aim**: To assemble a complete, stateful Retrieval-Augmented Generation (RAG) system with conversational memory.

**Explanation**: In this final lab, we combine document splitting, vector storage, and memory. The `ConversationalRetrievalChain` manages the flow: searching for context, merging with history, and generating informed responses.

```
# 1. SETUP LLM & RETRIEVER
llm = ChatGoogleGenerativeAI(temperature=0.7, model=MODEL)
```

```
retriever = vectorstore.as_retriever(search_kwargs={"k": 3})

# 2. SETUP CONVERSATION MEMORY
memory = ConversationBufferMemory(memory_key='chat_history', return_messages=True)

# 3. CREATE CONVERSATIONAL RETRIEVAL CHAIN
rag_chain = ConversationalRetrievalChain.from_llm(
    llm=llm,
    retriever=retriever,
    memory=memory,
    verbose=False
)

def chat_rag(message, history):
    # This function is used by Gradio to maintain its own history if needed,
    # but the chain's memory also tracks it.
    result = rag_chain.invoke({"question": message})
    return result["answer"]

print(f"✅ Conversational RAG Chain ready with {MODEL}.")
```

```
# Launch the final Conversational RAG interface
view_rag = gr.ChatInterface(chat_rag, type="messages").launch(inbrowser=True)
```

---

# Instructor's Evaluation & Lab Summary

## Assessment Criteria

1. **Technical Implementation**: Adherence to the lab objectives and code functionality.
2. **Logic & Reasoning**: Clarity in the explanation of the underlying AI principles.
3. **Best Practices**: Use of secure environment variables and structured prompts.

**Lab Completion Status: Verified Focus Area**: Language Modelling & Deep Learning Systems.

# Week 6 Lab Manual

## Foundations of Deep Learning & AI Functionality

**Instructor Note**: This lab manual provides the aim, code, and explanation for each practical task. Focus on the architectural patterns and the transition from theoretical concepts to functional AI implementations.

---

# Week 6: Local LLM Deep Dive & Quantization

Welcome to Week 6. After exploring RAG systems, we now focus on the engine that powers many of these systems locally: **Ollama** and the concept of **Quantization**.

## Weekly Table of Contents



## Learning Objectives

1. Understand the benefits and trade-offs of Local vs Cloud LLMs.
2. Learn the basics of LLM Quantization (4-bit, 8-bit).
3. Master the Ollama Python SDK for advanced control.
4. Build a local benchmarking tool to measure tokens-per-second (TPS).

---

# 6.1 Local LLMs & The VRAM Problem

Cloud models (like Gemini) have trillions of parameters and require massive server farms. To run these on a consumer laptop, we use:

- **Smaller Architectures:** Like Gemma 2 (2B or 9B parameters).
- **Quantization:** Reducing the precision of weights to save memory.

```
# 📦 WEEK 6 INITIALIZATION
import os
import time
import ollama
from dotenv import load_dotenv
from IPython.display import Markdown, display
```

```python
# --- CONFIGURATION ---
load_dotenv(override=True)
LOCAL_MODEL = "gemma2:2b" # Updated to gemma2 for better performance

# Help with Ollama connection
try:
    ollama.list()
    print(f"✅ Ollama is running.")
except Exception as e:
    print(f"❌ ERROR: Ollama is not running. Please start the Ollama application.")

print(f"✅ Week 6 Ready. Local Model: {LOCAL_MODEL}")
```

# Lab 6.1: Advanced Local Control with Ollama SDK

**Aim**: To programmatically interact with local LLMs using the Ollama Python SDK, enabling streaming responses and custom system instructions.

**Explanation**: This lab demonstrates how to move beyond basic Ollama CLI commands:

1. **SDK Integration**: We use the `ollama` Python library to initiate chat sessions directly from code.
2. **Streaming**: Implementing `stream=True` to improve perceived latency by displaying tokens as they are generated.
3. **Custom Modelfiles**: Showing how to create new model variants (e.g., `expert-gemma`) with baked-in system instructions and temperature parameters.

```python
# Custom system prompt for a technical expert
system_prompt = "You are a senior C++ engineer. Give concise, low-level answers."

def chat_stream(prompt):
    messages = [
        {'role': 'system', 'content': system_prompt},
        {'role': 'user', 'content': prompt}
    ]

    response = ollama.chat(model=LOCAL_MODEL, messages=messages, stream=True)

    for chunk in response:
        print(chunk['message']['content'], end='', flush=True)

# Test the stream
chat_stream("How do I optimize a for loop for vectorization?")
```

# Lab 6.2: Local LLM Benchmarking (Speed vs Model Size)

**Aim**: To build a diagnostic tool that measures the performance of local LLMs in terms of Tokens Per Second (TPS) and response latency.

**Explanation**: Performance metrics are critical for local deployment:

1. **Timer Logic**: We capture `start_time` and `end_time` around the generation call.
2. **Token Counting**: Utilizing Ollama's `eval_count` metadata to get the exact number of tokens generated.
3. **Efficiency Analysis**: By running this across different models (2B vs 7B), students can visualize the direct correlation between parameter count and inference speed on their specific hardware.

```
def benchmark_model(model_name, prompt):
    print(f"\nBenchmarking {model_name}...")
    start_time = time.time()

    try:
        response = ollama.generate(model=model_name, prompt=prompt)

        end_time = time.time()
        total_time = end_time - start_time

        tokens = response['eval_count']
        tps = tokens / total_time

        print(f"Total Time: {total_time:.2f}s")
        print(f"Total Tokens: {tokens}")
        print(f"Tokens Per Second: {tps:.2f} tokens/s")
    except Exception as e:
        print(f"Error benchmarking {model_name}: {e}. Ensure the model is downloaded.")

# Run benchmark
benchmark_model(LOCAL_MODEL, 'Explain quantum physics in one paragraph.')
```

# Lab 6.3: Understanding Quantization Levels & Modelfiles

**Aim**: To understand how model weight precision impacts performance and how to create custom local model instances using Ollama Modelfiles.

**Explanation**: Ollama allows us to customize model behavior permanently:

1. **Quantization**: We discuss the math behind FP32 vs INT4. A 4-bit model typically offers the best balance for consumer hardware.
2. **Modelfiles**: Similar to Dockerfiles, these allow us to set parameters like `temperature` and `system_prompt` so they are baked into a new model name (e.g., `expert-gemma` ).

```
# Step 4: Programmatic Modelfile Creation
# We will create a model called 'expert-gemma' that is specifically tuned for system administr

modelfile = f"""
FROM {LOCAL_MODEL}
PARAMETER temperature 0.2
SYSTEM You are an expert Linux System Administrator. Answer only with technical commands.
"""

print("Creating custom model 'expert-gemma'...")
```

```
# ollama.create(model='expert-gemma', modelfile=modelfile)

# Test the custom model
print("\nTesting 'expert-gemma':")
# response = ollama.generate(model='expert-gemma', prompt="How do I check open ports on Ubuntu
# print(response['response'])
print("Note: ollama.create is commented out to prevent repeated model creation during walkthro
```

```
# Final summary and completion

print("="*60)
print("WEEK 6 COMPILED NOTEBOOK - SETUP COMPLETE")
print("="*60)
print()
print("✅ Local LLM environment configured")
print("✅ Lab 6.1: Streaming SDK control ready")
print("✅ Lab 6.2: Performance benchmarking tool ready")
print("✅ Lab 6.3: Custom Modelfile generation logic ready")
print()
print("🚀 Ready to explore local inference and quantization!")
print("📊 Use benchmark_model() to compare different local models.")
print("="*60)
```

---

# Instructor's Evaluation & Lab Summary

## Assessment Criteria

1. **Technical Implementation**: Adherence to the lab objectives and code functionality.
2. **Logic & Reasoning**: Clarity in the explanation of the underlying AI principles.
3. **Best Practices**: Use of secure environment variables and structured prompts.

**Lab Completion Status: Verified Focus Area**: Language Modelling & Deep Learning Systems.

# Week 7 Lab Manual

## Foundations of Deep Learning & AI Functionality

**Instructor Note**: This lab manual provides the aim, code, and explanation for each practical task. Focus on the architectural patterns and the transition from theoretical concepts to functional AI implementations.

---

# Week 7: Advanced RAG & Reasoning

# Solving Complex Queries with Gemini 2.5 Flash

## Weekly Table of Contents



## Learning Objectives

Basic RAG finds text; Advanced RAG finds *answers*. This week we master:

1. **Multi-Query Retrieval**: Using Gemini to rewrite user queries into multiple variations.
2. **Contextual Compression**: Filtering out "noise" from retrieved documents.
3. **Reranking**: Using a Rerank model (or Gemini's reasoning) to sort context by relevance.
4. **Long-Context RAG**: Leveraging Gemini's 1.5M token window for "needle-in-a-haystack" tasks.

---

## 7.1 Deep Learning: Reranking & Compression

### The Retrieval Gap

Sometimes semantic search finds a document that is *semantically* similar but *factually* irrelevant.

- **The Solution**: A "Reranker" acts as a second stage. It looks at the Top 20 results from semantic search and carefully scores them against the question.

### Contextual Compression

LLMs have a context window, but filling it with 10,000 words of "maybe relevant" text can confuse the model. We use compression to extract only the sentences that actually help answer the question.

---

```
# 📦 WEEK 7 INITIALIZATION
import os
from dotenv import load_dotenv
import google.generativeai as genai
from IPython.display import Markdown, display

# LANGCHAIN & LANGGRAPH
from langchain_google_genai import ChatGoogleGenerativeAI, GoogleGenerativeAIEmbeddings
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor
from langgraph.graph import END, StateGraph
from typing import List, TypedDict

# --- CONFIGURATION ---
load_dotenv(override=True)
MODEL = "gemini-1.5-flash"

# Ensure API Key is available
GEMINI_API_KEY = os.getenv("GOOGLE_API_KEY") or os.getenv("GEMINI_API_KEY")
if GEMINI_API_KEY:
    os.environ["GOOGLE_API_KEY"] = GEMINI_API_KEY
    genai.configure(api_key=GEMINI_API_KEY)
else:
    print("⚠️ WARNING: GEMINI_API_KEY not found.")

# Cloud Model Instance
llm_cloud = ChatGoogleGenerativeAI(model=MODEL, temperature=0)

print(f"✅ Week 7 Ready: Advanced RAG with {MODEL}")
```

# Lab 7.1: Multi-Query Retrieval & Gemini Re-Ranking

**Aim**: To overcome the limitations of standard distance-based similarity search by expanding a single user query into multiple perspectives and re-ranking the results for maximum precision.

**Explanation**: This lab implements two critical "Advanced RAG" patterns:

1. **Query Expansion**: We use an LLM to generate variations of the user's question. This ensures that even if the user uses non-technical language, at least one variation will likely match the technical index.
2. **Re-Ranking**: We take raw results and apply a second "reasoning" pass (using Gemini 1.5 Flash's massive context window) to ensure factual alignment before the final answer generation.

```
# 1. MULTI-QUERY RETRIEVAL (Query Expansion)
# We use Gemini to rewrite the question into 5 variations to catch different semantic angles.
```

```
multi_query_prompt = ChatPromptTemplate.from_template("""
You are an AI assistant tasked with generating five different versions of the given user
question to retrieve relevant documents from a vector database. My goal is to overcome
limitations of standard distance-based search.

Original question: {question}

Provide 5 variations, one per line:""")

generate_queries = (
    multi_query_prompt
    | llm_cloud
    | StrOutputParser()
    | (lambda x: [q.strip() for q in x.split("\n") if q.strip()])
)

# Test query expansion
test_question = "What is the impact of salt on climate change?"
# queries = generate_queries.invoke({"question": test_question})
# print("--- Generated Variations ---")
# for i, q in enumerate(queries): print(f"{i+1}: {q}")
```

```
# 2. GEMINI RE-RANKER
# Once we have results, we use Gemini's reasoning to pick the most factually aligned one.

rerank_prompt = ChatPromptTemplate.from_template("""
You are an expert re-ranker. Given the following user question and a list of retrieved documer
rank the documents from most relevant to least relevant. Return only the index numbers in orde

Question: {question}

Documents (Index: Content):
{documents}

Ranked Indices (comma separated):""")

reranker_chain = rerank_prompt | llm_cloud | StrOutputParser()

# Simulated retrieval results
docs = [
    "0: Salt mining in the Himalayas has increased recently.",
    "1: Increased ocean salinity can affect thermohaline circulation and global climate patter
    "2: Salt is commonly used for de-icing roads in winter.",
    "3: Soil salinization is a major threat to global food security but is indirectly linked t
]

# result = reranker_chain.invoke({"question": test_question, "documents": "\n".join(docs)})
# print(f"Ranked Indices: {result}")
```

# 1. Beyond Basic RAG: The Need for Advanced Orchestration

Basic RAG (Retrieval-Augmented Generation) often fails when:

1. **Poor Query Formulation:** The user's question doesn't match the technical language of the documents.
2. **Low Precision Retrieval:** The top-k results contain irrelevant noise that confuses the LLM.
3. **Complex Reasoning:** The answer requires synthesizing information from multiple distant parts of the dataset.

### Advanced Techniques:

- **Multi-Query Retrieval:** Using an LLM to generate multiple versions of a query to capture different semantic angles.
- **Re-Ranking:** Retrieving a large set of documents (e.g., top 20) and using a more powerful model (or cross-encoder) to pick the most relevant 3.
- **Cyclic Workflows (Agents):** Instead of a linear pipeline, we use a loop where the model can "reflect" on whether the retrieved information is sufficient.

---

## 2. Introducing LangGraph

While LangChain is great for chains, **LangGraph** allows us to build stateful, multi-actor applications with loops. It is the core framework for building robust AI Agents.

### Core Concepts:

1. **State:** A shared object that evolves as nodes execute.
2. **Nodes:** Python functions that take the state and return an update.
3. **Edges:** Define the flow between nodes (can be conditional).

## Lab 7.2: Building a Self-Reflective Agent with LangGraph

**Aim**: To transition from linear RAG chains to cyclic, stateful agentic workflows that can "reflect" on the quality of retrieved data and re-try queries if the initial results are insufficient.

**Explanation**: This represents the cutting edge of LLM application development:

1. **LangGraph Orchestration**: We define a "State" that tracks the conversation and a "Graph" that defines the logic flow (e.g., Search -> Evaluate -> Answer).
2. **Self-Correction**: The agent includes a "Node" that checks if the retrieved context actually answers the question. If not, it loops back to perform a broader search.
3. **Hybrid Reasoning**: Uses Gemini 1.5 Flash for the complex planning/reflection steps.

```
# 🪄 Lab 7.2: SELF-REFLECTIVE AGENT
# Transitioning from linear chains to cycles.

# 1. Define the Graph State
```

```python
class GraphState(TypedDict):
    question: str
    generation: str
    documents: List[str]

# 2. Define the Graph Nodes (Logic Units)
def retrieve(state: GraphState):
    print("---[RETRIEVING]---")
    # Simulate DB search
    return {"documents": ["Salt concentration affects ocean density."], "question": state["que

def grade_documents(state: GraphState):
    print("---[GRADING]---")
    # In a full app, we'd use llm_cloud to check if the doc actually answers the question
    if "Salt" in state["documents"][0]:
        return "generate"
    else:
        return "rewrite"

def generate(state: GraphState):
    print("---[GENERATING]---")
    # Use llm_cloud to generate final response based on docs
    response = llm_cloud.invoke(f"Use these docs: {state['documents']} to answer: {state['ques
    return {"generation": response.content}

# 3. Assemble the StateGraph
workflow = StateGraph(GraphState)

# Add Nodes
workflow.add_node("retrieve", retrieve)
workflow.add_node("generate", generate)

# Define Logic Flow
workflow.set_entry_point("retrieve")

# Add conditional routing from 'retrieve' node
workflow.add_conditional_edges(
    "retrieve",
    grade_documents,
    {
        "generate": "generate",
        "rewrite": "retrieve" # Loop back if info is missing
    }
)

workflow.add_edge("generate", END)

# 4. Compile and Run
app = workflow.compile()

# Test Execution
inputs = {"question": "How does salt affect climate?"}
print("--- STARTING LANGGRAPH EXECUTION ---")
for output in app.stream(inputs):
    for key, value in output.items():
        print(f"Node completed: {key}")
```

```
        if "generation" in value:
            print(f"\nFinal Answer: {value['generation']}")
```

---

# Instructor's Evaluation & Lab Summary

## Assessment Criteria

1. **Technical Implementation**: Adherence to the lab objectives and code functionality.
2. **Logic & Reasoning**: Clarity in the explanation of the underlying AI principles.
3. **Best Practices**: Use of secure environment variables and structured prompts.

**Lab Completion Status: Verified Focus Area**: Language Modelling & Deep Learning Systems.

# Week 8 Lab Manual

## Foundations of Deep Learning & AI Functionality

**Instructor Note**: This lab manual provides the aim, code, and explanation for each practical task. Focus on the architectural patterns and the transition from theoretical concepts to functional AI implementations.

---

# Week 8: Model Evaluation & Monitoring

Welcome to Week 8. As we move towards production (Week 9), we must answer the most critical question: **"How do we know if our model / agent is actually good?"**

## Weekly Table of Contents



- LLM-as-a-Judge Logic
- Automated Scoring with JSON Parsers
- The RAG Triad: Faithfulness, Relevance, and Context

## Learning Objectives

1. Understand the concept of **LLM-as-a-Judge**.
2. Learn how to define evaluation metrics for RAG and Agents.
3. Use Gemini 1.5 Flash to evaluate the outputs of our local Ollama models.
4. Build an automated evaluation pipeline.

---

# 8.1 Why Evaluation is Hard

Unlike traditional software, LLM outputs are non-deterministic and text-based. Checking for "exact matches" is useless. Instead, we use:

- **Benchmarks:** Static datasets (MMLU, GSM8K).
- **Human-in-the-loop:** Expensive and slow.
- **LLM Judges:** Using a superior model (Gemini 1.5 Flash) to grade a smaller model (Gemma).

---

```
# 📦 WEEK 8 INITIALIZATION
import os
from dotenv import load_dotenv
from IPython.display import Markdown, display

# LANGCHAIN EVALUATION
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_community.llms import Ollama
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import JsonOutputParser

# --- CONFIGURATION ---
load_dotenv(override=True)
MODEL = "gemini-1.5-flash"
LOCAL_MODEL = "gemma2:2b"

# The Judge (Cloud)
judge_llm = ChatGoogleGenerativeAI(model=MODEL, temperature=0)

# The Model being tested (Local or lower-capability)
test_llm = Ollama(model=LOCAL_MODEL)

print(f"✅ Week 8 Ready: Evaluation using {MODEL} as Judge.")
```

# Lab 8.1: Building an LLM-as-a-Judge Pipeline

**Aim**: To establish an automated quality assurance workflow where a high-capability model (Gemini 1.5 Flash) assesses the performance of a local "student" model (Gemma 2).

**Explanation**: This lab implements the "LLM-as-a-Judge" pattern:

1. **Metric Definition**: We define a 1-5 scale for "Faithfulness" and "Accuracy."
2. **Structured Evaluation**: Using a `JsonOutputParser`, we force the judge to provide both a quantitative score and a qualitative reason.
3. **Benchmarking**: This allows developers to iterate on prompts or RAG settings and see a numerical improvement in model performance without manual review.

```
from langchain_core.output_parsers import JsonOutputParser

eval_prompt = ChatPromptTemplate.from_template("""
You are an unbiased evaluator. Grade the 'Student Response' based on its 'Reference Facts'.
Give a score from 1 to 5 (5 being perfectly accurate) and a brief reasoning.

Reference Facts: {reference}
Student Response: {response}

Return your answer in the following JSON format:
{{
    "score": int,
    "reasoning": "string"
}}
```

```
""")

eval_chain = eval_prompt | judge_llm | JsonOutputParser()

# Test Case
reference_fact = "The capital of France is Paris. It has the Eiffel Tower."
student_response = "Paris is the capital of France and is known for the Eiffel Tower."

print("Running LLM-as-a-Judge Evaluation...")
result = eval_chain.invoke({"reference": reference_fact, "response": student_response})
print(f"Score: {result['score']}/5")
print(f"Reasoning: {result['reasoning']}")
```

## Lab 8.2: Evaluating RAG with the "RAG Triad"

**Aim**: To implement the industry-standard "RAG Triad" evaluation framework to identify specific points of failure in a retrieval-augmented generation system.

**Explanation**: We break down RAG performance into three distinct components:

1. **Context Relevance**: Checks if the retriever found the right information.
2. **Faithfulness**: Ensures the generator didn't hallucinate or add outside knowledge.
3. **Answer Relevance**: Verifies that the final output actually answers the user's specific question. By measuring these separately, we can determine whether to fix the "Retrieval" (Vector DB) or the "Generation" (Prompting/Model).

```
def calculate_relevance(question, context):
    """
    Measures Context Relevance (Part of the RAG Triad)
    """
    prompt = f"""
    On a scale of 0 to 1, how relevant is the context below to the question?
    Return ONLY a numerical value.

    Question: {question}
    Context: {context}
    Relevance Score:"""

    response = judge_llm.invoke(prompt)
    try:
        # Extracting numerical value from potential text
        score_text = response.content.strip()
        score = float(score_text)
        return score
    except:
        return 0.0

# Example Scenario
test_question = "What is the speed of light?"
test_context = "The speed of light in a vacuum is exactly 299,792,458 metres per second."

print(f"Testing RAG Triad - Context Relevance...")
```

```
score = calculate_relevance(test_question, test_context)
print(f"Context Relevance Score: {score}")
```

```
# Final summary and completion

print("="*60)
print("WEEK 8 COMPILED NOTEBOOK - SETUP COMPLETE")
print("="*60)
print()
print("✅ Evaluation environment initialized using Gemini 1.5 Flash as Judge")
print("✅ Lab 8.1: LLM-as-a-Judge pipeline with JSON parsing ready")
print("✅ Lab 8.2: RAG Triad evaluation logic ready")
print()
print("🚀 Ready to scientifically measure model performance!")
print("📊 Use eval_chain.invoke() to test your student models.")
print("="*60)
```

---

# Instructor's Evaluation & Lab Summary

## Assessment Criteria

1. **Technical Implementation**: Adherence to the lab objectives and code functionality.
2. **Logic & Reasoning**: Clarity in the explanation of the underlying AI principles.
3. **Best Practices**: Use of secure environment variables and structured prompts.

**Lab Completion Status: Verified Focus Area**: Language Modelling & Deep Learning Systems.

# Week 9 Lab Manual

## Foundations of Deep Learning & AI Functionality

**Instructor Note**: This lab manual provides the aim, code, and explanation for each practical task. Focus on the architectural patterns and the transition from theoretical concepts to functional AI implementations.

---

# Week 9: Model Specialization, Production & Capstone Project

# From Notebooks to Production-Ready AI Systems

## Weekly Table of Contents



- Synthetic Data Generation for Fine-Tuning
- FastAPI Integration with Server-Sent Events (SSE)
- Real-time Performance Optimization

## Learning Objectives

This final week focuses on transitioning from a "working prototype" to a "production system." You will cover:

1. **Advanced Model Tuning**: Deep dive into PEFT, LoRA, and the logic of Synthetic Data Generation.
2. **Streaming & UI Optimization**: Implementing real-time feedback loops for better user experience.
3. **Production Architectures**: Deploying LLM logic via **FastAPI** with background tasks and streaming.
4. **Capstone Integration**: Building a unified "Autonomous Researcher" agent.

---

## 9.1 Theory: Fine-Tuning vs. RAG (The Final Verdict)

By now, you've seen RAG (Week 5) and Prompt Engineering (Week 4). Why bother with Fine-Tuning?

| Feature | Prompting / RAG | Fine-Tuning |
|---|---|---|
| **Knowledge Update** | Easy (Update Vector DB) | Hard (Requires retraining) |

| Feature | Prompting / RAG | Fine-Tuning |
|---|---|---|
| **New Task Adaptation** | Good | Excellent |
| **Cost to Latency** | High (Long context) | Low (Shorter prompts) |
| **Steerability** | Variable | Very High (Consistent tone/format) |

**Modern Convergence**: Most production systems use a **Hybrid Approach**:

- Fine-tune for **style, format, and reasoning**.
- RAG for **up-to-date facts and data**.

# Lab 9.1: Specialized Dataset Lab

**Aim**: To generate a high-quality, synthetic training dataset using a "Teacher" model (Gemini 1.5 Flash) to prepare a smaller "Student" model (Gemma 2) for specialized tasks through fine-tuning.

**Explanation**: This lab demonstrates the first step in the fine-tuning pipeline:

1. **Structured Generation**: We use `Pydantic` to ensure the teacher model returns perfectly formatted JSON training pairs.
2. **Specialization**: By providing industry-specific instructions (e.g., Legal Contract Analysis), we generate data that captures professional jargon and complex risk assessment patterns.
3. **JSONL Export**: The output is saved in `.jsonl` format, the standard for training modern LLMs via techniques like LoRA.

```
# 📦 WEEK 9 INITIALIZATION
import os
import json
import asyncio
from typing import List
from pydantic import BaseModel, Field
from dotenv import load_dotenv

# Load environment variables
load_dotenv(override=True)

# Standardized Model Definitions
MODEL = "gemini-1.5-flash"
LOCAL_MODEL = "gemma2:2b"

# API Clients
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.prompts import PromptTemplate

# Initialize Teacher Model
llm = ChatGoogleGenerativeAI(model=MODEL, temperature=0.7)

# --- Lab 9.1: Specialized Dataset Lab ---
```

```python
# Define the structure of our training data
class TrainingExample(BaseModel):
    instruction: str = Field(description="The user query or prompt")
    response: str = Field(description="The ideal expert assistant response")

class Dataset(BaseModel):
    examples: List[TrainingExample]

parser = JsonOutputParser(pydantic_object=Dataset)

# Generation Prompt
prompt = PromptTemplate(
    template="Generate 5 diverse and high-quality training examples for a Legal Contract Analy
    input_variables=[],
    partial_variables={"format_instructions": parser.get_format_instructions()},
)

chain = prompt | llm | parser

print("🚀 Generating synthetic legal dataset...")
# Note: Limit to 5 for the lab demonstration speed
result = chain.invoke({})

# Export to JSONL for Fine-Tuning
output_filename = "legal_training_data.jsonl"
with open(output_filename, "w") as f:
    for ex in result["examples"]:
        # Standard Alpaca/Gemma format: {"instruction": "...", "response": "..."}
        f.write(json.dumps(ex) + "\n")

print(f"✅ Successfully created {len(result['examples'])} training pairs in {output_filename}
print("\n--- Preview of Example 1 ---")
print(f"Q: {result['examples'][0]['instruction'][:80]}...")
print(f"A: {result['examples'][0]['response'][:80]}...")
```

# Lab 9.2: Production Streaming API

**Aim**: To build a production-ready, asynchronous API service using FastAPI that supports token-by-token streaming for real-time user experiences.

**Explanation**: This lab focuses on the engineering requirements of deployment:

1. **FastAPI Integration**: We create a RESTful endpoint that accepts user prompts and returns a `StreamingResponse`.
2. **SSE (Server-Sent Events)**: The backend uses Python generators to yield tokens as they arrive from the LLM, keeping the HTTP connection open until the response is complete.
3. **Latency Optimization**: By streaming results, we minimize the "Time To First Token" (TTFT), significantly improving the perceived speed of the application.

```python
# --- Lab 9.2: Production Streaming API ---
from fastapi import FastAPI
```

```python
from fastapi.responses import StreamingResponse
from pydantic import BaseModel
import ollama
import uvicorn
from langchain_google_genai import ChatGoogleGenerativeAI

app = FastAPI()

class ChatRequest(BaseModel):
    prompt: str
    stream: bool = True
    provider: str = "cloud" # "cloud" or "local"

async def generate_gemini_stream(prompt):
    # Using the standardized Gemini 1.5 Flash
    llm = ChatGoogleGenerativeAI(model=MODEL)
    async for chunk in llm.astream(prompt):
        yield f"data: {chunk.content}\n\n"
    yield "data: [DONE]\n\n"

async def generate_ollama_stream(prompt):
    # Using the standardized local model Gemma 2
    stream = ollama.generate(model=LOCAL_MODEL, prompt=prompt, stream=True)
    for chunk in stream:
        if 'response' in chunk:
            yield f"data: {chunk['response']}\n\n"
    yield "data: [DONE]\n\n"

@app.post("/chat/stream")
async def chat_stream(request: ChatRequest):
    """
    Production-ready endpoint for real-time LLM interaction
    """
    if request.provider == "cloud":
        return StreamingResponse(generate_gemini_stream(request.prompt), media_type="text/ever
    else:
        return StreamingResponse(generate_ollama_stream(request.prompt), media_type="text/ever

print("🕯️ Streaming API Service Defined.")
print(f"Cloud Model: {MODEL}")
print(f"Local Model: {LOCAL_MODEL}")
print("\nInstructions to run this service:")
print("1. Save this code to a file named 'main.py'")
print("2. Run 'uvicorn main:app --reload' in your terminal")
print("3. Test with: curl -X POST http://localhost:8000/chat/stream -d '{\"prompt\": \"Hello\'
```

# Lab 9.3: Capstone Project The Autonomous Multi-Modal Researcher

**Aim**: To architect and deploy a production-ready, agentic system that integrates RAG, LangGraph-based reasoning, and multi-modal fallback capabilities.

**Explanation**: This capstone is the culmination of the curriculum, requiring the integration of:

1. **Dynamic Ingestion**: Automated monitoring and parsing of document folders.
2. **Agentic Reasoning**: A LangGraph state machine that decides between retrieval and web research.
3. **Production APIs**: Deployment via FastAPI and Gradio with real-time thought visualization.
4. **Automated Evaluation**: A separate model acting as a judge to verify the accuracy and completeness of generated reports.

---

# Instructor's Evaluation & Lab Summary

## Assessment Criteria

1. **Technical Implementation**: Adherence to the lab objectives and code functionality.
2. **Logic & Reasoning**: Clarity in the explanation of the underlying AI principles.
3. **Best Practices**: Use of secure environment variables and structured prompts.

**Lab Completion Status: Verified Focus Area**: Language Modelling & Deep Learning Systems.