**Summary:**  
In this project, we began with a simple Python training script for an autoregressive text generation model, using GPT as a template, trained on the julien040/hacker-news-posts dataset from Hugging Face. The dataset consists of Hacker News titles, which were split into training and validation sets with a fixed seed and ratio. The model was trained to predict the next token in the training titles and evaluated on the validation set.

Over the past two weeks, I introduced major changes to the codebase, including restructuring, implementation of new methods, integration of training optimization techniques, monitoring tools, and sweep functionality. These improvements were made to ensure the feasibility and robustness of training and testing, while also enhancing readability and aligning the repository with best practices in machine learning development.

This report also includes supporting research and justifications for the approaches taken, particularly regarding minimizing validation loss.

**Problem Statement**

The initial iteration of model training revealed several critical challenges that limited development efficiency and model performance:

1. **CPU-Only Computational & Manual Environment Constraints**
   * Training a ~27-million-parameter model on CPU required approximately 7.6 hours per run, severely restricting the ability to perform multiple experiments or hyperparameter sweeps. [see Appendix A for details]
   * Manual environment setup (configuring paths, running Docker containers, modifying hyperparameters for optimization) added extra overhead and increased the risk of errors.
2. **Suboptimal Model Performance**
   * Both the GPT-2-based architecture and the training pipeline required further optimization to reduce validation loss, improve predictive accuracy, and ensure efficient convergence. [see Appendix A for details]
3. **Non-Modular Codebase**
   * The existing code lacked structure and modularity, complicating the integration of new components, architectures, or attention mechanisms. [see Appendix B for details]
4. **Inefficient Hyperparameter Exploration**
   * Long runtimes and manual setup made systematic evaluation of different hyperparameter configurations impractical. [see Appendix B for details]
5. **Absence of Monitoring and Logging Mechanisms**
   * The training process lacked a standardized monitoring system, reducing visibility into metrics, loss curves, and experiment reproducibility.

**Literature review**

**Scaling Laws for Neural Language Models** [**https://arxiv.org/pdf/2001.08361**](https://arxiv.org/pdf/2001.08361)

Introduction

Language models have become central to modern natural language processing tasks. Understanding how their performance scales with model size, dataset size, and compute is crucial for efficient resource allocation and effective model design. This report explores the empirical findings of scaling laws for training neural language models, particularly focusing on factors that influence performance.

Methodology

The study analysed the impact of several key factors on model performance using the WebText dataset:

* Model size (N): Number of non-embedding parameters in the network.
* Dataset size (D): Total number of training tokens.
* Model shape: Width and depth of the network.
* Context length: Number of previous tokens considered.
* Batch size: Number of examples per training iteration.

Experiments systematically varied these factors to observe their effect on test loss and learning efficiency.

Useful Findings

1. Performance depends strongly on scale, weakly on model shape
   * Increasing model size and dataset size significantly reduces test loss, while changing model shape (depth vs. width) has a smaller impact.
2. Smooth power-law relationships
   * Test loss improves predictably following smooth power laws as model size (N), dataset size (D), and compute (C) increase.
3. Joint scaling of model and data
   * To maximize performance, model size (N) and dataset size (D) must be scaled together. Scaling only one leads to diminishing returns.
4. Sample efficiency of larger models
   * Larger models reach the same performance levels with fewer optimization steps, making them more sample-efficient than smaller models.
5. Compute efficiency
   * When neither model size nor dataset size is bottlenecked, increasing the compute (C) directly improves test loss.

Summary:  
Given the dataset size (D) is limited in this assessment, the first step in optimization is to increase compute and scale up model parameters, consistent with the key findings from the scaling laws paper. This strategy allows us to explore performance improvements until we either hit GPU VRAM constraints or encounter diminishing returns due to overfitting. While the ideal batch size was not explicitly calculated as suggested in the paper, sweeping hyperparameters and testing GPU memory limitations provides a practical approach. Given the hardware constraints, batch sizes larger than 256 would restrict the maximum model size that can be trained effectively.

**Generating Long sequences with sparse transformer** [**https://www.alphaxiv.org/abs/1904.10509**](https://www.alphaxiv.org/abs/1904.10509)

The reason I chose this paper is that currently there is a growing number of MoE models, including Grok and DeepSeek, which are among the most computationally efficient. The core layer of MoE typically uses sparse attention as its attention mechanism. It provides me with the opportunity to explore sparse attention in depth and investigate further variations of sparse attention mechanisms.

This paper introduces Sparse Transformers, a modified Transformer architecture designed to efficiently process much longer sequences than traditional Transformers. The core innovation lies in introducing sparse factorizations of the attention matrix, which reduces the quadratic to time and memory complexity of standard self-attention to . This is achieved by separating the full attention computation into several faster attention operations that, when combined, approximate the dense attention operation.

There are two variants of attention layer explored : strided attention and fixed attention patterns. These variants achieve state-of-the-art results in density modeling across diverse data types, including images, raw audio, and natural language. They demonstrate the ability to model sequences tens of thousands of timesteps long, generate globally coherent samples, and even show the potential to model sequences of length one million or more, all while requiring significantly fewer operations compared to standard Transformers.

**Abstract Level Overview:**

A diagram of a model

AI-generated content may be incorrect.

Functional Unit 1: **Automation & Configuration**

Problem addressed: **CPU-Only Computational & Manual Environment Constraints**

Short summary:

This unit automates and manages the execution of training and experimentation workflows. By leveraging GitHub Actions, Docker, and GPU acceleration, it ensures reproducible environments, streamlined setup, and efficient monitoring, effectively serving as a lightweight MLOps layer for rapid iteration and reliable experiment orchestration.

**Subunits:**

* **GPU Status Check** – monitors GPU availability and memory usage to ensure resources are ready for training.
* **Training Functions** – executes model training in a reproducible, containerized environment.
* **Sweep Functions** – automates hyperparameter tuning through W&B sweeps with full logging and tracking.
* **Update Mechanism** – keeps the self-hosted runner synchronized with the latest codebase, enabling smoother CI/CD workflows.

Functional Unit 2: Configuration Management

Problem: **Inefficient Hyperparameter Exploration**

Short Summary:

This unit centralizes all experiment settings, including base hyperparameters, sweep configurations, and best-performing (SOTA) configurations, ensuring reproducibility, clarity, and easy management of training experiments without relying on manual flags or ad-hoc changes.

**Subunits:**

* **All experiment settings**, including:
  + Sweep configurations
  + Base hyperparameters
  + SOTA configuration

Functional Unit 3: Train pipeline

Problem: **Absence of Monitoring and Logging Mechanisms, Non-Modular Codebase**

Short Summary:

This unit orchestrates end-to-end training by reading hyperparameters from YAML configuration files and setting up runs accordingly. It dynamically interacts with the **Model Core** to build models with the requested architecture and attention layers. The pipeline supports flexible schedules, optimizers, and training optimization techniques, while integrating monitoring library for monitoring and logging, making training modular, reproducible, and adaptable to new experiments.

**Subunits:**

* **Hyperparameter Loader –** Reads base hyperparameters, sweep configurations, and SOTA configs from YAML files.
* **Model Integrator –** Passes configuration to Model Core to build the appropriate architecture and attention layers**.**
* **Dynamic Training Manager** – Supports multiple schedulers, optimizers, and mixed-precision training.
* **Monitoring & Logging** – Integrates W&B to track metrics, losses, and experiment results in real time.

Functional Unit 4: Model Core

Problem: **Non-Modular Codebase, Suboptimal Model Performance**

Short Summary:

The Model Core contains the fundamental components of the system, including the architecture, attention mechanisms, and tokenizer. It serves as the computational engine, dynamically building models according to configurations provided by the Training Pipeline. The modular design allows for easy integration of new architectures, attention variants, and tokenization strategies.

**Subunits:**

* **Architecture –** Defines the backbone of the model (e.g., GPT, UNet) and manages the forward pass.
* **Attention Mechanisms –** Implements different attention layers, enabling experimentation with variants.
* **Tokenizer –** Handles preprocessing and tokenization of input text, converting raw data into model-ready tokens.

**List of function units and subunits**

|  |  |
| --- | --- |
| FU1 [Automation & Configuration] | |
| FU1.1 | GPU status check |
| FU1.2 | Training functions |
| FU1.3 | Sweep functions |
| FU1.4 | Update mechanism |
| FU2 [Configuration Management] | |
| FU3 [Train pipeline] | |
| FU3.1 | Configurable Training Framework |
| FU3.2 | Monitoring & Logging |
| FU4 [Model Core] |  |
| FU4.1 | Architecture |
| FU4.2 | Attention Mechanisms |
| FU4.3 | Tokenizer |

**FU1.1: GPU Status Check**

Added workflow:

* .github/workflows/checknvidia.yml *(See Appendix C for code and output)*

**Functionalities:**

* Performs manual checks to ensure GPU availability before starting training.
* Verifies that GPU memory is sufficient for the training session and confirms memory is released after training.

**FU1.2–1.3: Training and Sweeping Functions**

*(See Appendix D for code)*

Added workflows:

* .github/workflows/train.yml
* .github/workflows/sweep.yml

**Functionalities:**

* Update repository: Ensures the code is up-to-date before training. *(Steps: update repo)*
* Train models / hyperparameter sweeps: Uses Weights & Biases (W&B) for experiment tracking.
* GPU utilization in Docker: Runs training and sweep agents with --gpus all for faster computation. *(Steps: Run Sweep Agent in Docker in sweep.yml; Run training in Docker in train.yml)*
* Automatic environment setup: Configures the environment to ensure reproducibility and reduce manual errors. *(Lines 27–30: setup commands)*

**FU1.4: Update Mechanisms**

*(See Appendix E for code)*

**Changes:**

* **Added workflow:** .github/workflows/update.yml

**Functionalities:**

* Automatically updates the runner’s code.
* Triggers on every commit to ensure the latest version is always used.

FU2 Configuration Management

(see Appendix F for hierarchy and code sample)

Changes:

* Added YAML files to manage and test different configurations.
* Separated hyperparameters into distinct segments and categorized parameters for clarity.
* Added parameters:
  + **Hyperparameters:** Scheduler, optimizer , amp\_bool (enable torch.amp to minimise memory )
  + **Model configuration:** Hidden layer size (intermediate layer dimension within Block class), attention layer (specifies which attention mechanism is used)
  + **Attention configuration:** Sparse attention parameters (attn\_type – type of sparse attention, num\_verts, local\_attn\_ctx)
* Added sota\_config folder to store the current state-of-the-art model configuration, ensuring reproducibility and easy reference for experiments.

**Functionalities:**

* Structured configuration for easy modification and clean implementation
* Historical configuration tracking
* Parameter sweeping support for experiments

**FU3.1** Configurable Training Framework

**(see appendix G for code)**

**Changes:**

* **Train.py**:
  + Convert configuration files into dictionaries, enabling direct passing into classes without additional processing.
  + Selection of optimizer and scheduler.
  + Added parser arguments:
    - --sweep and --original\_yaml specify configuration YAMLs:
      * **Sweep configuration:** Path for parameter sweeping.
      * **Original YAML:** Path for base hyperparameters, usually updated with results from sweeps to improve performance over time.
    - --test and --sweep specify the type of run:
      * **Testing:** Allows dynamic import or removal of training-monitoring and Docker-related functionality.
      * **Sweep:** Activates W&B sweep functionality for hyperparameter optimization.
  + Segment for dynamically selecting the attention layer and architecture, informing the model’s core functional unit during construction.

**FU3.2 Monitoring and Logging**

**Changes / Functionalities:**

* **W&B Integration:**
  + **Added wandb.init to specify runs and log hyperparameters.**
  + **Added wandb.log to track training and validation loss over time.**
  + **Implemented wandb.sweep for automated hyperparameter exploration and updated the configuration dictionary for input parameters.**
* **Logger Enhancements:**
  + **Added logger.log to record model parameters and all hyperparameters for the current run.**
  + **Logs are output to both the terminal and W&B.**
  + **Log files are also uploaded to W&B for centralized tracking and visualization.**

**Minor change :**

* **train.py & train.sh:** Standardized training entry points for Python and shell environments.
* **Taskfile.yml:** Replaced direct python3 train.py calls with train.sh for easier command management and reproducibility.
* **hyperparam\_class.py:** Added for clearer hyperparameter management and readability.

**Extensive Data Analysis**

The dataset was thoroughly explored before training to understand its structure and tokenization behavior. Key steps and findings include:

* **Dataset Inspection**
  + Loaded both training and validation titles.
  + Checked dataset lengths before and after tokenization.
  + Printed the first 10 train\_ids for verification.
* **Longest Title Strings**
  + Training set: longest string length = **98** characters, found at index **6099** with title:  
    *"Rough silicon nanowires potentially allow much more efficient waste-heat to electricity conversion"*
  + Validation set: longest string length = **91** characters, found at index **229** with title:  
    *"Official Google Blog: 'This site may harm your computer' on every search result??"*
* **Longest Tokenized Title**
  + Training set: maximum token length = **68**, found with the title:  
    *"Z͌̈́̾a͊̈́l͊̿g̏̉͆o̾̚̚S̝̬ͅc̬r̯̼͇ͅi̼͖̭͔͜p̲̘̘̹͖t̠͖̟̹͓͇ͅ"* (visual fuzzing text with non-standard characters).
  + Validation set: maximum token length = **44**, found with the title:  
    *"Bangladeshi model Farhana Akhtar Nisho (ফারহানা আখতার নিশ্) hot and sexy photo"*
  + **Observation:** Titles containing non-English or visually obfuscated characters are harder to encode with BPE, which leads to longer tokenization sequences.
* **Vocabulary and Token Statistics**
  + Unique words:
    - Training: **25,516**
    - Validation: **4,177**
    - Combined: **27,707**
    - Slightly higher than training alone since BPE concatenates characters rather than recognizing full words.
  + Unique tokens:
    - Training: **14,403**
    - Validation: **4,669**
    - Combined: **14,751**
    - This is just below the defined vocabulary size parameter (**16,000**), which corresponds to the output dimension of the model’s final layer.
  + Tokens in validation not seen in training: **348**.

**4. Model Enhancements**

* **Attention Layers:** Implemented two attention variants, with plans to add a third for comparative analysis.
* **Tokenizer Improvements:** Refined tokenization process for better dataset handling.
* **Architecture Extensions:** Supported both GPT and U-Net style architectures, increasing flexibility.

**Training Performance & Feasibility**

With **CPU-only training**, a model of **27,172,864 parameters** required **27,403 s (~7.6 h)** for a full run. This runtime severely constrained experimentation: hyperparameter sweeps were impractical, and at most ~3 full trainings could be completed in 24 hours.

Switching to an **RTX 3090 GPU** reduced the end-to-end runtime for **7 epochs** to **~100 s (~1.67 min)**, i.e., **~14.3 s/epoch**, a **~274× speedup**.

**Implications:**

* **Hyperparameter Sweeps:** From nearly infeasible to easily completed within hours.
* **Architecture Iteration:** Enables rapid testing of new backbones and attention layers.
* **Research Velocity:** From one run taking a day to interactive prototyping.
* **Monitoring & QA:** Faster runs allow tighter feedback loops for loss curves, overfitting checks, and ablations.

**Notes for Fair Comparison:** Batch size, sequence length, data pipeline, precision (fp32 vs mixed precision), and seed were held constant; GPU runs can further benefit from **AMP** and **DataLoader optimizations**.

**Bottom Line:** The GPU reduces training from ~7.6 hours to ~1.7 minutes, making hyperparameter sweeps and rapid architectural experimentation feasible within the project’s two-week timeframe.

Appendix A

Run initial run

<https://github.com/tedasdf/mainrun/actions/runs/17524976010/job/49774162371>

Appendix B

**Codebase Structure and Hyperparameter Management**

* **Non-modular codebase:** The train.py script contains multiple components—model definition, tokenization, and causal self-attention—within a single file, making navigation and maintenance more difficult.
  + **Key components (for reference ):**
    - Lines 114–127: BPETokenizer class
    - Lines 138–160: CausalSelfAttention class
    - Lines 162–171: MLP class
    - Lines 173–183: Block class
    - Lines 185–218: GPT class
* **Inefficient hyperparameter exploration:** Hyperparameters are clustered within train.py, limiting flexibility for experimentation and automated sweep workflows.
  + **Key components (for reference):**
    - Lines 16–32: Hyperparameters class
* **Logging Limitations**
* **Uncomprehensive logging:** Although the logger records loss, the lack of visualization tools and difficulty in comparing runs make it suboptimal for systematic model optimization and hyperparameter exploration.
  + **Key components (for reference):**
    - Lines 225–226: Logger initialization
    - Lines 232, 229, 246, 263, 295, 304: Logging of device used, dataset info, model parameters, training step, loss, elapsed time, validation step, loss, and elapsed time
    - Lines 34–76: DualLogger class keeps track of loss over time

Note: The full codebase is available at [GitHub link](https://github.com/MaincodeHQ/mainrun/blob/main/mainrun/train.py).

Appendix C

Code for check nvidia

A computer screen shot of a program

AI-generated content may be incorrect.

Workflow page

A screen shot of a computer

AI-generated content may be incorrect.

Appendix D

Code for Train Model

A screenshot of a computer program

AI-generated content may be incorrect.

Code for Run Sweep

A screenshot of a computer program

AI-generated content may be incorrect.

Appendix E

Code for Auto Update Code

A screenshot of a computer program

AI-generated content may be incorrect.

Appendix F

Hierarchy of Configuration Management

A screenshot of a computer

AI-generated content may be incorrect.

Hyperparameter sample

A screen shot of a computer program

AI-generated content may be incorrect.

Sweep configuration

A screenshot of a computer program

AI-generated content may be incorrect.