COMS 6998-015
PRAC DEEP LEARNING SYS
Instructor: Parijat Dube and Chen Wang
Due: Nov. 21, 2024

Homework 4

Problem 1 - *Approximate Nearest Neighbors* 25 points

Given a randomly-generated dataset of vectors in a high-dimensional space, implement and analyze an Approximate Nearest Neighbors (ANN) solution using the Hierarchical Navigable Small World (HNSW) approach.

Tasks:

a) (10 points) Implement a function `construct_HNSW(vectors, m_neighbors)` that builds a hierarchical graph structure where:
   - `vectors` is a numpy array of shape (n_vectors, dimension)
   - `m_neighbors` is the number of nearest neighbors to connect in each layer
   - Return a list of networkx graphs representing each layer

b) (8 points) Implement a function `search_HNSW(graph_layers, query)` that performs approximate nearest neighbor search. Your function should:
   - Accept the graph layers from `construct_HNSW` and a query vector
   - Return the nearest neighbor found and the search path taken
   - Use the layer-wise search strategy discussed in class

c) (7 points) Evaluate your implementation by:
   - Comparing results against brute force search for a dataset of 100 vectors in 2D space
   - Measuring and reporting search time for both methods
   - Visualizing one example search path through the layers
   - Calculating and reporting the accuracy of your approximate solution

Note: Use the following test parameters:

- Number of vectors: 100
- Dimension: 2
- M nearest neighbors: 2
- Test with query vector [0.5, 0.5]

Required Libraries: numpy, networkx, matplotlib

Submission: Submit your code as a Python file or Jupyter notebook with clear documentation and visualizations of the graph structure and search process.

Bonus: (+3 points) Implement and compare the performance of your solution with different values of m_neighbors (2, 4, and 8). (+2 points) Test your algorithm on a real dataset embedding (like Wikipedia) and report your results.

Problem 2: *Multilingual Retrieval Augmented Generation* 25 points

Implement a multilingual search and retrieval augmented generation system using the OPUS Books dataset, which contains parallel text in English and Italian. You will create a system that can search across languages and generate content based on the retrieved passages.

Homework 4

Tasks:

a) (8 points) Set up the vector search system:

- Use sentence-transformers' multilingual model 'paraphrase-multilingual-MiniLM-L12-v2'
- Create vector embeddings for the OPUS Books text passages
- Build a FAISS index for efficient similarity search
- Save and load the index for reuse
- Basic code structure:

```
from sentence_transformers import SentenceTransformer
import faiss
import numpy as np
from datasets import load_dataset

# Load dataset
dataset = load_dataset("opus_books", "en-it", split="train[:2000]")

# Initialize model
model = SentenceTransformer('paraphrase-multilingual-MiniLM-L12-v2')

# Create embeddings
texts_en = [item['translation']['en'] for item in dataset]
texts_it = [item['translation']['it'] for item in dataset]
embeddings = model.encode(texts_en + texts_it)

# Build FAISS index
index = faiss.IndexFlatL2(embeddings.shape[1])
index.add(embeddings)
```

b) (8 points) Implement multilingual search:

- Create a search function that accepts queries in either English or Italian
- Add metadata filtering capability to search in specific languages
- Return top-k most relevant passages with scores
- Implement efficient batch processing for multiple queries
- Example usage:

```
def search(query, lang=None, k=5):
    query_vector = model.encode([query])
    D, I = index.search(query_vector, k)

    results = []
    for d, i in zip(D[0], I[0]):
        # Determine if index i is from English or Italian portion
        is_english = i < len(texts_en)
        text = texts_en[i] if is_english else texts_it[i - len(texts_en)]
        lang_code = 'en' if is_english else 'it'
```

```
            if lang is None or lang_code == lang:
                results.append({
                    'text': text,
                    'lang': lang_code,
                    'score': float(d)
                })
        return results
```

c) (9 points) Add RAG capabilities:

- Use the Hugging Face Transformers library with mBART-large-50
- Implement both single-document and multi-document content generation
- Create a system to generate book recommendations and comparative analyses
- Compare different prompt strategies
- Example structure:

```
from transformers import AutoModelForSeq2SeqGeneration, AutoTokenizer

model = AutoModelForSeq2SeqGeneration.from_pretrained(
    "facebook/mbart-large-50-many-to-many-mmt")
tokenizer = AutoTokenizer.from_pretrained(
    "facebook/mbart-large-50-many-to-many-mmt")

def generate_content(context, prompt):
    input_text = f"{prompt}\n{context}"
    inputs = tokenizer(input_text, return_tensors="pt", max_length=512)
    outputs = model.generate(**inputs)
    return tokenizer.decode(outputs[0], skip_special_tokens=True)
```

Test Requirements:

- Test search with queries in:
  - English: “stories about adventure and discovery”
  - Italian: “storie di avventura e scoperta”
- Dataset: Use at least 1000 parallel texts from OPUS Books
- Compare search results across languages
- Generate two types of content:
  - Book recommendations based on single passages
  - Comparative analysis of multiple passages

Required Libraries:

- datasets
- sentence-transformers
- faiss-cpu
- transformers
- torch
- numpy
- tqdm

Submission:

Submit your code as a Python file or Jupyter notebook with:

- Implementation of all required components
- Example outputs showing bilingual capabilities
- Performance analysis (search time, memory usage)
- Discussion of results quality across English and Italian

Bonus:

(+5 points) Implement semantic caching to improve performance for repeated similar queries.

Notes:

- The OPUS Books dataset provides parallel texts, making it ideal for testing cross-lingual search and generation
- Focus on quality of translations and semantic similarity across languages
- Consider how to handle different writing styles and literary contexts

Problem 3: Building an Intelligent Assistant with LangChain (25 points)

Your task is to build an intelligent assistant using LangChain that can handle multiple types of queries by implementing custom tools and a routing system. Submit a Jupyter notebook demonstrating your implementation with proper documentation, example outputs, and test cases for each component.

Using the template code below as a starting point:

```python
from langchain.agents import tool
from langchain.chat_models import ChatOpenAI
from langchain.agents.output_parsers import OpenAIFunctionsAgentOutputParser
from pydantic import BaseModel, Field

# Your implementation here
```

Complete the following tasks:

1. (8 points) Implement custom tools using the @tool decorator:
   - (4 points) Create a NewsSearchTool that:
     - Accepts a query string and returns relevant news headlines
     - Uses any free news API (document your choice)
     - Returns at least 3 headlines with publication dates
     - Includes proper error handling for API failures and invalid inputs
     - Contains clear documentation and markdown explanations
   - (4 points) Create a StockPriceTool that:
     - Retrieves current stock prices using yfinance or similar library
     - Returns current price, daily high, and daily low
     - Implements input validation for stock symbols
     - Handles invalid stock symbols and API failures
     - Includes example queries and responses
2. (7 points) Define tool schemas and format for OpenAI functions:
   - (3 points) Create Pydantic schemas for both tools:
     - Define input fields with type hints
     - Add field descriptions using the Field class
     - Implement input validation rules
     - Demonstrate schema usage with valid/invalid inputs
   - (4 points) Format tools for OpenAI function integration:
     - Convert tools using format_tool_to_openai_function
     - Create unified function list for the assistant
     - Show formatted function descriptions
     - Demonstrate model's tool selection process
3. (10 points) Implement the routing system:
   - (4 points) Construct the routing chain:
     - Create ChatPromptTemplate with system message
     - Configure ChatOpenAI model with functions
     - Add OpenAIFunctionsAgentOutputParser
     - Document chain construction process
   - (3 points) Create routing logic:
     - Handle both tool calls and direct responses
     - Implement error handling for failed executions

  - Add debugging logs
  - Show routing for different query types
- (3 points) Test and document the system:
  - Test 5+ different query types
  - Include edge cases
  - Demonstrate error handling
  - Document limitations and potential improvements
  - Provide example conversations

Your submission should be a Jupyter notebook that implements all components above with clear markdown documentation, example outputs, and comprehensive testing for each feature. Include proper error handling throughout your implementation and demonstrate the system's behavior with various types of queries.

Problem 4 - *Quantization 25 points*

Introduction

Machine learning models are typically trained using 32-bit floating-point data. However, floating-point arithmetic is computationally expensive in terms of area, performance, and energy when implemented in hardware. While high precision may be necessary during training to capture small gradient steps, such precision is often unnecessary during inference. This assignment explores post-training quantization techniques to optimize neural network inference.

Objective

You will implement post-training quantization on a pre-trained convolutional neural network (CNN) for the CIFAR-10 dataset, converting both weights and activations from floating-point to fixed-point representations.

Setup Instructions

1. Access the provided Jupyter notebook template
2. The notebook contains a pre-trained CNN with:
   - Two convolutional layers (conv1, conv2)
   - Three fully connected layers (fc1, fc2, fc3)
   - ReLU activation functions
   - MaxPooling layers

Tasks and Grading Breakdown

Part A: Weight Analysis and Quantization (10 points)

1. **Weight Distribution Analysis (4 points)**
   - Plot histograms of weights for each layer
   - Calculate statistical measures (range, mean, standard deviation)

- Analyze distribution patterns across layers

2. **Weight Quantization Implementation (6 points)**
    - Complete the `quantized_weights()` function:

```
def quantized_weights(weights: torch.Tensor) -> Tuple[torch.Tensor, float]:
    '''
    Quantize weights to 8-bit integers (-128 to 127)
    Returns: (quantized_weights, scale_factor)
    '''
```

    - Implement scaling strategy for optimal accuracy
    - Validate quantized values are within int8 range

Part B: Activation Quantization (10 points)

1. **Activation Analysis (4 points)**
    - Generate activation distribution plots
    - Calculate full range and 3-sigma range
    - Document distribution patterns across network depth
2. **Activation Quantization Implementation (6 points)**
    - Complete the `NetQuantized` class:

```
class NetQuantized(nn.Module):
    def quantize_initial_input(pixels: np.ndarray) -> float:
        '''Scale factor for initial input'''
        pass

    def quantize_activations(activations: np.ndarray,
                             n_w: float,
                             n_initial_input: float,
                             ns: List[Tuple[float, float]]) -> float:
        '''Scale factor for layer outputs'''
        pass
```

    - Ensure all intermediate values are 8-bit integers
    - Maintain model accuracy

Part C: Bias Quantization (5 points)

- Implement 32-bit bias quantization in `quantized_bias()`:

```
def quantized_bias(bias: torch.Tensor,
                   n_w: float,
                   n_initial_input: float,
                   ns: List[Tuple[float, float]]) -> torch.Tensor:
    '''Quantize bias to 32-bit integers'''
```

```
        pass
```

- Account for weight and activation scaling factors
- Validate quantized values are within int32 range

Submission Requirements

1. Completed Jupyter notebook with:
   - All implemented functions
   - Generated plots
   - Model accuracy measurements
2. Written analysis including:
   - Weight and activation distribution analysis
   - Quantization strategy explanation
   - Impact on model accuracy
   - Discussion of trade-offs

Evaluation Criteria

- Correctness of implementation (40%)
- Quality of analysis (30%)
- Code documentation (15%)
- Visualization clarity (15%)

Note: Focus on maintaining model accuracy while implementing efficient quantization strategies. Consider hardware constraints and numerical stability in your implementations.