COMS 6998-013
LLM based GenAI
Instructor: Parijat Dube and Chen Wang

Homework 3

Due: Nov. 01, 2025

General Instructions

This homework must be turned in on Gradescope by 11:59 pm on the due date. It must be your own work and your own work only—you must not copy anyone's work, or allow anyone to copy yours. This extends to writing code. You may consult with others, but when you write up, you must do so alone. Your homework submission must be written and submitted using Jupyter Notebook (.ipynb). **No handwritten solutions will be accepted**. You should submit:

1. One Jupyter Notebook(.ipynb) of your solutions for each of the problems in this assignment. (For example, you will need to submit 4 notebooks for this assignment.)
2. For questions with provided notebook, please **copy a version of yours** and make edition with your notebook

Please make sure your answers are clearly structured in the Jupyter Notebooks:

1. Label each question part clearly. Do not include written answers as code comments. The code used to obtain the answer for each question part should accompany the written answer.
2. All plots should include informative axis labels and legends. All codes should be accompanied by informative comments. All output of the code should be retained.
3. Math formulas can be typesetted in Markdown in the same way as LaTeX. A Markdown Guide is provided for reference.

For more homework-related policies, please refer to the syllabus.

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. provided notebook

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:

Homework 3

- 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 - *Semantic Search for RAG Systems* 30 points

Given a collection of text documents, implement and analyze a semantic search engine that serves as the retrieval component for a Retrieval-Augmented Generation (RAG) system. provided notebook

Tasks:

a) (12 points) Document Processing and Vector Store Setup:

- Load text documents using appropriate document loaders (e.g., PyPDFLoader for PDFs)
- Split documents into chunks using RecursiveCharacterTextSplitter with `chunk_size` and `chunk_overlap` parameters
- Generate embeddings for each chunk using a pre-trained embedding model
- Store the embeddings in a vector database (e.g., Chroma) with proper metadata
- Demonstrate the setup with sample documents and verify chunk creation

b) (10 points) Implement Semantic Search with Multiple Methods:

- Implement similarity search using `vector_store.similarity_search()` method
- Implement Maximum Marginal Relevance (MMR) search using `vector_store.max_marginal_relevance_search()` method
- MMR Definition: A retrieval strategy that balances relevance with diversity to reduce redundant results. MMR iteratively selects documents that are both relevant to the query AND different from already-selected documents. Use the formula: $MMR = \lambda \times Sim(doc, query) - (1 - \lambda) \times \max(Sim(doc, selected))$ where $\lambda = 0.5$
- Demonstrate both search methods with the same queries and compare results

Homework 3

- Include search with similarity scores using `similarity_search_with_score()`

c) (8 points) Evaluate your implementation by:

- Testing on a provided dataset of 20 short articles about different topics
- Comparing retrieval quality between different chunk sizes (500, 1000, 1500 characters)
- Measuring and reporting search latency for different values of k (1, 5, 10)
- Analyzing the effect of different search types on result diversity (compare similarity vs MMR results for the same query)
- Demonstrating scenarios where MMR provides better coverage than similarity search
- Creating visualizations of embedding similarities using dimensionality reduction

Algorithm Notes:

- Follow the LangChain workflow: Document Loading → Text Splitting → Embedding Generation → Vector Store Creation → Similarity Search
- Use `RecursiveCharacterTextSplitter` with `add_start_index=True` to preserve chunk location metadata
- For MMR, use the built-in `max_marginal_relevance_search()` method or implement manually using the provided formula
- MMR reduces redundancy by penalizing documents similar to already-selected ones
- This technique is widely used in RAG systems and is implemented in LangChain's VectorStoreRetriever with `search_type="mmr"`
- MMR helps avoid the "cluster problem" where all retrieved documents cover the same aspect of a query
- Use cosine similarity for all similarity calculations
- Handle edge cases where k exceeds available documents

  Background Reading:

  - Reference: LangChain VectorStore documentation on MMR search
  - Original paper: "The Use of MMR, Diversity-Based Reranking for Reordering Documents" (Carbonell & Goldstein, 1998)

  Test Parameters:

  - Dataset: Collection of 8-10 diverse text documents (see Dataset Source above)
  - Chunk overlap: 200 characters
  - Test queries: ["artificial intelligence applications", "climate change effects", "Olympic games history", "economic policy impacts"]
  - Embedding model: sentence-transformers/all-MiniLM-L6-v2 (or OpenAI embeddings if available)
  - Ensure queries span different topics to demonstrate search diversity

  Required Libraries: numpy, pandas, sentence-transformers, scikit-learn, matplotlib, seaborn, langchain-community, langchain-text-splitters, langchain-chroma (or other vector store), pypdf

  Dataset Source:

- Primary: Create a collection of 8-10 text documents covering different topics (technology, science, sports, politics, history)
- Each document should be 500-2000 words to allow for meaningful chunking analysis
- Suggested sources: Wikipedia articles, news articles, or academic abstracts on diverse topics
- Alternative: Use the Nike 10-K PDF from LangChain Example Data Repository plus additional documents
- Ensure topic diversity to demonstrate the effectiveness of MMR vs similarity search

Deliverables:

- Python implementation with clear documentation
- Evaluation report with performance metrics and visualizations
- Analysis of optimal chunk size and search parameters
- Discussion of trade-offs between search accuracy and efficiency

Bonus: (+4 points) Implement a hybrid search combining semantic search with keyword-based BM25 scoring. (+3 points) Create a simple web interface demonstrating your search engine. (+3 points) Implement and compare different embedding models (e.g., OpenAI, Cohere, local models).

Problem 3 - Pretraining 25 points

Read the paper "Training Compute-Optimal Large Language Models" Hoffmann et al. [2022] carefully and answer the following questions. For each question, provide specific evidence and citations from the paper to support your answer. provided notebook

1. (5 points) The paper presents three different approaches to determine the optimal trade-off between model size and number of training tokens. For Approach 2 (IsoFLOP profiles), what were the exponents a and b found for the relationships $N_{opt} \propto C^a$ and $D_{opt} \propto C^b$? How do these values differ from Kaplan et al.'s findings, and what is the practical implication of this difference for training large language models?
2. (5 points) For a given compute budget of 576×10^{23} FLOPs (same as Gopher), what is the optimal model size and number of training tokens according to the paper's analysis? Compare this to Gopher's actual configuration.
3. (5 points) Did Chinchilla's improvements in performance come at a higher computational cost compared to Gopher? Explain your answer using evidence from the paper about compute budget and model efficiency.
4. (5 points) On the MMLU benchmark, what was Chinchilla's average accuracy and how did it compare to both Gopher and human expert performance? Cite specific numbers from the paper.
5. (5 points) According to the paper's analysis, what are the implications for training a 1 trillion parameter model? Would this be compute-optimal with current practices? Explain using evidence from Table 3 of the paper.

Note: Your answers should be specific and include relevant numerical evidence where appropriate. For each answer, clearly indicate which sections or tables of the paper you are referencing to support your arguments.

Homework 3

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.

References

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Amna Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022. URL https://arxiv.org/abs/2203.15556.