COMS 6998-013
LLM based GenAI
Instructor: Parijat Dube and Chen Wang
Due: Oct. 15, 2025

Homework 2

Problem 1 - *Distributed Pipeline Parallelism in Pytorch* 20 points

This problem explores distributed pipeline parallelism in PyTorch, a critical technique for training large-scale deep learning models that exceed the memory capacity of a single GPU. You will first study the Pytorch distributed pipelining API torch.distributed.pipelining to demonstrate your understanding of different pipeline scheduling strategies and then implement and evaluate their performance empirically using the PyTorch distributed pipelining API.

Part 1: Conceptual Understanding of Pipeline Schedules (4 points)

Answer the following open-ended questions to demonstrate your understanding of different pipeline parallelism schedules. Each response should be clear, technical, and demonstrate deep understanding of the trade-offs involved.

1. **(2 points)** Explain the concept of *pipeline bubbles* in the context of pipeline parallelism. How do different schedules (GPipe vs. 1F1B vs. Interleaved1F1B) address the bubble problem? Provide a quantitative or qualitative analysis of bubble overhead for each schedule.
2. **(2 points)** Describe the `Interleaved1F1B` schedule. What is the key innovation that distinguishes it from regular 1F1B? What is meant by "multiple stages per rank" and how does this improve pipeline efficiency?

Part 2: Implementation and Performance Evaluation (16 points)

In this part, you will implement and evaluate different pipeline parallelism schedules using PyTorch's distributed pipelining API. Follow the tutorial at `https://docs.pytorch.org/tutorials/intermediate/pipelining_tutorial.html` as your starting point.

Setup and Implementation (3 points)

1. Set up your environment for distributed pipeline parallelism:
   - Install the required PyTorch version with distributed support
   - Verify your multi-GPU setup (document the number of GPUs available)
   - Follow the tutorial to understand the basic pipeline parallelism workflow
   - Document your setup process, including hardware specifications, PyTorch version, and any configuration steps
2. Implement a transformer model with pipeline parallelism support using the following schedules:
   - `GPipe`
   - `1F1B`
   - `Interleaved1F1B`

   Ensure your implementation allows for easy modification of model hyperparameters (number of layers, number of attention heads, hidden dimensions, etc.). Document your code structure and any modifications made to the tutorial code.

Experimental Evaluation (13 points)

1. **(8 points)** Design and conduct experiments to evaluate the performance of different pipeline schedules by varying the following hyperparameters:
   - **Number of layers:** Test 3 different configurations: 4, 8, 12 layers
   - **Number of attention heads:** Test 3 different configurations: 4, 8, 12 heads
   - **Number of processes:** Test 2 different configurations: 2 and 4
   - Keep other parameters (batch size, sequence length, hidden dimension) constant

   For each configuration and schedule combination, measure and record:
   - Training throughput (tokens/second)
   - End-to-end training time for a fixed number of iterations

   Present your results in clear tables and/or plots showing the relationship between model configuration and performance for each schedule.

2. **(3 points)** Compute the speedup and scaling efficiency.
   - **Speedup:** Calculate training throughput speedup relative to GPipe baseline.
   - **Scaling Efficiency:** Compute scaling efficiency as:

   $$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of processes}} \times 100\%$$

   - Note that in your experiments you have 9 different model sizes (different combinations of number of layers and attention heads). For different model sizes plot the speedup and scaling efficiency achieved with `1F1B` and `Interleaved1F1B` relative to `GPipe`. You will produce two plots, one for speedup and one for scaling efficiency.

3. **(2 points)** Provide an analysis of your results:
   - Which schedule performs best for different model configurations? Why?
   - How does performance scale with increasing model size (layers and heads)?

Submission Requirements

Submit your work as:

- A Jupyter notebook (.ipynb) with clear markdown cells explaining each step

Your submission should include:

1. Detailed step-by-step documentation of your setup and implementation process
2. All code used for implementation and experiments (well-commented)
3. Complete experimental results with tables and visualizations
4. Calculated speedup and scaling efficiency metrics
5. Written (typed) responses to all conceptual questions in Part 1

6. Comprehensive analysis and discussion of your empirical results from Part 2
7. A brief section on reproducibility: document random seeds, hardware specs, PyTorch version, and any other relevant configuration details

Note: If you do not have access to multiple GPUs, you may use CPU-based distributed training for demonstration purposes, but clearly document this limitation and discuss how results might differ with actual GPUs.

Problem 2 - *Kubernetes & Docker Warm-Up* 20 points

Goal. Get hands-on experience with Docker, Docker Hub, `kind` (Kubernetes-in-Docker), and `kubectl` by containerizing a simple Python web server and deploying it to a local kind cluster.

Instructions. Unless stated otherwise, include the exact commands you ran, brief explanations (1–2 sentences each), terminal outputs or screenshots, and links where requested. Use a public Docker Hub repository. You may implement the web server using Python's `http.server` or a minimal Flask app. Use container port `8000` and expose via a Kubernetes `Service`.

(a) **Create a Docker Hub account and repository (2 pts).**
Sign up for a free Docker Hub account at Docker Hub (if you do not already have one) and create a public repository named `<your-username>/k8s-warmup` (or similar). Provide a screenshot of the repository page showing the empty (or initial) Tags list and the repository URL.

(b) **Write a minimal Python web server (2 pts).**
Implement a simple HTTP server that responds on `/` with a short message (e.g., greeting, hostname, and timestamp). Place the code in a small project folder (e.g., `app/`). Show the file listing and the server code in your write-up.

(c) **Create a Dockerfile and run locally (2 pts).**
Install Docker if needed (installation guide). Write a `Dockerfile` (e.g., based on `python:3.11-slim`) that installs any dependencies, copies your code, sets `ENV PORT=8000`, exposes `8000`, and starts the server as a non-root user. Build the image (e.g., `docker build -t <your-username>/k8s-warmup:dev .`) and run it locally (`docker run -p 8000:8000 ...`). Show a local browser or `curl` screenshot proving the server works.

(d) **Tag and push to Docker Hub (2 pts).**
Tag the image (e.g., `:v0.1`) and push it to your Docker Hub repo
(e.g., `docker push <your-username>/k8s-warmup:v0.1`). Include CLI output and a screenshot of the Docker Hub Tags/Activity page showing the pushed image.

(e) **Install kind and create a cluster (2 pts).**
Install `kind` from the quick-start guide. Create a cluster (e.g., `kind create cluster --name k8s101`). Show `kind get clusters` and `kubectl cluster-info` (or `kubectl get nodes`) confirming the cluster is running.

(f) **Install `kubectl` (if needed) and verify context (1 pt).**
Install `kubectl` from the official tools page. Ensure `kubectl` is configured to talk to your kind cluster (context typically `kind-k8s101`). Show `kubectl config get-contexts` highlighting the current context and `kubectl get nodes` output.

(g) **Write and apply a Deployment manifest (3 pts).**
Create `deploy.yaml` defining a `Deployment` named `web` with `replicas: 2`, container image `<your-username>/k8s-warmup:v0.1`, `env: PORT=8000`, and `containerPort: 8000`. Add a simple `readinessProbe` (e.g., HTTP GET on `/`) if your server supports it. Apply it (`kubectl apply -f deploy.yaml`) and show rollout status: `kubectl rollout status deploy/web` and `kubectl get pods -o wide`.

(h) **Create and apply a Service manifest (2 pts).**
Create `service.yaml` with a `ClusterIP Service` named `web`, mapping `port: 80` to `targetPort: 8000`. Apply it and show `kubectl get svc web` output.

(i) **Port-forward and test in a browser (2 pts).**
Use `kubectl port-forward svc/web 8080:80` and verify the app at `http://localhost:8080` in your browser. Include a screenshot of the page (or `curl` output) proving the request was served by a pod in the cluster.

(j) **Documentation & submission (2 pts).**
Submit either (i) a Markdown file or (ii) a Jupyter notebook that presents a clear, step-by-step narrative of everything above, including:

- The Docker Hub repository URL and screenshots of tag/push history.
- The Python server code and `Dockerfile`.
- The Kubernetes manifests (`deploy.yaml`, `service.yaml`).
- Commands executed and brief explanations.
- Screenshots showing local run, cluster status, and browser verification via port-forward.

Clarity, completeness, and reproducibility are graded here.

Optional cleanup (no points). Delete resources and cluster: `kubectl delete -f service.yaml -f deploy.yaml` and `kind delete cluster --name k8s101`.

Problem 3 - *Dify on Docker Swarm (Self-Hosted)* 20 points

Goal. Deploy the Dify self-hosted stack locally using **Docker Swarm** (single-node swarm on your laptop), configure model provider credentials and an `INIT_PASSWORD`, then create and export a small Dify app with a workflow of at least 2–3 steps. Submit both the exported Dify app YAML and a step-by-step Markdown tutorial of what you did.

Reference. Follow Dify's official docker-compose guide: docs.dify.ai/en/getting-started/install-self-hosted/docker-compose. You will adapt it to *Docker Swarm* using `docker stack deploy`.

Instructions. Unless stated otherwise, include the exact commands you ran, brief explanations (1–2 sentences), terminal outputs or screenshots, and links where requested. **Mask all API keys in your submission** (e.g., show only the first/last 4 characters). Use a single-node Swarm on your own machine.

(a) **Prerequisites check & repository setup (2 pts).**
Verify Docker is installed (`docker version`) and that Compose V2 is available (`docker compose version`). Create a new project folder `dify-swarm/` and place the upstream `docker-compose.yml` from the Dify docs (download or copy/paste). Show the folder listing and the top of your compose file.

(b) **Initialize a single-node Docker Swarm (2 pts).**
Initialize Swarm locally: `docker swarm init`. Show `docker info | grep -i swarm` and `docker node ls` to confirm the node is a manager. Briefly explain why we use `docker stack deploy` (Swarm mode) instead of `docker compose up`.

(c) **Environment configuration (`.env`) (3 pts).**
Create a `.env` file in your project directory to provide required variables referenced by Dify's compose, including `INIT_PASSWORD` and at least one model provider key (e.g., `OPENAI_API_KEY` or another supported provider). Generate a secure init password (e.g., `openssl rand -base64 24`) and *do not* hardcode secrets in the compose file. Show the redacted `.env` contents and explain which services use which variables.

(d) **(Optional if needed) Compose for Swarm tweaks (2 pts).**
If the upstream compose file already uses version "3.x" and named volumes/networks, you can deploy it as-is. If not, make minimal adjustments so `docker stack deploy` works (e.g., ensure `version: "3"` or higher; avoid `container_name`; confirm named volumes). Show a short diff or snippet of any changes and explain why they are required in Swarm.

(e) **Deploy the stack with Swarm (3 pts).**
Deploy as a stack named `dify`: `docker stack deploy -c docker-compose.yml dify`. Show:

- `docker stack ls`, `docker stack services dify`
- `docker service ps <service-name>` for at least one core service
- How you inspected logs for readiness (e.g., `docker service logs -f dify_api` or web)

When the web UI is reachable (typically `http://localhost:3000`), include a screenshot of the login page.

(f) **First login and provider configuration (3 pts).**
Log into Dify Studio using your `INIT_PASSWORD`. In Settings → Model Providers, configure at least one provider (e.g., OpenAI, Azure OpenAI, Anthropic, etc.) using the API key you set in `.env`. Provide screenshots of the provider page (with keys masked) and a brief note on the default model(s) you plan to use.

(g) **Create a small app with 2–3 steps (3 pts).**
Create an app in Dify Studio with a workflow consisting of at least 2–3 steps (e.g., retrieval, transformation, generation, or tool calls). The design is open-ended: you may build a PDF chatbot, a simple agent, or any workflow of your choice, as long as it demonstrates chaining multiple steps. Provide a screenshot of the workflow canvas and a test run in the Studio showing a successful response.

(h) **Export your Dify app configuration (YAML) (1 pt).**
Export the app definition as YAML (from the Studio's export function). Submit the exported YAML file as `app-export.yaml`. (If your export contains secrets, redact them before submission.)

(i) **Documentation & submission (1 pt).**
Submit a concise Markdown tutorial `README.md` that a classmate can reproduce:

- Prereqs, Swarm init, environment setup (`.env`)
- Stack deployment, service health checks, and URL
- Provider configuration steps
- App creation steps (with *brief* screenshots)
- Export instructions and where to find the YAML

Clarity and reproducibility are graded here.

Deliverables.

- Redacted `.env` (inline in the write-up or as `.env.example`).
- Screenshots/outputs requested above.
- `app-export.yaml` (exported Dify app configuration, secrets masked).
- `README.md` (step-by-step tutorial).

Security & hygiene notes. *Never* commit real API keys. Provide a `.env.example` with placeholder values for submission, and redact secrets in screenshots.

Optional cleanup (no points). Remove the stack and leave Swarm: `docker stack rm dify` then after services stop, `docker swarm leave --force`.

Problem 4: Contrastive Decoding 20 points

This problem explores the contrastive decoding proposed in Li et al. (2022). Contrastive decoding is a novel approach to text generation that improves the quality of language model outputs by contrasting the predictions of expert and amateur models. You will first demonstrate your understanding of the theoretical foundations and then implement and evaluate the method empirically.

Part 1: Conceptual Understanding (6 points)

Answer the following open-ended questions to demonstrate your understanding of contrastive decoding. Each response should be clear, concise, and demonstrate a thorough understanding of the concepts.

1. **(2 points)** Explain the core intuition behind contrastive decoding. How does it differ from standard decoding strategies like greedy decoding, beam search, or nucleus sampling? What problem is contrastive decoding trying to solve?
2. **(2 points)** Describe the mathematical formulation of the contrastive objective. Why unconstrained maximization of the contrastive objective may result in false negatives and false positives during generation.
3. **(2 points)** Why do we need an adaptive plausibility constraint in the full contrastive decoding formulation? How does the choice of α affect the generated text?

Part 2: Implementation and Empirical Evaluation (14 points)

In this part, you will implement and evaluate contrastive decoding using the Wikitext-103 dataset. You will compare three different amateur model selection strategies as described in Section 3.4 of Li et al. (2022).

COMS 6998-013
LLM based GenAI
Instructor: Parijat Dube and Chen Wang
Due: Oct. 15, 2025

Homework 2

Setup and Implementation (4 points)

1. **(2 points)** Clone the contrastive decoding repository from `https://github.com/XiangLi1999/ContrastiveDecoding` and set up the environment. Load the Wikitext-103 dataset and prepare it for text generation evaluation. Document your setup process, including any dependencies installed and preprocessing steps taken.

2. **(2 points)** Implement or adapt the provided code to perform contrastive decoding by choosing an amateur model as a smaller model from the same model family as the expert model, e.g., for GPT-2 XL as the expert and GPT-2 small as the amateur. Using your implementation perform ablation studies by:

   - **Varying temperature of amateur model**: Experiment with 3 different temperature settings 0.5,1,1.5
   - **Restricting context window of amateur model**: Experiment with 3 different settings: maximum context window, half of the maximum context window, context window of 1 (i.e., only allowing the amateur LM to condition on the last token).

   In your code you should clearly document how you changed the temperature and context window of the amateur model.

Evaluation and Analysis (10 points)

1. **(6 points)** Generate text samples using contrastive decoding for each of the configurations of the amateur model (you have 9 different experimental configurations) on Wikitext-103. For each configuration, evaluate the generated text using the following metrics (as described in Section 5.1 of Li et al. (2022)):

   - **Diversity:** Measure lexical diversity using metrics such as distinct-n scores
   - **MAUVE:** Compute the MAUVE score to measure the gap between generated and human text distributions
   - **Coherence:** Evaluate coherence using an appropriate metric (e.g., perplexity or coherence scoring model)

   Present your results in a clear table format comparing all 9 different configurations across these metrics.

2. **(2 points)** Analyze and interpret your results. Which amateur model configuration performs best according to each metric? Are there trade-offs between different metrics? Discuss your findings in the context of the original paper's results.

3. **(2 points)** Provide qualitative examples by showing at least 2-3 generated text samples from the best and the worst strategy as per the Diversity metric. Briefly comment on observable differences in generation quality.

Submission Requirements

Submit your work as:

- A Jupyter notebook (.ipynb) with clear markdown cells explaining each step

Your submission should include:

1. Detailed step-by-step documentation of your setup and implementation process

2. All code used for implementation and evaluation (well-commented)
3. Results tables and visualizations
4. Written responses to all conceptual questions in Part 1
5. Analysis and discussion of your empirical results from Part 2
6. Generated text samples for qualitative evaluation

Note: Ensure your work is reproducible by documenting random seeds, hyperparameters, and model versions used.

Problem 5: Prompt Engineering for Math Problem Solving 20 points

Objective: To explore and optimize the effectiveness of different prompting techniques in guiding a Large Language Model (LLM) to solve a specific math word problem from the GSM8K dataset.

Tasks:

1. (3 points) **Dataset Setup:** Utilize the Hugging Face `datasets` library to load the GSM8K dataset. Randomly select one question from the test set for your experiments.
2. (5 points) **Model Selection:** Choose a suitable LLM from the Hugging Face Model Hub or OpenAI's API, considering factors like performance and computational resources.
3. (8 points) **Prompt Engineering:** Implement the following prompting functions for your chosen problem:
   - `generate_solution(prompt, problem)`: A generic function that takes a prompt and a problem as input and returns the model's generated solution.
   - `one_shot_prompting_numeric(problem_to_solve)`: Implements one-shot prompting with a focus on numerical answers.
   - `two_shot_prompting_numeric(problem_to_solve)`: Implements two-shot prompting, also focused on numerical answers.
   - `two_shot_cot_prompting(problem_to_solve)`: Implements two-shot Chain-of-Thought (CoT) prompting, encouraging step-by-step reasoning. Refer to the work of Wei et al. [2022] for more details on CoT prompting.
4. (3 points) **Prompt Refinement:** Experiment with variations of your prompts to improve accuracy for the chosen problem:
   - Try different phrasings, instructions, or examples in your prompts.
   - Explore adding specific instructions for common math operations or problem-solving strategies.
   - Experiment with prompts that encourage the model to double-check its work or consider alternative approaches.
5. (3 points) **Evaluation:** Test your implemented functions and refined prompts on your chosen problem. Compare the performance of different prompting techniques and analyze their effectiveness in producing the correct answer.

6. (3 points) **Summary:** In your notebook, include a brief summary (no more than 200 words) of your prompt engineering experience. Discuss the most effective prompting strategies you discovered, challenges you encountered, and any insights gained from the experiment.

Hints:

- Consider using the `transformers` library for model loading and inference if using Hugging Face models.
- Pay attention to how different wordings or structures in your prompts affect the model's reasoning process.
- Experiment with prompts that break down complex problems into smaller, manageable steps.
- Consider prompts that encourage the model to explain its reasoning, which may lead to more accurate results.

Submission:

- Submit your Jupyter notebook containing:
  - All implemented functions and experiments
  - The randomly chosen question and its correct answer
  - Results and outputs from your various prompting techniques
  - Your 200-word summary of the prompt engineering experience
- Ensure your code is well-commented and includes instructions for running the experiments

References

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.