Design Document: Comparative Analysis of Nash Learning vs. DPO for Iterative Mathematical Reasoning

Authors: Hetvi Bagdai, Vivaan Garg

Date: January 7, 2026

Hardware Target: Single Tesla T4 (16GB VRAM)

---

1. Executive Summary

The objective of this project is to develop a stable **Iterative Self-Improvement Loop** for Large Language Models (LLMs) in the domain of **Mathematical Reasoning**.

Standard alignment methods like **DPO (Direct Preference Optimization)** often suffer from "Mode Collapse" (loss of diversity and reasoning capability) when trained on self-generated data. We propose implementing and evaluating **Nash Learning from Human Feedback (NLHF)**, specifically the **Nash-MD** algorithm, as a superior alternative. We hypothesize that Nash-MD's game-theoretic approach (optimizing against a mixture policy) will provide the stability required for continuous self-improvement.

---

2. System Architecture

The system is designed as a closed-loop feedback system consisting of three primary nodes: **The Generator**, **The Verifier**, and **The Learner**.

2.1 High-Level Data Flow

1. **Seeding:** The system starts with a base model (M_0) and a seed dataset (GSM8K).
2. **Generation Phase (Rollout):** M_i generates N solutions per problem with high temperature to encourage exploration.
3. **Verification Phase (The Judge):** A deterministic Python Execution Engine evaluates the solutions. Correct solutions are labeled **Winners (y_w)**; incorrect ones are **Losers (y_l)**.
4. **Training Phase (The Update):**
    - **Track A:** M_{i+1} is trained using **DPO Loss**.
    - **Track B:** M_{i+1} is trained using **Nash-MD Loss** (with geometric mixture).
5. **Iteration:** The improved model M_{i+1} becomes the new Generator for the next round.

---

3. Component Design Details

3.1 The Model Engine (Constraint Optimization)

Since we are limited to a single Tesla T4 (16GB VRAM), we cannot load the standard Llama-3-8B model (which requires ~16GB just for weights, leaving no room for training).

Design Decision: Use **Unsloth** with **QLoRA (Quantized Low-Rank Adaptation)**.

- **Base Model:** `unsloth/llama-3-8b-bnb-4bit` (Occupies ~5.5GB VRAM).
- **Trainable Parameters:** Only LoRA adapters (Rank $r=16$, Alpha $\alpha=16$).
- **Memory Footprint:**
  - Model Weights: 5.5 GB
  - Activations (with Gradient Checkpointing): ~4 GB
  - Optimizer State (Paged AdamW 8-bit): ~1 GB
  - **Total:** ~10.5 GB (Safe within 16GB limit).

3.2 The Verification Module (The "Judge")

Unlike text summarization (subjective), Math is objective. We replace human labeling with a programmatic verifier.

- **Input:** Raw text output from the model.
- **Logic:**
  1. **Extraction:** Regex parser finds the final answer (e.g., `\boxed{42}`).
  2. **Execution:** If the model outputs Python code, execute it in a sandbox to get the result.
  3. **Comparison:** Compare result against GSM8K ground truth.
- **Output:** Binary Label (Correct/Incorrect).

3.3 The Training Module (The Algorithm)

This is the core novelty. We implement two distinct Loss Functions.

A. DPO Baseline (Standard)

- Optimizes the likelihood of the chosen response relative to the rejected response.
- *Risk:* Prone to overfitting on "False Positives" (lucky guesses).

B. Nash-MD Implementation (Novel)

- Equation: Uses a Geometric Mixture Policy (π_{mix}) as the reference.
  $$\pi_{mix} = \pi_{model}^{(1-\beta)} \cdot \pi_{ref}^{\beta}$$
- **Mechanism:** The model plays a game against this mixture. If it deviates too far from the reference (π_{ref}), the gradient pulls it back.
- **Implementation:** We will override the `TRL` trainer to calculate probabilities against this dynamic mixture rather than a static reference.

4. Implementation Roadmap

Phase 1: Infrastructure & Baseline (Weeks 1-4)

- **Objective:** Get the pipeline running on T4 GPU.
- **Deliverables:**
    - Setup Conda environment with unsloth, trl, peft.
    - Download and format GSM8K dataset.
    - Run **DPO** on static data (GSM8K Train Set) to establish a baseline score.

Phase 2: The Nash Implementation (Weeks 5-8)

- **Objective:** Implement the Game-Theoretic Loss.
- **Deliverables:**
    - Custom NashTrainer class inheriting from HuggingFace Trainer.
    - Implement the "Geometric Mixture" probability calculation.
    - Verify that Nash training converges on static data.

Phase 3: The Self-Improvement Loop (Weeks 9-12)

- **Objective:** Connect the components into a loop.
- **Deliverables:**
    - Build generate_data.py: Uses the model to create synthetic datasets.
    - Build verify_data.py: Labels the synthetic data.
    - Run 3 Iterations of **Self-Play**:
        - Round 1: Train on Model_0 generated data.
        - Round 2: Train on Model_1 generated data.
        - Round 3: Train on Model_2 generated data.

---

5. Evaluation Metrics

How will we prove success?

1. **Pass@1 Accuracy:** The percentage of GSM8K Test Set problems solved correctly (Single attempt).
2. **Win Rate:** In a head-to-head comparison, how often does the Nash Model beat the DPO Model?
3. **Stability Metric:** We will plot Accuracy vs. Training Iterations.
    - *Hypothesis:* DPO curve will plateau or drop (collapse). Nash curve will remain monotonic (steady increase).

---

6. Technical Stack Summary

Component	Technology Selection	Justification
Language	Python 3.10	Standard for ML.
Framework	PyTorch 2.4	Required for Unsloth optimization.
Optimization	**Unsloth** + **BitsAndBytes**	Essential for 4-bit loading on Tesla T4.
Training Lib	HuggingFace TRL	Provides base DPOTrainer class.
Tracking	Weights & Biases (WandB)	Visualization of loss curves.
Dataset	GSM8K	Gold standard for math reasoning.

---

7. System Interfaces & Deployment (Product Layer)

To demonstrate the practical applicability of the research, we will develop three production-grade interfaces.

7.1 The "Nash Arena" (Comparative Frontend)

A user-facing evaluation interface modeled after the LMSYS Chatbot Arena.

- **Purpose:** Qualitative evaluation of model reasoning.
- **Architecture:**
  - **Frontend:** React.js (Single Page Application).
  - **Backend:** FastAPI for handling inference requests.
- **Workflow:** Users input a math problem. The UI displays side-by-side solutions from the DPO Model and Nash Model (blinded). The user votes for the better reasoning chain.
- **Highlight:** Syntax highlighting for reasoning errors using diff-checking logic.

7.2 The "Training Command Center" (Observability Dashboard)

A real-time dashboard to visualize the self-improvement loop.

- **Purpose:** Monitoring data quality and training stability.
- **Tech Stack: Streamlit** (Python-native web app).
- **Key Visualizations**:
    - **Live Pass Rate:** Line chart updating every epoch.
    - **Data Explorer:** Table view of self-generated samples (Winner/Loser pairs) with filters for "False Positives" detected by the verifier.

7.3 High-Performance Inference API

A robust deployment architecture to serve the trained model.

- **Purpose:** Scalable model serving.
- **Tech Stack:**
    - **Containerization:** Docker (Standardized runtime).
    - **API Framework:** FastAPI (Async handling).
    - **Queue Management:** Celery + Redis (To handle request backpressure without OOM errors).
- **Mechanism:** The API implements a "Model Lifespan Manager" that loads the quantized model into GPU memory **once** at startup, preventing high-latency cold starts.