Optimizing nanoGPT for Multi-Digit Addition through Hardware, Algorithm, and MLOps Restructuring

Author: Lam Pham

Github Link: https://github.com/tlamDson/comp560-lamPham

Abstract

This study details a comprehensive suite of optimization strategies designed to achieve 100% mathematical accuracy in a character-level nanoGPT model. Challenged by the rigorous constraint of reducing convergence time to under 3 minutes on consumer-grade hardware, the research followed a dual-track methodology:

- **System Infrastructure & Algorithmic Refinement:** Initial efforts focused on eliminating systemic I/O bottlenecks and re-engineering algorithmic data flows to streamline processing efficiency.
- **Hyperbot-Driven MLOps:** An automated MLOps framework, titled Hyperbot, was developed to systematically navigate and optimize the hyperparameter space.

1. Introduction

Standard Transformer models frequently encounter significant obstacles in mathematical reasoning due to their inherent left-to-right sequence prediction, which contradicts the right-to-left "carry-over" logic fundamental to human arithmetic. Furthermore, training small-scale models on consumer-grade GPUs often falls into "Latency Bound" traps. These scenarios happen where hardware reports high utilization, yet the actual Model Flops Utilization (MFU) remains critically low due to coordination latency.

This study utilizes the nanoGPT framework to address these challenges in multi-digit addition problems ranging from 3 to 100 digits. By isolating and systematically removing specific bottlenecks across the entire pipeline, this research demonstrates the ability to maximize throughput and minimize training duration within the constraints of limited hardware resources. Central to this success is the development of "Hyperbot." Inspired by automated research workflows of Andrej Karpathy, this autonomous bot system was engineered to fully automate hyperparameter space exploration, dynamically adapting to each digit-addition complexity level.

2. Set Up: Bottleneck Resolution & Manual Optimization

2.1 Hardware Configuration

- **GPU:** NVIDIA RTX 4060 (8GB VRAM).
- **Environment:** Native Linux file system (ext4) running via WSL2 (executed through Windows, not native Linux).

2.2 Environment & Data Flow Optimization (I/O Bottleneck Resolution)

- **Elimination of File System Drift:** Migrating to ext4 on WSL2 bypassed NTFS protocol latency, reducing checkpoint serialization time from 5.5s to 1.1s.
- **Direct RAM Data Loading:** Replaced `np.memmap` with `np.fromfile` combined with `pin_memory()`. This mechanism avoids Page Faults and enables Direct Memory Access (DMA) for asynchronous data transfer to RAM.
- **Persistent GPU Inference:** Instead of re-initializing Python and CUDA for every test case, the model is loaded once to process inference in batches for all 100 test cases, compressing sampling time from several minutes to under 1 second.
- **RAM Disk Logging:** Terminal output streams were redirected to a RAM Disk (`/dev/shm`) to eliminate CPU rendering overhead.

2.3 Algorithmic & Data Breakthroughs

- **Reverse Target Generation:** Formatted target sequences in reverse (e.g., 7482 + 8517 = 99951). This simulates human mathematical intuition, allowing the model to manage carry variables more effectively.
- **Answer-Only Loss Masking:** Applied `ignore_index = -1` to the prompt portion of the sequence. The loss metric is calculated exclusively on the result, accurately reflecting mathematical reasoning capability.
- **Data Stratification:** Modified the `prepare.py` file for digit addition using a specialized distribution: 70% stratified random samples, 10% cascading carries, 10% length imbalance, and 10% edge cases. This ensured maximum training efficiency and robustness.

3. MLOps Automation with Hyperbot

3.1 Autonomous Hyperparameter Optimization System

- **Navigation Mechanism:** Designed an automated fine-tuning pipeline using the Optuna library to map configurations for the nanoGPT architecture.
- **Variable Mapping:** The system automatically navigates complex variables including the ratio of `n_embd` to `block_size`, `learning_rate`, `n_layer`, and `n_head`.
- **Bot Pruner:** Integrated an intelligent loss-monitoring algorithm. If the loss shows no significant breakthrough within the first 200 iterations, the process is immediately terminated to release GPU resources.

3.2 Multi-Phase Successive Halving

- **Warm-start:** Developed an orchestrator script to allocate time budgets across multiple stages.
- **Champion Extraction:** A dedicated Python script (`get_latest_champion.py`) automatically scans log directories to extract the `champion.json` file from previous stages, using it as a seed for subsequent runs.
- **System Overclocking:** In each successive phase, pruning conditions (e.g., `target_duration_sec`) are progressively tightened to force faster model convergence.

3.3 Scalable and Fault-Tolerant Design

- **Profile-Driven Design:** Eliminated all hard-coded values from the source code.
- **Unified Routing:** A Bash workflow (`safe_run.sh`) acts as the central router, while critical operational constraints are maintained in YAML configuration files.
- **CLI Scaling:** Scaling the research from 4-digit to 100-digit addition requires only a single CLI command.
- **SQLite Fault Tolerance:** Search states are saved in real-time to an SQLite database. An `--auto-resume` flag allows for the seamless recovery of the Optuna TPE (Tree-structured Parzen Estimator) analysis tree following interruptions such as crashes or Out-Of-Memory (OOM) errors.
- **Reporting:** All trial results are extracted to CSV files after each phase, providing transparent tracking of hyperparameter shifts across runs.

3.4 Convergence Records via Hyperbot

Section 3.4.1: 4-Digit Addition

The 4-digit milestone serves as the baseline to verify Hyperbot's optimization capabilities for extremely short token sequences. This stage highlights the strategic trade-off between batch size and network depth to maximize hardware efficiency.

1. Baseline Configuration and Performance

- n_layer: 4
- n_head: 8
- n_embd: 64
- batch_size: 1024
- learning_rate: 6e-3
- **Training Time (Baseline):** 15 seconds

2. Optimized Configuration (Champion)

- n_layer: 2 (50% reduction)
- n_head: 3 (62.5% reduction)
- n_embd: 96 (50% increase)
- batch_size: 1248 (22% increase)
- learning_rate: 3.47e-3
- **Average Training Time:** 12.55 seconds

3. Experimental Results (Bench Run Summary)

Trial	Real Time	Final MFU%	Accuracy%
Trial 1	9.689s	0.81	100.0
Trial 2	16.381s	0.63	100.0
Trial 3	10.802s	0.68	100.0
Trial 4	15.961s	0.69	100.0
Trial 5	9.934s	0.72	100.0
Average	**12.553s**	**0.71%**	**100.0%**

4. Optimization Logic and Analysis

- **Significant Time Reduction:** Average convergence time dropped by ~16%. Trial 1 established a record convergence of 9.68 seconds, demonstrating Hyperbot's ability to pinpoint the optimal parameter "sweet spot."
- **"Wide but Thin" Strategy:** Hyperbot identified that 4 layers were redundant for the logic required in 4-digit addition. By reducing to 2 layers, the model minimized computational volume per pass. To maintain representational power, it increased the embedding space (`n_embd`) to 96, enabling the model to capture carry rules more effectively.
- **VRAM Optimization for Short Sequences:** With a `block_size` of only 20, each data sample consumed minimal memory. Hyperbot pushed the `batch_size` to 1248 to maximize GPU workload per cycle, resulting in an MFU of 0.71%—the highest across all training milestones.
- **Absolute Accuracy:** The optimized configuration was not only faster but maintained 100% accuracy across all carries (0-4), proving that reducing heads and layers did not compromise the model's logical reasoning.

Section 3.4.2: 20-Digit Addition

The 20-digit milestone represents a low-medium difficulty problem where the model begins handling longer carry-over logic sequences. In this phase, Hyperbot executed a significant reversal in system resource restructuring.

1. Baseline Configuration and Performance

- `n_layer`: 4
- `n_head`: 8
- `n_embd`: 64
- `batch_size`: 1024
- `learning_rate`: 7e-3
- **Training Time (Baseline):** 32 seconds

2. Optimized Configuration (Champion)

- `n_layer`: 2 (50% reduction)
- `n_head`: 5 (37.5% reduction)

- n_embd: 160 (150% increase)
- batch_size: 248 (~75% reduction)
- learning_rate: 3.08e-3
- **Average Training Time:** 10.12 seconds

3. Experimental Results (Bench Run Summary)

Trial	Real Time	Final MFU%	Accuracy%
Trial 1	12.407s	0.15	99.5
Trial 2	10.813s	0.16	100.0
Trial 3	8.434s	0.14	100.0
Trial 4	10.980s	0.16	99.5
Trial 5	7.990s	0.16	99.5
Average	**10.125s**	**0.15%**	**99.7%**

4. Optimization Logic and Analysis

- **Speed Leap:** Convergence time plummeted from 32 seconds to 10.12 seconds (a 68% reduction). This is one of Hyperbot's most successful optimizations, bringing logic model training to a near-instantaneous level (under 10 seconds in Trials 3 and 5).
- **The Batch Size Paradox:** Unlike the 4-digit stage where batch size was increased, Hyperbot chose to sharply decrease the batch size from 1024 to 248 for the 20-digit stage.
- **Reasoning:** Hyperbot identified that oversaturating the batch for 20-digit sequences degraded the quality of gradient updates per step. By reducing batch size and significantly increasing embedding width (n_embd from 64 to 160), the model learned more "deeply" from specific examples, resulting in rapid convergence.

- **"Flat" Architecture Optimization:** Maintaining the n_layer=2 philosophy proved effective. Reducing depth significantly lowered computational costs on the RTX 4060, while the expanded 160-dimension embedding space provided sufficient room for attention vectors to accurately isolate digit positions in addition with carries.
- **Accuracy vs. MFU:** Although the MFU was reported as low (0.15%), this demonstrates the system's priority on Wall-clock time. Hyperbot favored an ultra-lightweight model to complete the task in 10 seconds over a heavier model that keeps the GPU busier but takes longer to reach the objective.

Section 3.4.3: 50-Digit Addition

At the 50-digit milestone, the problem enters a new threshold of sequence logic complexity. This is where Hyperbot demonstrates that optimization is not just about finding the fastest parameters, but about identifying the most stable architecture for handling long carry-over chains.

1. Baseline Configuration and Performance

- n_layer: 6
- n_head: 12
- n_embd: 192
- batch_size: 256
- learning_rate: 1e-3
- **Training Time (Baseline):** 3 minutes 50 seconds (230 seconds)
- **Characteristics:** A "heavy" model with numerous layers and attention heads, resulting in a slow backward pass and prolonged convergence times per epoch.

2. Optimized Configuration (Champion)

- n_layer: 2 (66% reduction)
- n_head: 7 (~42% reduction)
- n_embd: 224 (16% increase)
- batch_size: 304 (18% increase)
- learning_rate: 9.03e-3 (9x increase)
- **Average Training Time:** 47.71 seconds

3. Experimental Results (Bench Run Summary)

Trial	Real Time	Final MFU%	Accuracy%
Trial 1	55.932s	0.25	99.4
Trial 2	54.401s	0.25	99.4
Trial 3	50.483s	0.26	99.4
Trial 4	58.953s	0.25	99.4
Trial 5	38.781s	0.24	99.4
Average	**47.710s**	**0.25%**	**99.4%**

4. Optimization Logic and Analysis

- **Breaking the Time Barrier:** Hyperbot reduced convergence time from 230 seconds to an average of 47.7 seconds (nearly an 80% reduction). Trial 5 achieving 38.7 seconds proves that an ultra-thin model can capture mathematical logic features at high speed if provided with a sufficiently high learning rate (LR).
- **"Wide & Shallow" Strategy:** Instead of using the 6 layers seen in the baseline, Hyperbot strictly compressed the model to 2 layers. It compensated for processing power by increasing the embedding space (`n_embd`) to 224.
- **Reasoning:** For 50-digit addition, carry logic requires a wider vector space to isolate individual digits (units, tens, hundreds) without noise interference, but it does not require a deep stack of non-linear processing layers.
- **Accuracy Stability:** Despite the model being much thinner, the accuracy (Exact Match) remained stable at 99.4% across all five trials. This proves that a 2-layer network is entirely sufficient to resolve 50-digit addition logic if the attention configuration (7 heads) is properly tuned.
- **Throughput Optimization:** By slightly increasing the `batch_size` to 304, Hyperbot ensured the RTX 4060 GPU was constantly supplied with enough data per step without clogging the VRAM, maintaining a smooth and continuous training pace.

Section 3.5.4: 80-Digit Addition

At the 80-digit milestone, the challenges of maintaining attention across long carry-over sequences become more apparent than ever. Hyperbot had to recalibrate the network architecture to balance processing speed with mathematical accuracy.

1. Baseline Configuration and Performance

- n_layer: 6
- n_head: 8
- n_embd: 256
- batch_size: 256
- learning_rate: 1e-3
- **Training Time (Baseline):** 4 minutes 36 seconds (276 seconds)

2. Optimized Configuration (Champion)

- n_layer: 3 (50% reduction)
- n_head: 5 (37.5% reduction)
- n_embd: 160 (37.5% reduction)
- batch_size: 264 (slight increase)
- learning_rate: 8.4e-3 (8x increase)
- **Average Training Time:** 1 minute 5.5 seconds (65.56 seconds)

3. Experimental Results (Bench Run Summary)

Trial	Real Time	Final MFU%	Accuracy%
Trial 1	1m 29.2s	0.31	99.3
Trial 2	57.447s	0.29	99.3
Trial 3	59.146s	0.30	99.3
Trial 4	1m 07.7s	0.30	99.3
Trial 5	54.232s	0.31	99.3
Average	**65.563s**	**0.30%**	**99.3%**

4. Optimization Logic and Analysis

- **Record-Breaking Time Savings:** Hyperbot reduced convergence time from 276 seconds to 65.5 seconds (a reduction of nearly 76%). This is clear evidence that the Champion configuration can bring a time-intensive problem back to a stable and rapid operational level.
- **The 3-Layer Pivot:** Unlike the 4, 20, and 50-digit milestones (which typically used 2 layers), Hyperbot decided to increase the depth to 3 layers for the 80-digit task.
- **Reasoning:** As the carry logic extends to 80 positions, 2-layer networks begin to struggle with gradient stability. Adding an extra layer (Layer 3) provides an additional tier of abstraction to "verify" calculation results, maintaining high accuracy (99.3%) while remaining lightweight enough to run 4x faster than the baseline.
- **Narrowing Width for Speed:** In a surprising move, Hyperbot reduced `n_embd` from 256 to 160.
- **Strategy:** Rather than maintaining an excessively wide embedding that wastes computation, Hyperbot prioritized the speed of each iteration using leaner weight matrices, then compensated with an ultra-high learning rate (LR) to ensure early convergence.

- **Hardware Stability:** MFU remained stable at 0.30% across all five trials, indicating that this configuration found the "sweet spot" for the RTX 4060 on 80-digit problems: complex enough to solve the math accurately without being heavy enough to cause bottlenecks.

Section 3.4.5: 100-Digit Addition

The 100-digit milestone represents the ultimate optimization challenge. At this scale, the performance gap between the default configuration and the AI-tuned configuration becomes a massive divide in practical efficiency.

1. Baseline Configuration and Performance

- `n_layer`: 6
- `n_head`: 4
- `n_embd`: 128
- `batch_size`: 128
- `learning_rate`: 3e-3
- **Training Time (Baseline):** 6 minutes 14 seconds (374 seconds)

2. Optimized Configuration (Champion)

- `n_layer`: 2 (66% reduction)
- `n_head`: 6 (50% increase)
- `n_embd`: 192 (50% increase)
- `batch_size`: 216 (68% increase)
- `learning_rate`: 0.0115 (nearly 4x increase)
- **Average Training Time:** 44.30 seconds

3. Experimental Results (Bench Run Summary)

Trial	Real Time	Final MFU%	Accuracy%
Trial 1	35.845s	0.29	99.4
Trial 2	55.166s	0.29	99.7
Trial 3	43.672s	0.29	99.6
Trial 4	39.548s	0.29	99.7
Trial 5	47.282s	0.29	99.9
Average	**44.303s**	**0.29%**	**99.7%**

4. Optimization Logic and Analysis

- **Spectacular Performance Leap:** Hyperbot slashed convergence time from 374 seconds to 44.3 seconds. This record-breaking reduction (~88%) transformed a heavy computational task into a process completed in under a minute.
- **The "Ultra-Shallow" Strategy:** Despite the maximum difficulty of 100 digits, Hyperbot strictly maintained a minimal depth of `n_layer=2`. This reinforces the core philosophy: for addition logic, network depth primarily increases latency. Instead, Hyperbot increased `n_head` to 6 and `n_embd` to 192 to grant the model a broader "vision" over the 100-digit sequence.
- **Pushing Learning Rate to the Limit:** With such a thin model, Hyperbot dared to push the learning rate to an extreme 0.0115. This allowed the model to converge almost instantly, leveraging Early Stopping to lock in results at peak accuracy.
- **Large-Scale Stability:** Despite the immense data volume, the average accuracy remained exceptionally high at 99.7%. Precise performance across all carry positions (0 to 100) indicates that the model did not merely memorize data but truly mastered arithmetic rules, supported by Reverse Target Generation and Hyperbot’s optimal parameter set.

4. Discussion

The results of this study highlight a common misconception in training small-scale models on consumer GPUs: modern hardware is often "suffocated" by the operating system and data flow overhead long before it reaches its actual computational limits. The initial baseline of 6 minutes and 14 seconds did not reflect the true power of the GPU, but rather the inefficiency of I/O operations and kernel initialization latency.

From an algorithmic perspective, the implementation of Reverse Target generation proves that aligning model architecture with the underlying logic of the data is critical. The Transformer’s attention mechanism becomes significantly more efficient when the spatial distance for storing "carry-over" variables is minimized.

Crucially, a recurring architectural pattern observed across all digit-addition scales is the superiority of shallow-and-wide networks (e.g., `n_layer` =2 with a higher `n_embd`) over deeper architectures. This suggests that for strict logical carry-over tasks, network depth introduces unnecessary latency and gradient instability, whereas embedding width provides the requisite spatial capacity to resolve digits accurately. Ultimately, the application of MLOps through Hyperbot demonstrates that manual search processes can be entirely replaced by automation, discovering non-intuitive architectural setups that deliver results far exceeding the limits of human intuition.

5. Limitations

While the optimization strategies yielded significant improvements, this study is constrained by its specific hardware environment. All experiments were conducted on a single consumer-grade NVIDIA RTX 4060 (8GB VRAM) running via WSL2. Consequently, it remains undetermined whether the specific hyperparameter "sweet spots" discovered by Hyperbot are universally transferable to other microarchitectures (such as TPUs or Apple Silicon) or if they are intrinsically overfitted to NVIDIA's specific CUDA memory allocation and caching behaviors.

6. Conclusion

This study demonstrates that the perceived limitations of training arithmetic language models on consumer-grade hardware are primarily rooted in system-level bottlenecks and sub-optimal hyperparameters, rather than a raw compute deficiency. By implementing Reverse Target formatting and a stratified 70/10/10/10 data distribution, coupled with the Hyperbot MLOps framework, this research achieved dramatic performance gains. For the 100-digit addition task, training time was compressed from an initial 374 seconds to an average of 44.3 seconds, fundamentally shattering the project's 3-minute constraint.

7. Future Work

Building upon the Hyperbot framework and the optimized data pipeline, future research will focus on the following dimensions:

- **Algorithmic & Task Expansion:** The methodology will be applied to mathematical operations with higher spatial complexity, such as Multiplication and Division, to test the limits of the shallow-and-wide architectural hypothesis.
- **Hardware Diversity & Cloud Scalability:** Since the current framework is hyper-optimized for an NVIDIA RTX 4060, future deployments will benchmark performance across diverse hardware infrastructures, including CPU-only environments, TPUs, and distributed cloud clusters, to verify vertical and horizontal scaling capabilities.
- **Enhanced Observability & Scaling Laws:** To better understand the optimization process, the pipeline will be integrated with MLOps tracking tools like Weights & Biases (WandB). This will facilitate the creation of empirical "Training Curves" or Scaling Laws, visually mapping the Pareto frontier of average convergence time against task complexity (from 4 to 100+ digits).