Final Project Assignment

Issued: Week 9 (November 6th)
Due: Week 13 (December 7th)
Group Size: 1-3 students

1. Introduction

This final capstone project is the culmination of your work this semester. It is designed to synthesize the skills you have developed across all three major parts of our course:

- **Part 1: Python Fundamentals:** You will use your knowledge of data structures, functions, OOP, and error handling to build a robust, reproducible codebase.
- **Part 2: Python for Machine Learning:** You will apply your understanding of data pipelines (NumPy, Pandas), visualization (Matplotlib), model evaluation, and validation.
- **Part 3: Python for Deep Learning and LLMs:** You will directly engage with the concepts from Weeks 9-12, working with PyTorch, Transformers, fine-tuning, and AI agentic patterns.

The project is challenging but designed to be achievable. It will provide you with a significant, industry-relevant piece for your portfolio. Your goal is to choose **one** of the three project options below and deliver a complete, well-documented, and functional solution.

2. Project Options

Please review the three options carefully. Each project includes **Core Objectives** (which form the baseline for the project) and **Advanced Directions** (opportunities to explore for a higher grade).

Project A: Fine-Tune a Small Language Model for a Practical Task

Theme: Parameter-efficient fine-tuning (PEFT) on a compact model for a real-world task (classification or generation).

Why this works: This project provides concrete exposure to PyTorch, the Transformers library, data pipelines, training loops, evaluation metrics, and overfitting control. It delivers high impact with modest compute requirements.

🌟 Real-World Motivation

Imagine you work at a fast-growing startup. The customer support team is overwhelmed with emails. Your goal is to build a model that helps them.

- **Classification Path:** Build a classifier that automatically tags incoming emails as Bug Report, Feature Request, Billing Issue, or General Inquiry so they can be routed to the right person.

- **Generation Path:** Build a generative model that drafts short, polite, professional "thank you" or "we're looking into it" replies to common questions.

Suggested models (choose one based on hardware):

- **CPU / Colab (free tier):** distilbert-base-uncased (text classification), distilgpt2 (short-form generation)
- **Single GPU (T4/V100):** TinyLlama or GPT2-small with LoRA/QLoRA

Candidate datasets (small, realistic, and safe):

- **Classification:** SST-2 (sentiment), AG News (topic), IMDb. Or find a small "customer support intent" dataset online.
- **Generation:** DailyDialog, product review summarization.
- You can sample a **subset** if the original dataset is too large.
- **Your Choice:** You may find and propose your own dataset (e.g., from Kaggle, a public API, or another course), as long as it is of a manageable size and suitable for the modest compute budget.

💡 Hints & Getting Started

- We recommend using the Hugging Face datasets library for easy data loading and tokenization.
- Start by getting a baseline model (e.g., distilbert-base-uncased) running with the Trainer API for a simple classification task.
- Once the baseline works, focus on integrating PEFT (like LoRA) using the peft library. This often involves just a few lines of code to modify your model config.
- Pay close attention to your data collator, especially for generation tasks, to ensure padding and masking are handled correctly.

🎯 Core Objectives

Your goal is to demonstrate a full fine-tuning workflow. Success here means:

1. **Building a full pipeline:** This includes loading and cleaning the dataset, tokenization, train/val splits, PEFT (LoRA) fine-tuning, and implementing mechanisms like early stopping and basic checkpointing.
2. **Reporting standard metrics:**
   - **For Classification:** You should report Accuracy/F1 and consider confidence intervals (via bootstrap or multiple seeds) to show the stability of your results.
   - **For Generation:** You should report ROUGE-L scores. To capture qualitative differences, you must supplement this by implementing a lightweight "LLM-as-Judge" evaluator. You will use a separate LLM (e.g., GPT-3.5 or another API) to score the quality (e.g., politeness, relevance) of your model's generated outputs (minimum 20 samples) based on a clear rubric you design.
3. **Analyzing overfitting control:** You should experiment with and compare different hyperparameters, such as a learning-rate sweep or weight decay, and visualize the impact with Matplotlib loss/metric curves.

🚀 Advanced Directions (Explore at least two. You may also propose your own.)

- **Data-centric variant:** How does the data itself impact the model? Try implementing active sampling or simple data augmentation and compare your results (ideally with statistical tests).
- **Generalization variant:** How well does your model learn? Conduct a zero-shot vs. few-shot vs. fine-tuned comparison. You could also analyze the scaling of validation performance as you increase the number of training examples.
- **Efficiency variant:** What are the trade-offs? Compare QLoRA (4/8-bit) vs. full-precision fine-tuning (if your GPU permits) and report on differences in training time and memory usage.
- **Safety/robustness check:** How robust is your model? Implement a simple toxicity or prompt-injection probe and document potential mitigations (e.g., filtering, refusal templates).
- **Your Own Variant:** Propose an alternative advanced direction of equivalent or greater complexity. **This must be discussed and approved by the lecturer.**

📦 Deliverables (Project A)

1. Reproducible code (ideally a one-click run with a config file). Detailed README.md file.
2. A 6–8 page report: problem, method, ablations, results, failure modes, limitations, etc.
3. A short (≤ 3 min) pre-recorded screencast demoing your project.
4. A Final Presentation at Week 13.

📊 Grading Rubric (Project A)

- Work: 60 % (including Significance, Originality, Results & Analysis, Difficulty & Workload, Code Quality, Report Clarity, etc.)
- Presentation: 40 % (Teacher&TA&Peer review, including Overall Impression, Clarity of Speech, Organization, etc.)

Project B: Tool-Using LLM Agent with the ReAct Pattern

Theme: An LLM-powered agent that can reason step-by-step, consult tools, and return grounded, auditable answers.

Why this works: Agents are a highly relevant industry topic. This project allows you to practice function-calling, error handling, control flow, and the evaluation of reasoning reliability.

🌟 Real-World Motivation

Imagine you are building a "Personal Travel Assistant." A user wants to ask: "What's a good 3-day weekend trip from my city, and how much would a hotel cost?" An LLM alone would hallucinate. Your agent can solve this by:

1. Using a search tool to find "weekend trips near [user_city]." (e.g., Observation: Found cities A, B, C.)

2. Using the search tool again for "hotels in city A." (e.g., Observation: Found hotels X, Y, Z.)
3. Using a calculator tool to "multiply [hotel_price] by 3."
4. Finally, synthesizing this information into a grounded answer.

Suggested toolset (select ≥2):

- **Search/Docs:** A local corpus or Wikipedia API subset; simple retrieval via a local vector store (e.g., FAISS).
- **Computation:** A safe math/eval tool (Python evaluator sandboxed to basic arithmetic).
- **Planner:** A deterministic planner that breaks tasks into substeps (your own or a lightweight plan skeleton).

💡 Hints & Getting Started

- Start simple! Create one tool, like a calculator (def add(a, b): return a + b), and write a ReAct prompt by hand. Your first goal should be to parse the LLM's output (e.g., Action: add(2, 3)), call your Python function, and feed the Observation: 5 back into the loop.
- Focus heavily on prompt engineering. Your prompt must clearly explain the Thought -> Action -> Observation cycle and provide a few examples (few-shot learning).
- Error handling is critical. What happens if the LLM calls a tool that doesn't exist or provides malformed JSON? Your agent needs to handle this gracefully and report an error observation.

🎯 Core Objectives

Your goal is to build a reliable agent that can successfully use tools to answer questions. Success here means:

1. **Implementing a minimal ReAct loop:** Your agent should follow the *thought* → *action (tool call)* → *observation* → *final answer* cycle. This requires defining structured tool schemas (JSON in/out) and parsing the LLM's responses.
2. **Adding essential guardrails:** A reliable agent needs safety features. You should implement basic but critical guardrails, such as a **maximum number of steps** (to prevent infinite loops) and **timeouts** for tool calls.
3. **Building a small evaluation set:** Create an evaluation set of **~20-30 queries** with "gold" final answers to test your agent's ability to use its tools correctly.
4. **Enabling basic logging for debugging:** For an agent to be useful, it must be debuggable. You should log each Thought, Action, and Observation to a file or the console to create a "trace" of the agent's reasoning steps.

🚀 Advanced Directions (Explore at least two. You may also propose your own.)

- **Self-consistency or majority-vote decoding:** How can you improve reliability? Try running K traces (multiple agent runs) and pick the best answer via a verifier model or a simple heuristic.
- **Hallucination control:** How do you ensure factual accuracy? Integrate retrieval confidence scores from your search tool and add a citation requirement. Your agent could auto-fail if no high-confidence evidence is found.

- **Judge model:** Can you automate evaluation? Use a lightweight evaluator model (e.g., a fine-tuned classifier) to score the factuality and tool-use efficiency of your agent's traces.
- **Safety filter:** How do you defend against attacks? Implement a prompt-injection detector using heuristic patterns and demonstrate both an attack and your mitigation.
- **Your Own Variant:** Propose an alternative advanced direction of equivalent or greater complexity. **This must be discussed and approved by the lecturer.**

📦 Deliverables (Project B)

1. Agent framework code (ideally a one-click run with a config file) + tool specifications (well-documented). Detailed README.md file.
2. An evaluation set (20-30+ queries with gold answers) and a scoring script.
3. A 6–8 page report: design choices, failure analysis, ablations (e.g., with/without retrieval, with/without verifier), etc.
4. A short (≤ 3 min) pre-recorded screencast demoing your project.
5. A Final Presentation at Week 13.

📊 Grading Rubric (Project B)

- Work: 60 % (including Significance, Originality, Results & Analysis, Difficulty & Workload, Code Quality, Report Clarity, etc.)
- Presentation: 40 % (Teacher&TA&Peer review, including Overall Impression, Clarity of Speech, Organization, etc.)

Project C: Retrieval-Augmented Generation (RAG) with End-to-End Evaluation

Theme: Build a domain-specific assistant that answers questions from a provided corpus using RAG, emphasizing indexing, chunking, and evaluation.

Why this works: RAG is the most ubiquitous enterprise LLM pattern today. This project provides hands-on experience with data engineering, vector search, prompt design, batching, caching, and objective evaluation.

🌟 Real-World Motivation

Imagine you want to build a "Q&A Bot" for a specific domain.

- **For Tech:** Build a bot that can answer highly specific questions about a library like pandas or scikit-learn by reading its official documentation. No more hunting through 10 different web pages for an answer.
- **For Fun:** Build a "Game Lore Expert" for a video game like *Elden Ring* or *Cyberpunk 2077*. You feed it the game's wiki, and it can answer deep questions about characters, locations, and history, complete with citations.

Provided assets (pick one domain corpus):

- **Tech docs:** (e.g., Python, NumPy, PyTorch, or pandas excerpts)
- **Policy/handbook:** (e.g., academic policies, housing rules)
- **Mini research corpus:** (10–20 survey/tutorial papers; abstracts + sections)
- **Your choice:** A corpus of your own (e.g., a game wiki, a set of regulations) of similar size.

💡 Hints & Getting Started

- Your first priority should be the data pipeline. Start by writing a script to load, clean, and chunk your documents. A simple fixed-size chunker is a great starting point.
- Use a library like sentence-transformers to generate embeddings and faiss-cpu for a simple, fast vector index.
- Build your evaluation set *early*. Expanding on the provided starter set will make it much easier to objectively measure your progress as you experiment with different chunkers, retrievers, and prompts.
- Your prompt is key. A good RAG prompt clearly instructs the LLM to *only* use the provided context to answer the question and to say "I don't know" if the answer isn't present.

🎯 Core Objectives

Your goal is to build and evaluate a full RAG system, focusing on the quality of both retrieval and generation. Success here means:

1. **Building the full RAG pipeline:** This involves data cleaning, implementing a chunking strategy (e.g., fixed-size vs. semantic split), generating embeddings, creating a vector index, building a retriever, and passing the retrieved context to a generator (LLM).
2. **Creating a strong evaluation:** You must implement a "closed-book" (no RAG) vs. RAG evaluation. Your job is to create from scratch a strong evaluation set of at least 50 total pairs (with ground-truth answers and source citations) to test your system.
3. **Running comparison experiments:** You should compare at least two retrievers (e.g., a sparse retriever like BM25 vs. a dense retriever; or two different dense models) and at least two prompts (e.g., a vanilla prompt vs. a more specialized instruction-tuned prompt).
4. **Reporting comprehensive metrics:** You need to report retrieval metrics (Recall@k, MRR) and answer quality metrics (Exact Match / F1 or ROUGE). This should be supplemented by implementing an "LLM-as-Judge" evaluator (for at least 30 samples) to assess the qualitative aspects of your answers (e.g., faithfulness, relevance).

🚀 Advanced Directions (Explore at least two. You may also propose your own.)

- **Query rewriting:** What if the user's query is bad? Implement a technique like HyDE (Hypothetical Document Embeddings) or a "condense question" step (using an LLM to rewrite the query) and evaluate the gains.
- **Re-ranking:** Your retriever finds 10 documents, but are the top 3 the best? Add a lightweight cross-encoder re-ranker to re-order the initial retrieved documents and perform an ablation study.
- **Latency & memory profiling:** How efficient is your system? Profile factors like batch size, max token length, and caching. Can you produce a throughput vs. quality chart?
- **Safety & privacy:** What if your documents contain sensitive info? Implement a step to redact

PII-like patterns in the corpus and measure the impact this has on answerability.

- **Your Own Variant:** Propose an alternative advanced direction of equivalent or greater complexity. **This must be discussed and approved by the lecturer.**

📦 Deliverables (Project C)

1. Reproducible repository (ideally a one-click run with a config file). Detailed README.md file.
2. Your evaluation set (50+ QA pairs) and your scorer script.
3. A 6–8 page report: your design, experiments, cost/latency-quality trade-off, etc.
4. A short (≤ 3 min) pre-recorded screencast demoing your project.
5. A Final Presentation at Week 13.

📊 Grading Rubric (Project C)

- Work: 60 % (including Significance, Originality, Results & Analysis, Difficulty & Workload, Code Quality, Report Clarity, etc.)
- Presentation: 40 % (Teacher&TA&Peer review, including Overall Impression, Clarity of Speech, Organization, etc.)

3. Shared Logistics and Requirements

Compute & Environment

- **Target:** All projects are scoped for CPU or a single T4/V100/A10 GPU on Google Colab/Kaggle (8–12GB VRAM).
- **Reproducibility:** You must enforce deterministic seeds and provide a requirements.txt file.
- **Constraint:** Training runs (for Project A) should be limited to **≤ 2 hours** of wall-clock time on a T4 GPU.

Overall Grading Philosophy

Your grade will be based on a holistic assessment of your work, weighted according to the project-specific rubrics. Beyond the technical components, we are looking for:

- **Technical depth & correctness:** Does your project work, and is it built on a solid understanding of the concepts?
- **Experimental design & reproducibility:** Did you design fair experiments to test your ideas? Can someone else run your code and get the same results?
- **Results quality & analysis:** Did you go beyond just reporting numbers and provide insightful analysis of *why* your system behaves the way it does (including failure analysis)?
- **Report quality & communication:** Is your report, demo, and code clear, professional, and easy to understand? Did you thoughtfully analyze the **limitations** and ethical implications?
- **Presentation/demo:** Did you effectively communicate your project's goals, methods, and results in the allotted time?

Academic Integrity

You are permitted to use open-source libraries and pretrained checkpoints. However, you **must** require a "Bill of Materials" in your report, listing all assets, libraries, and checkpoints used, with proper citations. Team contribution statements are required. Be prepared for random oral checks on your submission.