

Maincode skill Assessment

LLM training and optimisation

Teddy Lo

Contents

Problem Statement	2
Related Works:	3
Training Paradigm of Language Models	3
Scaling Laws for Language Models [1]	3
An Empirical Model of Large-Batch Training [2]	4
Attention Variants	4
Code Design:	5
Abstract Level Overview:	5
List of function units and subunits	8
Experiment:	11
Data Analysis[6]	11
Result	12
Initial iteration: <i>(see Appendix J)</i>	12
Second iteration: <i>(see appendix K)</i>	12
Third iteration: <i>(See appendix L)</i>	12
Fourth iteration: <i>(See appendix M)</i>	13
Fifth iteration: <i>(See appendix N)</i>	13
Sixth iteration: <i>(See appendix O)</i>	14
Seventh iteration: <i>(See appendix P)</i>	14
Appendix	15

Introduction

In this project, we started 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 contains Hacker News titles, which were split into training and validation sets using a fixed seed and ratio. The model was trained to predict the next token in the titles and evaluated on the validation set.

Over the past two weeks, significant improvements were introduced to the codebase, including restructuring, implementation of new methods, integration of training optimization techniques, monitoring tools, and sweep functionality. These changes enhanced training feasibility and robustness, improved code readability, and aligned the repository with best practices in machine learning development. The report also provides supporting research and justification for the approaches, particularly in 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 Computation & Manual Environment Constraints** *(see Appendix A for information)*
 - Training a ~27-million-parameter model on CPU required ~1 hours per run, restricting the ability to perform multiple experiments or hyperparameter sweeps.
 - Manual environment setup (configuring paths, Docker containers, and hyperparameters) added overhead and increased the risk of errors.
- 2. Suboptimal Model Performance** *(see Appendix A for information)*
 - Both the GPT-2-based architecture and the training pipeline required further optimization to reduce validation loss, improve predictive accuracy, and ensure efficient convergence.
- 3. Non-Modular Codebase**
 - The existing code lacked structure and modularity, complicating the integration of new components, architectures, or attention mechanisms.
- 4. Inefficient Hyperparameter Exploration** *(see Appendix B for information)*
 - Long runtimes and manual setup made systematic evaluation of different hyperparameter configurations impractical.
- 5. Absence of Monitoring and Logging Mechanisms** *(see Appendix B for information)*
 - The training process lacked a standardized monitoring system, reducing visibility into metrics, loss curves, and experiment reproducibility.

Related Works:

Training Paradigm of Language Models

To optimize the performance of a language model, it is essential to understand the **training paradigm**. This includes how performance scales with **model size**, **dataset size**, and **compute resources**, as well as the influence of **batch size** on training dynamics. Mastery of these concepts provides the foundation for systematically exploring and testing model parameters.

Scaling Laws for Language Models [1]

This study examines the impact of several key factors on model performance using the WebText dataset:

- Model size(N)
- Dataset size (D)
- Compute (C) [Approximately $\approx 6 \times N \times \text{Batch_Size} \times \text{Number_of_optimisation_steps}$]

The key findings are:

1. Performance depends strongly on scale, weakly on model shape

- Increasing model size and dataset size significantly reduces test loss, while adjusting model shape (depth vs. width) has less effect.

2. Smooth power-law relationships

- Test loss decreases predictably according to power-law scaling with respect to model size (N), dataset size (D), and compute (C).

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 achieve the same loss levels in fewer optimization steps, making them more sample efficient.

5. Compute efficiency

- When neither model size nor dataset size is the bottleneck, additional compute (C) yields predictable improvements in test loss.

Key takeaway: Model performance is governed by scaling laws. For any fixed dataset size and compute, there exists an **optimal model size**, too small underfits, too large wastes capacity, which can be found by systematic sweeps

An Empirical Model of Large-Batch Training [2]

A complementary study on the training paradigm of language models emphasizes the **empirical law of large-batch training**:

- **Gradient noise scale:** Batch size directly influences the variance (noise) in gradient estimates.
- **Exploration vs. convergence:**
 - Smaller batch sizes introduce more noise, encouraging the model to explore local minima.
 - Larger batch sizes reduce noise, allowing faster convergence but at the cost of potentially poorer generalization.
- **Dataset dependence:** The optimal batch size depends primarily on dataset size rather than model size.

In this project, these insights motivated systematic exploration of batch size under hardware constraints. Although the mathematical tools from the paper to precisely estimate optimal batch size were not applied, an initial **sweep test of batch size** was conducted as the first attempt at optimization.

Attention Variants

Speaking of current LLM architectures, **Mixture-of-Experts (MoE) models** are widely used, with examples including DeepSeek R1 and Grok. Sparse attention is commonly applied in MoE models, as it can **maintain or even improve performance** compared to full causal attention. By leveraging **sparse factorization**, sparse attention significantly **reduces memory and computation costs**, making it more efficient for large-scale models.

Generating Long Sequences with Sparse Transformers [3]

Sparse Attention [3] is a mechanism that reduces memory and computation costs to $O(n\sqrt{n})$. Two factorized schemes are commonly used: **strided** and **fixed**. In strided attention, one head attends to a local window of the previous l tokens, while another head attends to every l -th token further back, capturing both short- and long-range dependencies. In fixed attention, one head focuses on tokens within the same block of size l , while the other attends to a fixed set of “summary” positions in each block, efficiently summarizing local context. These mechanisms allow the model to retain important information without the quadratic cost of full self-attention. According to the paper, **fixed attention is particularly beneficial for text generation tasks**, improving efficiency while maintaining or enhancing performance.

Sparser is Faster and Less is More: Efficient Sparse Attention for Long-Range Transformers [4]

The core idea of this paper, SparseK Attention Layer, is to select a **constant number of key-value (KV) pairs** per query, resulting in **linear time complexity** and a **constant memory footprint** during generation.

Key components:

- **Scoring Network:** output the importance score of each KV pair without needing access to all queries.
- **Differentiable Top-k Mask Operator (SPARSEK):** Produces a soft, differentiable mask over KV pairs, enabling gradient-based optimization for efficient, learnable attention.

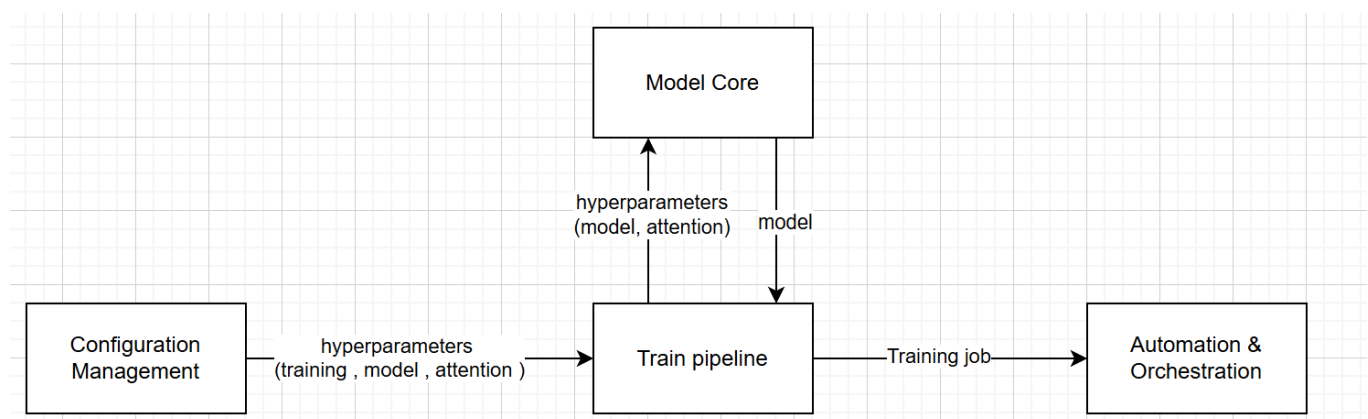
Although SPARSEK Attention theoretically supports a fully linear-time formulation, my current implementation only realizes a reduced-complexity variant with $O(m \log(m))$ efficiency, where m is the number of key-value pairs.

This approximation preserves most of the computational advantages of SPARSEK while offering a more practical implementation pathway. However, the SparseK operator itself was not fully implemented, which prevented me from achieving the exact theoretical linear-time performance.

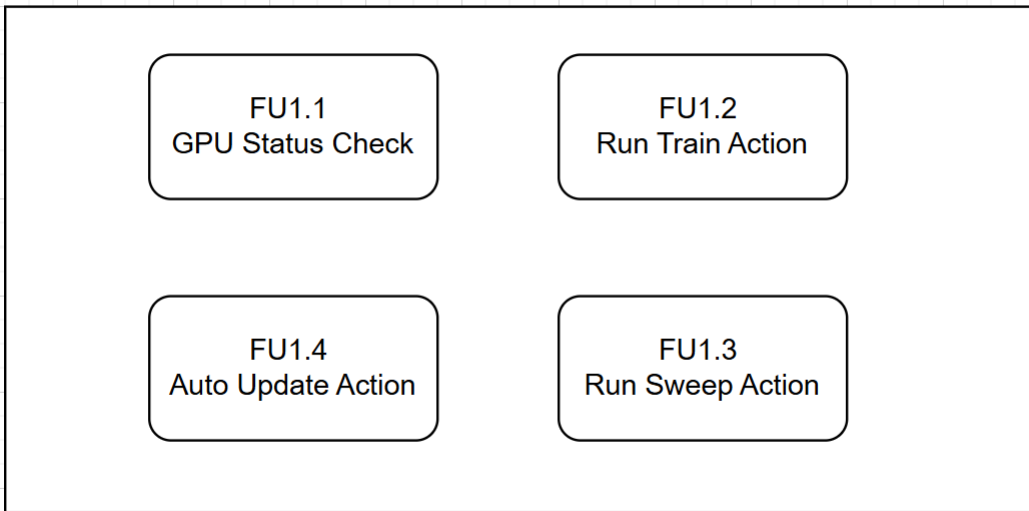
Future work could focus on integrating the full SparseK operator to bridge this gap and unlock the maximum efficiency promised by the method. (

Code Design:

Abstract Level Overview:



FU1 Automation & Configuration



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:

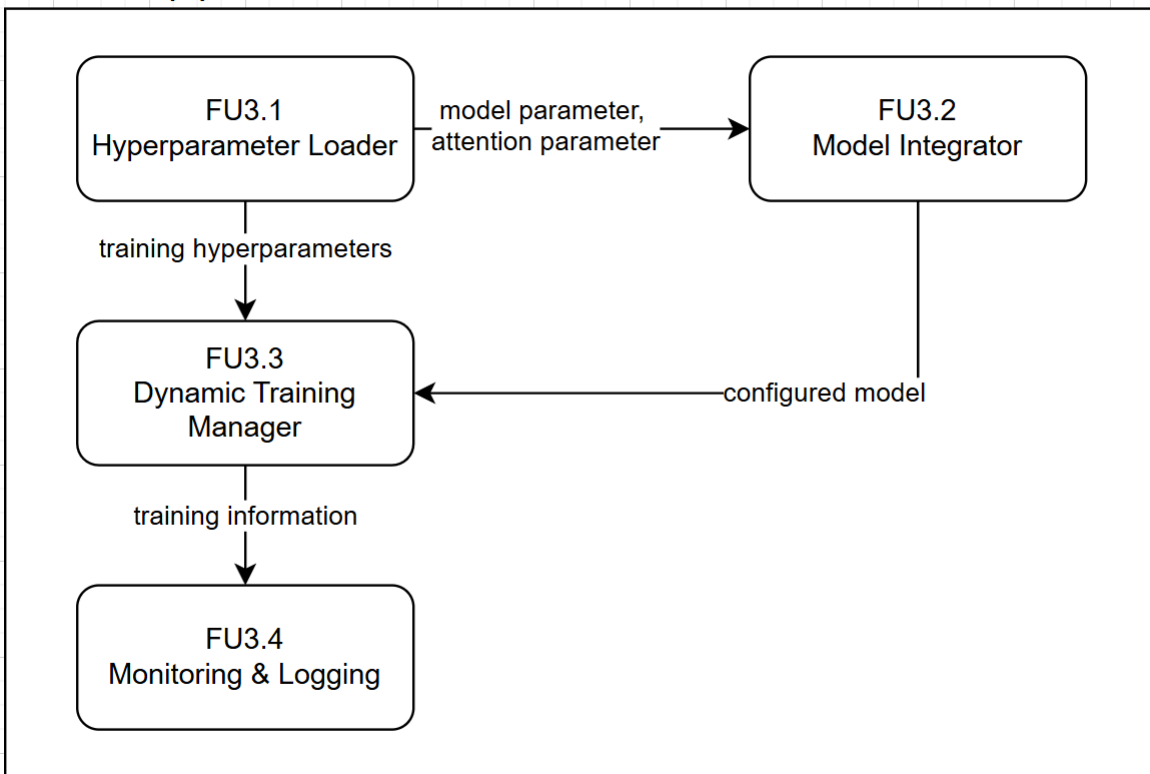
This unit centralizes all experiment settings, including base hyperparameters, sweep configurations, and best-performing (SOTA) configurations, ensuring reproducibility, clarity, and easy management of training experiments without relying on manual flags or ad-hoc changes.

Subunits:

- **All experiment settings**, including:
 - Sweep configurations
 - Base hyperparameters

- SOTA configuration

FU3 Train pipeline



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 system's fundamental components—architecture, attention mechanisms, and tokenizer. It functions as the computational engine, dynamically building models according to configurations provided by the **Training Pipeline**. A modular design is critical, enabling flexible integration of new architectures, attention variants, and tokenization strategies.

Subunits:

- **Memory Estimation** – Estimates memory usage per layer and overall model, helping detect and prevent out-of-memory errors.
- **Configurable Models and Layers** – Provides flexible configuration classes that control architecture parameters and manage dependencies across layers.
- **Model Organisation** – Establishes a clear class hierarchy to structure models, making the codebase easier to extend and maintain.

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]	

FU1.1: GPU Status Check *(See Appendix C for code and output)*

Changes:

- **Added workflow:** `.github/workflows/checknvidia.yml`

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)*

Changes:

- **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.

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 and optimization techniques
 - **Model configuration:** architectural parameters
 - **Attention configuration:** attention parameters
- 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 (in train.py)

- 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.
- Selection of optimizer, scheduler and training optimization technique.
- Segment for dynamically selecting the attention layer and architecture, informing the model's core functional unit during construction.
- **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:** hyperparameter class as a standalone component in hyperparam_class.py for clearer hyperparameter management and readability.

Functionalities:

- Centralized and flexible training configuration.
- Supports dynamic model and attention layer selection.
- Enables reproducible experiments and hyperparameter sweeps.
- Improves readability and maintainability of training scripts.

FU3.2 Monitoring and Logging *(see Appendix H for code)*

Changes (in train.py)

- Integrated Weights & Biases (W&B) for experiment tracking.
- Configured runs with hyperparameters logging
- Logged training/validation losses over time.
- Added wandb.sweep for automated hyperparameter search and updated configuration dictionary.

Functionalities

- Provides centralized experiment tracking.
- Enables hyperparameter optimization via sweeps.
- Visualizes metrics to compare training performance.

FU4 Model Core (see Appendix I for code)

Changes (in model folder):

- Added AttnConfig and GPTConfig to declare architecture and attention layer variables.
- GPUnetT (inherit from GPT class) and sparse attention (inherit from causal attention class)
- Memory estimation function for each layer and architecture
- Configurable setting (including weight initialisation, norm type etc in architecture and masking for different sparse attention scheme)

Functionalities:

- Centralizes architecture hyperparameters.
- Ensures consistent parameter propagation across layers.
- Facilitates modular experimentation with different configs.

Experiment:

Data Analysis [6]

Before training, the dataset was thoroughly explored to understand its structure, tokenization behaviour, and potential challenges. Key steps and findings are summarized below:

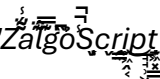
1. Dataset Inspection

- Loaded both training and validation titles.
- Checked dataset lengths before and after tokenization.
- Verified the first 10 training IDs to ensure correct loading.

2. Longest Title Strings

- **Training set:** Longest string = 98 characters, at index 6099:
"Rough silicon nanowires potentially allow much more efficient waste-heat to electricity conversion"
- **Validation set:** Longest string = 91 characters, at index 229:
"Official Google Blog: 'This site may harm your computer' on every search result??"

3. Longest Tokenized Titles

- **Training set:** Maximum token length = 68, title: *"ZalgoScript"*

 - Non-standard characters increase token length due to BPE encoding.
- **Validation set:** Maximum token length = 44, title: *"Bangladeshi model Farhana Akhtar Nisho (ফারহানা আখতার নিশ) hot and sexy photo"*

- **Observation:** Titles with non-English or visually obfuscated characters are harder to encode with BPE, resulting in longer token sequences.

Vocabulary and Token Statistics

	Unique words	Unique tokens
Training	25,516	14,403
Validation	4,177	4,669
Combined	27,707	14,751

- Note:

- The number of unique words in combined is slightly higher than training alone since BPE concatenates characters rather than recognizing full words.
- The number of unique tokens 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**.

Result

Initial iteration: (see *Appendix J*)

Observing the training loss graph, there is noticeable noise, indicating variability in gradient estimates likely caused by a relatively small batch size for this dataset. While this is not inherently problematic for training neural networks, given the limited number of epochs, the current batch size may not be optimal. Therefore, the first step is to determine the optimal batch size for this dataset, motivating a systematic sweep over different batch sizes.

Secondly, the validation loss is slightly above the baseline, suggesting that there is room for improvement in the current architecture. The validation loss generally follows the same trend as the training loss; however, it is important to note that the training loss ends at 8.32, while the validation loss is 1.78, indicating a substantial gap between the two.

Second iteration: (see *appendix K*)

After sweeping the batch size, the validation loss decreases as the batch size increases. This suggests that an ideal batch size for this dataset is 256, which was subsequently included as a sweep parameter in later GPT experiments alongside other hyperparameters. Larger batch sizes make it easier for the model to approach a global minimum, as the model can consider more information at each step and better estimate the general direction of the gradient. Notably, the validation loss for batch size 256 starts around 1.8, compared to approximately 2 for smaller batch sizes. Additionally, the noisiness observed in the training loss decreases as the batch size increases, reflecting a lower gradient noise ratio.

Third iteration: (See *appendix L*)

Sweep GPT parameters

In this sweep, the validation loss reached **1.27**, a significant improvement over the original **1.78**. The sweep tested batch sizes of 128 and 256, but only the runs with batch size 128 appeared in the dashboard, as all runs with batch size 256 crashed due to GPU out-of-memory errors. Scaling up the GPT model improved performance, supporting a key finding from the scaling laws: performance scales strongly with model size. The top 30 runs achieved parameter counts between **3,667,200 and 12,059,648**, with the best performance observed at **6–8 layers**. Models with either higher or lower numbers of layers performed worse, illustrating another central point of the scaling laws: under a fixed dataset size, increasing model size beyond a certain point lead to a performance bottleneck unless the dataset is also scaled accordingly.

Fourth iteration: *(See appendix M)*

Sweep UNet parameters

Based on experience from previous runs, I attempted to reduce memory usage by adopting variable dimensions in a GPT-UNet-like architecture, aiming to decrease the number of parameters and enable a batch size of 256. However, this approach was unsuccessful. To diagnose the issue, I implemented memory estimation functions to monitor GPU usage and confirmed that the model's memory footprint itself was within bounds. While the **n_layer depth remained within 6–8**, consistent with earlier findings, the variable-dimension design significantly degraded performance, with the best validation loss plateauing around **1.55**. Overall, this sweep neither provided useful insights nor resolved the memory bottleneck observed in earlier experiments.

Fifth iteration: *(See appendix N)*

Sweep sparse attention

During exploratory data analysis (EDA) of the dataset and task, it became evident that **full causal self-attention** was not always necessary for this text generation problem. Key observations included:

- **Local dependencies dominate** – Predictions mainly rely on relationships among nearby tokens within a title.
- **End-of-sequence classification** – Generation largely reduces to deciding whether to end the sequence or extend it, which does not require global context.
- **Independence across samples** – Each title is independent, so modelling long-range dependencies across samples is unnecessary.

Given these properties, full causal self-attention introduces **quadratic complexity** without meaningful performance benefits. In contrast, **sparse attention** focuses compute on relevant local dependencies, offering a more efficient solution.

A parameter sweep on sparse attention was conducted using a not-yet fully optimized GPT model. Validation losses ranged between **1.25 and 1.285**, demonstrating a measurable improvement in efficiency and performance. The best results were obtained with an **intermediate dimension of 64 and 8 attention heads**, even though these values were not at the edges of the tested parameter ranges.

Based on these results, the best-performing sparse attention configuration was integrated into the GPT configuration for subsequent training runs.

Sixth iteration: (See [appendix O](#))

Building on the success of sparse attention in the unoptimized GPT runs, I selected one of the top-performing parameter configurations and integrated it into the GPT architecture. By sweeping both **model parameters** and **batch sizes**, I achieved a validation loss of **0.94**—a measurable improvement over previous runs.

However, attempts with a batch size of **256** frequently failed due to **out-of-memory (OOM) errors**. This highlighted that the **fundamental memory constraint remained unresolved**, even with sparse attention providing computational efficiency.

To investigate further, I analysed inference and training memory usage. During memory usage analysis, I consulted the *Ultrascale Playbook* (Hugging Face Spaces) which indicates that in many large-scale training setups, the majority of GPU memory is taken up by optimizer state rather than the model's parameter storage. This insight guided my decision to focus on mixed precision, optimizer-level tweaks, and limiting batch size to avoid OOM errors.

As a result, I adopted a **batch size of 128** for stable runs while continuing to explore **optimizer-level techniques** (e.g., state sharding, memory-efficient optimizers, mixed precision) to further reduce memory usage and unlock larger-scale training.

Seventh iteration: (See [appendix P](#))

During training, out-of-memory (OOM) errors initially limited the feasible model size and batch configuration. Through research, I identified two effective strategies to mitigate this issue:

1. **Environment configuration** – Setting `PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True` allowed CUDA memory segments to grow dynamically, reducing fragmentation.
2. **Automatic mixed precision (AMP)** – Enabling `torch.amp` lowered memory requirements by storing activations in lower precision while maintaining numerical stability.

Together, these techniques alleviated the OOM bottleneck and enabled training of a **3-million-parameter model with a batch size of 256**.

By scaling the batch size from its original setting to **256**, the model successfully converged to a **validation loss of 0.945**, while maintaining stable performance. This improvement provided significantly more flexibility for experimentation and highlighted the importance of GPU memory optimizations in achieving efficient training.

Future Implementation:

- Adapt `test.ipynb` to integrate a **SPARSEK attention layer** for improved efficiency in long-sequence modeling.

- Explore and incorporate additional **memory optimization techniques** (e.g., optimizer state sharding, gradient checkpointing, mixed precision) to enable training with larger batch sizes and deeper models.

References

- [1] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*. <https://arxiv.org/abs/2001.08361>
- [2] McCandlish, S., Kaplan, J., & Amodei, D. (2018). *An Empirical Model of Large-Batch Training*. arXiv preprint arXiv:1812.06162. <https://www.alphaxiv.org/abs/1812.06162>
- [3] Child, R., Gray, S., Radford, A., and Sutskever, I. (2019). Generating Long Sequences with Sparse Transformers. *arXiv preprint arXiv:1904.10509*.
- [4] Chao Lou, Zixia Jia, Zilong Zheng, and Kewei Tu. "Sparser is Faster and Less is More: Efficient Sparse Attention for Long-Range Transformers." *arXiv preprint arXiv:2406.16747*, 2024.
- [6] <https://github.com/tedasdf/mainrun/blob/main/mainrun/EDA.ipynb>

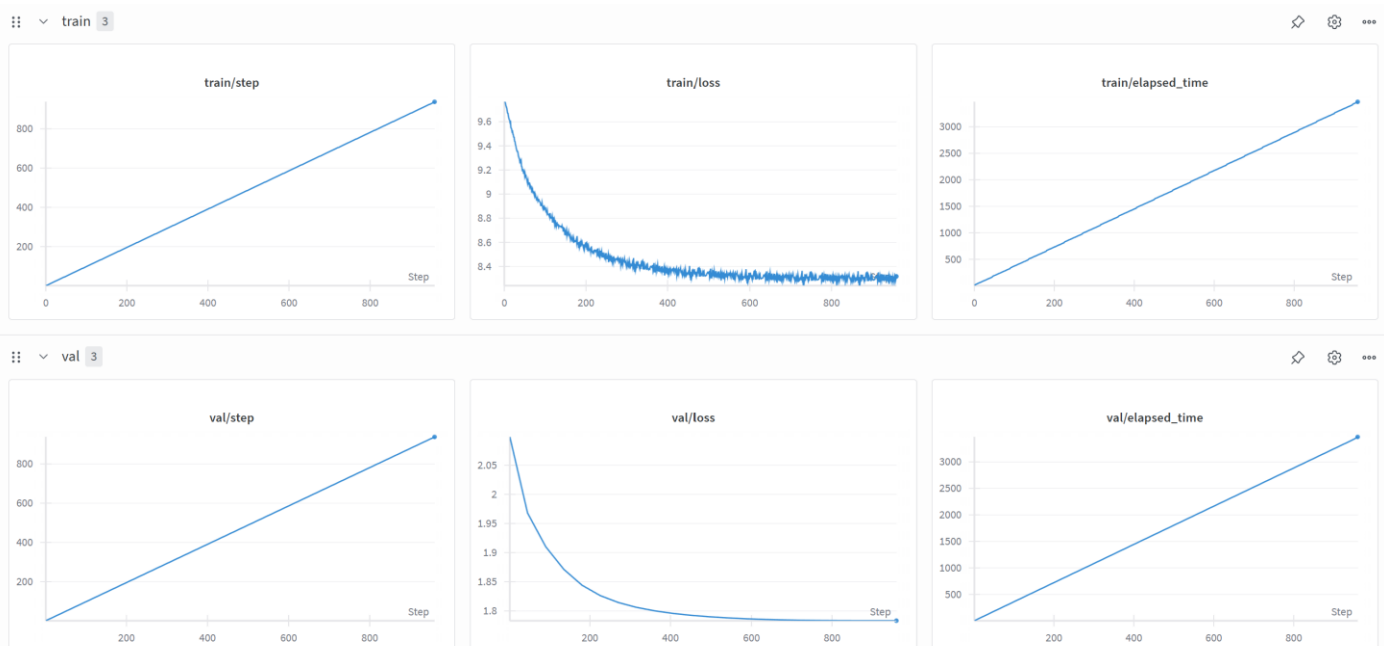
Appendix

Appendix A

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

hyperparameters_configured: seed=1337, epochs=7, val_frac=0.1, num_titles=100000, vocab_size=16000, context_length=128, log_file=./logs/mainrun.log, model_architecture=gpt, batch_size=64, lr=0.005, weight_decay=0.1, scheduler=cosine, optimizer=sgd, evals_per_epoch=3, amp_bool=False, device=cpu

device_info: device=cpu , [938/938] validation_step: loss=1.783221 time=3473.00s



Appendix B

<https://github.com/MaincodeHQ/mainrun/blob/main/mainrun/train.py>

Inefficient Hyperparameter Exploration

```
@dataclass
class Hyperparameters:
    block_size: int = 128
    batch_size: int = 64
    vocab_size: int = 16_000
    n_layer: int = 6
    n_head: int = 8
    d_model: int = 512
    dropout: float = 0.1
    lr: float = 6e-3
    weight_decay: float = 0.0
    evals_per_epoch: int = 3

    epochs: int = 7
    seed: int = 1337
    num_titles: int = 100_000
    val_frac: float = 0.10
    log_file: str = "./logs/mainrun.log"
```

Absence of monitoring and visualisation

```
def configure_logging(log_file: str):
    Path(log_file).parent.mkdir(parents=True, exist_ok=True)

    file_handler = open(log_file, 'w')

    structlog.configure(
        processors=[
            structlog.stdlib.filter_by_level,
            structlog.stdlib.add_logger_name,
            structlog.stdlib.add_log_level,
            structlog.stdlib.PositionalArgumentsFormatter(),
            structlog.processors.TimeStamper(fmt="iso"),
            structlog.processors.StackInfoRenderer(),
            structlog.processors.format_exc_info,
            structlog.processors.UnicodeDecoder(),
            structlog.processors.JSONRenderer()
        ],
        context_class=dict,
        logger_factory=structlog.stdlib.LoggerFactory(),
        cache_logger_on_first_use=True,
    )

    class DualLogger:
        def __init__(self, file_handler):
            self.file_handler = file_handler
            self.logger = structlog.get_logger()
```



```

def log(self, event, **kwargs):
    log_entry = json.dumps({"event": event, "timestamp": time.time(), **kwargs})
    self.file_handler.write(log_entry + "\n")
    self.file_handler.flush()

    if kwargs.get("prnt", True):
        if "step" in kwargs and "max_steps" in kwargs:
            tqdm.write(f"[{kwargs.get('step'):>5}/{kwargs.get('max_steps')}]")
        {event}: loss={kwargs.get('loss', 'N/A'):.6f} time={kwargs.get('elapsed_time', 0):.2f}s"
        else:
            parts = [f"{k}={v}" for k, v in kwargs.items() if k not in ["prnt",
"timestamp"]]

            if parts:
                tqdm.write(f"{event}: {' '.join(parts)}")
            else:
                tqdm.write(event)

    return DualLogger(file_handler)

logger = None

```

Appendix C

Code for check nvidia

```

name: check nvidia

on:
  workflow_dispatch:    # Trigger manually

jobs:
  sweep:
    runs-on: [self-hosted, x64, linux, docker]

    steps:
      - name: Update repo
        run: |
          cd /home/labadmin/Documents/mainrun
          git reset --hard
          git pull origin main

      - name: check nvidia in Docker
        run: |
          docker run --rm --gpus all \
            -v /home/labadmin/Documents/mainrun:/workspace \
            -w /workspace/mainrun \
            --env-file /home/labadmin/Documents/mainrun/.env \
            -e WANDB_PYTHON_EXECUTABLE=/usr/bin/python3 \
            -e PYTHONPATH=/workspace/mainrun \
            mainrun-env \
            bash -c "
              ps aux | grep wandb
            "

```

```
ps aux | grep python
nvidia-smi --query=compute-apps=pid,process_name,used_memory --format=csv

nvidia-smi"
```

Workflow page

The screenshot shows a GitHub Actions workflow page for a job named 'check nvidia #13'. The workflow is titled 'sweep' and has a status of 'succeeded now in 8s'. The job 'check nvidia in Docker' is expanded, showing its logs. The logs include the command 'Run docker run --rm --gpus all \', followed by the execution of 'nvidia-smi' and 'ps aux | grep python'. The output of 'nvidia-smi' is a table of GPU information, and the output of 'ps aux | grep python' is a list of running processes.

GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC
0	NVIDIA GeForce RTX 3090	On	00000000:01:00.0	Off	N/A

GPU	CI	PID	Type	Process name	GPU Memory Usage
No running processes found					

Check Nvidia Action: <https://github.com/tedasdf/mainrun/actions/workflows/checknvidia.yml>

Appendix D

Code for Train Model (due to the similarity of the code of both sweep and train , only code of train model is displayed. Those are separated as different actions for organisation and clarity)

```
name: Train Model

on:
  workflow_dispatch: # Run manually

jobs:
  train:
    runs-on: [self-hosted, x64, linux, docker]

    steps:
      - name: Update repo
        run: |
          cd /home/labadmin/Documents/mainrun
          git reset --hard
          git pull origin main
```

```

- name: Run training in Docker
  run: |
    docker run --rm --gpus all \
      -v /home/labadmin/Documents/mainrun:/workspace \
      -w /workspace \
      --env-file /home/labadmin/Documents/mainrun/.env \
      -e PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True \
      mainrun-env-new \
        bash -c " git config --global --add safe.directory /workspace && \
                  git config --global user.email 'teedsingyau@gmail.com' && \
                  git config --global user.name 'Ted Lo' && \
                  task train"

```

Train Model Action: <https://github.com/tedasdf/mainrun/actions/workflows/train.yml>

Sweep Action: <https://github.com/tedasdf/mainrun/actions/workflows/sweep.yml>

Appendix E

Code for Auto Update Code

```

name: Auto Update Code

on:
  push:
    branches:
      - main    # run only when you push to main

jobs:
  update:
    runs-on: [self-hosted, x64, linux, docker]    # your self-hosted runner
    steps:
      - name: Pull latest code safely using PAT
        env:
          GITHUB_TOKEN: ${ secrets.PAT_TOKEN }    # PAT stored as secret
        run: |
          cd /home/labadmin/Documents/mainrun
          git remote set-url origin https://$GITHUB_TOKEN@github.com/tedasdf/mainrun.git
          git fetch origin
          git reset --hard origin/main
          git clean -fd -e .env

```

Auto Update Code Action: <https://github.com/tedasdf/mainrun/actions/workflows/update.yml>

Appendix F

Hierarchy of Configuration Management

- ! hyperparams_gpt_sparse.yaml
 - ! hyperparams_unet.yaml
 - ! hyperparams.yaml
 - ! original_hyperparams.yaml
 - ! testhyperparams.yaml
 - ! gpt_sparse_1.08.yaml
 - ! model_gpt_sparse.yaml
 - ! model_vallos=1.27.yaml
 - ! sweep_gpt_sparse.yaml
 - ! sweep_gpt.yaml
 - ! sweep_unet.yaml
 - ! training.yaml

Sample Sweep configuration

```

program: train.py
method: random # or grid, or bayes
metric:
  name: val/loss
  goal: minimize
parameters:
  hyperparams.lr:
    values: [ 0.02 , 0.017, 0.012]
  hyperparams.batch_size:
    values: [ 64, 128 , 256]
  hyperparams.context_length:
    values: [256]
  hyperparams.optimizer:
    values: ["adamw", "adagrad"]
  hyperparams.weight_decay:
    values: [ 0.1 , 0.15, 0.3, 0.5]
  model_configs.gpt.dropout:
    values: [0.1, 0.3 , 0.4]
  model_configs.gpt.d_model:
    values: [ 128, 256]
  model_configs.gpt.n_layer:
    values: [ 6, 8]

  model_configs.gpt.init_method:
    values: ['xavier' , 'normal' , 'kaiming', 'uniform']

```

Hyperparameter sample

```

hyperparams:
#####
# FIXED hyperparameters
#####
seed: 1337
epochs: 7
val_frac: 0.10

```

```
num_titles: 100000
vocab_size: 16000 # Vocabulary size of the tokenizer
context_length: 256
#####
# CHANGEABLE hyperparameters
#####
model_architecture: "gpt" # gpt, unet_gpt
log_file: "./logs/mainrun.log"
#####
# Training hyperparameters
#####
batch_size: 256
lr: 0.02
weight_decay: 0.5
scheduler: "cosine" # none, linear, cosine
optimizer: "adamw"
evals_per_epoch: 3
amp_bool: True
```

model_configs:

```
gpt:
  d_model: 128
  hidden_layer: 128
  n_layer: 8
  dropout: 0.1
  init_method: 'xavier'
  attention_layer: 'sparse'
UNET_gpt:
  d_model: 512
  hidden_layer: 128
  n_layer: 6
  dropout: 0.1
  init_method: 'normal'
  attention_layer: 'causal'
  bottleneck_sizes: [512, 256, 256, 128, 128, 256]
```

attn_configs:

```
causal:
  n_head: 8
  intermediate_dim : 0 # if 0 , normal , elif > 0 , bottleneck
sparse:
  attn_type: 'fixed' # 'all', 'fixed', 'local'. 'strided'
  n_head: 8
  num_verts: 16 #
  local_attn_ctx: 32
  sparseblocksize: 128
  vertsize: 128
  n_bctx: 4
  intermediate_dim : 64 # if 0 , normal , elif > 0 , bottleneck
```

```
# UNET d_model: 512
```

```
# n_head: 8
# n_layer: 6
# dropout: 0.1
# bottleneck_size: 256
# unet_gpt:
```

Appendix G:

Parser Argument:

```
import torch

torch.cuda.empty_cache()
load_dotenv(dotenv_path=".env")

parser = argparse.ArgumentParser()
parser.add_argument("--test", action="store_true", help="Run test")
parser.add_argument("--sweep", action="store_true", help="Run hyperparameter sweep")
parser.add_argument("--sweep_config", type=str, help="Path to sweep YAML config")
parser.add_argument("--orig_yaml", type=str, default="config/hyperparams.yaml")
args = parser.parse_args()
```

Selection of optimizer and scheduler

```
### Optimizer and Scheduler
if args.optimizer == "sgd":
    opt = torch.optim.SGD(model.parameters(), lr=args.lr,
weight_decay=args.weight_decay)
elif args.optimizer == "adamw":
    opt = torch.optim.AdamW(model.parameters(), lr=args.lr,
weight_decay=args.weight_decay)
elif args.optimizer == "adam":
    opt = torch.optim.Adam(model.parameters(), lr=args.lr,
weight_decay=args.weight_decay)
elif args.optimizer == "adagrad":
    opt = torch.optim.Adagrad(model.parameters(), lr=args.lr,
weight_decay=args.weight_decay)
elif args.optimizer == 'RMSprop':
    opt = torch.optim.RMSprop(model.parameters(), lr=args.lr ,
weight_decay=args.weight_decay)
else:
    raise ValueError(f"Unsupported optimizer: {args.optimizer}")

if args.scheduler == "cosine":
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(opt, T_max=max_steps)
elif args.scheduler == "linear":
    scheduler = torch.optim.lr_scheduler.LinearLR(opt, start_factor=1.0,
end_factor=0.0, total_iters=max_steps)
elif args.scheduler == "step":
    scheduler = torch.optim.lr_scheduler.StepLR(opt, step_size=args.step_size,
gamma=args.gamma)
elif args.scheduler == "none":
```

```

        scheduler = torch.optim.lr_scheduler.LambdaLR(opt, lr_lambda=lambda step: 1.0)
    else:
        raise ValueError(f"Unsupported scheduler: {args.scheduler}")

```

Selection of attent layer and architecutre

```

### Attention setup
if modelparams['attention_layer'] == 'causal':
    attn = AttnConfig(
        d_model=modelparams['d_model'],
        block_size=args.context_length,
        dropout=modelparams['dropout'],
        **attnparams
    )

elif modelparams['attention_layer'] == 'sparse':
    attn = SparseAttnConfig(
        d_model=modelparams['d_model'],
        block_size=args.context_length,
        dropout=modelparams['dropout'],
        **attnparams
    )

#### Model setup
if args.model_architecture == "gpt":
    cfg = GPTConfig(
        vocab_size=args.vocab_size,
        block_size=args.context_length,
        attn_config = attn,
        activation_function = 'gelu',
        **modelparams
    )
    model = GPT(cfg).to(device)
elif args.model_architecture == "unet_gpt":
    cfg = UnetGPTConfig(
        vocab_size=args.vocab_size,
        block_size=args.context_length,
        attn_config = attn,
        activation_function='gelu',
        **modelparams
    )
    model = GPUUnetT(cfg).to(device)
else:
    raise ValueError(f"Unsupported model architecture: {args.model_arhitecture}")

```

Training optimization technique sample

```
ptr = 0
step = 0
t0 = time.time()

scaler = GradScaler()
for epoch in range(1, args.epochs + 1):
    for _ in tqdm(range(1, batches + 1), desc=f"Epoch {epoch}/{args.epochs}"):
        step += 1
        xb, yb, ptr = get_batch(train_ids, ptr, args.context_length, args.batch_size,
device)

        if args.amp_bool:
            with autocast(): # enables float16 for eligible ops
                _, loss = model(xb, yb)

            # Backward with gradient scaling
            scaler.scale(loss).backward()

            # Gradient clipping (scale before unscale!)
            scaler.unscale_(opt) # important for clip_grad_norm_

            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

            # Optimizer step
            scaler.step(opt)
            scaler.update()

            # Scheduler step (unchanged)
            scheduler.step()
        else:
            _, loss = model(xb, yb)

            # l1_norm = sum(p.abs().sum() for p in model.parameters())
            # l2_norm = sum(p.pow(2).sum() for p in model.parameters())

            # loss = loss + l1_norm * L1 + l2_norm * L2
            opt.zero_grad(set_to_none=True)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
            scheduler.step()
```

Appendix H:

sweep

```
def sweep_train():
    orig_cfg = OmegaConf.load(args.orig_yaml) # defaults
    cfg = copy.deepcopy(orig_cfg) # create a separate copy to modify
```



```

with wandb.init() as run:
    print("RUNCONFIG")
    print(dict(run.config))

    print("Before")
    print(cfg)

    for key, val in dict(run.config).items():
        parts = key.split(".") # split by all dots
        d = cfg
        # traverse down to the last dictionary
        for p in parts[:-1]:
            d = d[p]
        d[parts[-1]] = val # set the value

    print("After applying sweep")
    print(cfg)
    main(cfg , False)

if not args.test:
    wandb.login(key=os.getenv("WANDB_API_KEY"))
    import utils

if args.sweep:

    cfg = OmegaConf.load(args.sweep_config)
    # Convert to a plain dictionary
    cfg_dict = OmegaConf.to_container(cfg, resolve=True)

    sweep_id = wandb.sweep(cfg_dict, project="gpt-from-scratch", entity="arc_agi")
    wandb.agent(sweep_id, function=sweep_train , count=50)

```

Wandb logging

```

if not test:
    wandb.init(
        project="gpt-from-scratch",
        entity="arc_agi",
        config=hparams # <--- pass hyperparams to W&B
    )

```

```

logger.log("training_step",
           step=step,
           max_steps=max_steps,
           loss=loss.item(),
           elapsed_time=elapsed,
           prnt=False)

if not test:
    wandb.log({

```

```

        "train/loss": loss.item(),
        "train/step": step,
        "train/elapsed_time": elapsed
    })

    if step == 1 or step % eval_interval == 0 or step == max_steps:
        val_loss = evaluate()
        logger.log("validation_step",
                   step=step,
                   max_steps=max_steps,
                   loss=val_loss,
                   elapsed_time=elapsed)

        if not test:
            wandb.log({
                "val/step" : step,
                "val/loss": val_loss,
                "val/step": step,
                "val/elapsed_time": elapsed
            })

    if not test:
        artifact = wandb.Artifact("logs" , type="log")

        artifact.add_file(args.log_file)
        wandb.log_artifact(artifact)
        wandb.finish()

```

Appendix I

```

@dataclass
class ModelConfig:
    vocab_size: int
    context_length: int
    batch_size: int
    embed_dim: int
    dropout: float
    n_layers: int

@dataclass
class GPTConfig:
    vocab_size: int
    block_size: int
    n_layer: int

```

```

d_model: int
dropout: float
attn_config : AttnConfig
hidden_layer : int
attention_layer: str
norm_type: str = 'pre' # 'pre' or 'post'
activation_function: str = 'gelu' # 'relu' or 'gelu'
init_method: str = 'xavier'

class GPT(nn.Module):
    def __init__(self, cfg: GPTConfig):

@dataclass
class UnetGPTConfig(GPTConfig):
    hidden_layer_list: List[int] = None

class GPUUnetT(GPT):
    def __init__(self, cfg: UnetGPTConfig):

```

```

@dataclass
class AttnConfig:
    d_model: int
    n_head: int
    block_size: int
    dropout: float
    intermediate_dim: int # must be specified for bottleneck attention

@dataclass
class SparseAttnConfig(AttnConfig):
    attn_type: str # 'fixed_sparse' or 'strided_sparse'
    num_verts: int
    local_attn_ctx: int
    sparseblocksize: int
    vertsize: int
    n_bctx: int

class CausalSelfAttention(nn.Module):

class SparseCausalSelfAttention(CausalSelfAttention):

```

memory function

```

def memory_before_inference(self, dtype=torch.float32):
    elem_size = torch.tensor([], dtype=dtype).element_size()

```

```

total_mem = 0

# Token embedding
total_mem += self.token_emb.weight.numel() * elem_size

# Positional embedding
total_mem += self.pos_emb.numel() * elem_size

# Dropout has no persistent parameters

# Blocks
for i, block in enumerate(self.blocks):
    if hasattr(block, "memory_before_inference"):
        block_mem = block.memory_before_inference(dtype)
    else:
        block_mem = sum(p.numel() * elem_size for p in block.parameters())
    print(f"Block {i+1} memory: {block_mem / (1024**2):.3f} MB")
    total_mem += block_mem

# Final LayerNorm
total_mem += sum(p.numel() * elem_size for p in self.ln_f.parameters())

# Head
total_mem += self.head.weight.numel() * elem_size

print(f"Total GPT memory before inference: {total_mem / (1024**2):.3f} MB")
return total_mem / (1024**2) # MB

```

example of configurable setting (weight initialisation)

```

@staticmethod
def _init_weights(module, cfg):
    """Initialize weights based on cfg.init_method."""
    if isinstance(module, (nn.Linear, nn.Embedding)):
        if cfg.init_method == "normal":
            nn.init.normal_(module.weight, mean=0.0, std=0.02)
        elif cfg.init_method == "xavier":
            nn.init.xavier_normal_(module.weight)
        elif cfg.init_method == "kaiming":
            nn.init.kaiming_normal_(module.weight, mode='fan_in', nonlinearity='relu')
        elif cfg.init_method == "uniform":
            bound = 1.0 / (cfg.d_model ** 0.5) # Scaled by sqrt(d_model)
            nn.init.uniform_(module.weight, -bound, bound)
        else:
            raise ValueError(f"Unknown init_method: {cfg.init_method}")

    if isinstance(module, nn.Linear) and module.bias is not None:
        nn.init.zeros_(module.bias)

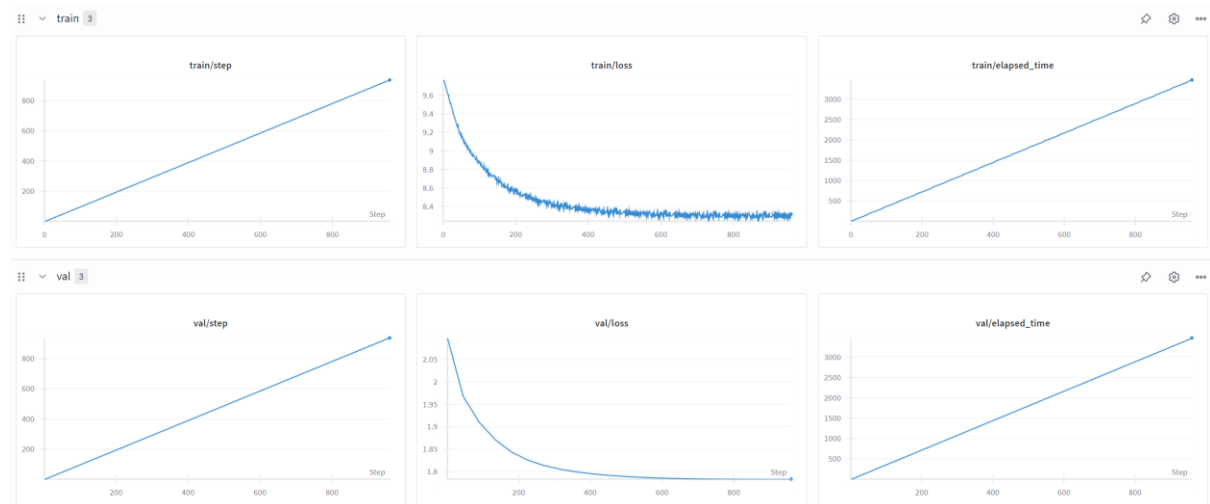
```

Appendix K

Configuration

https://github.com/tedasdf/mainrun/blob/main/mainrun/config/hyperparameter_config/hyperparams.yaml

Wandb display

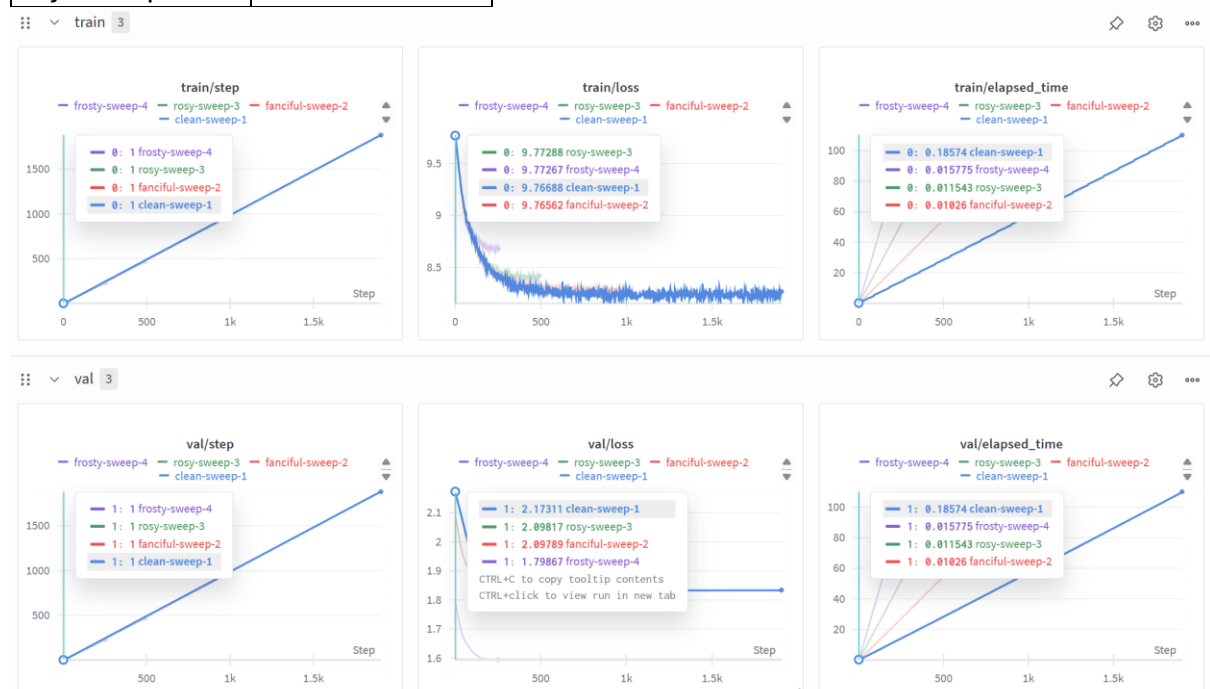


Model

validation loss: [938/938] validation_step: loss=1.783221 time=3473.00s

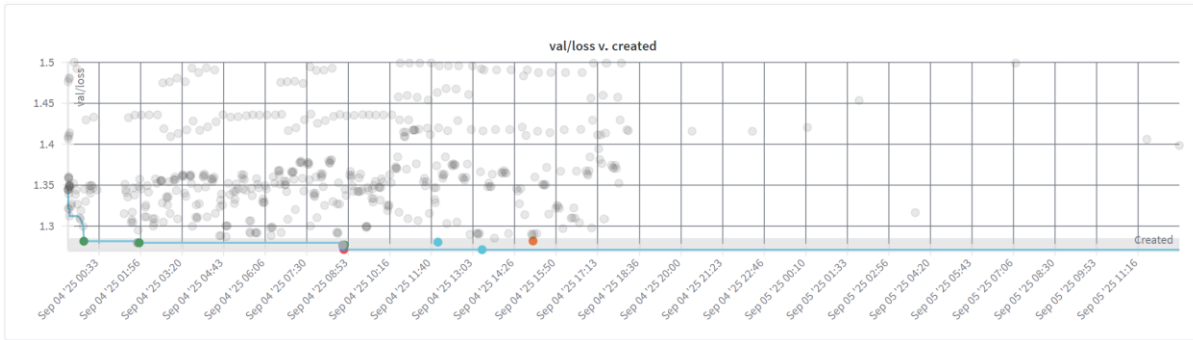
Appendix K

	Batch size
young-sweep-1	32
eager-sweep-2	64
noble-sweep-3	148
icy-sweep-4	256

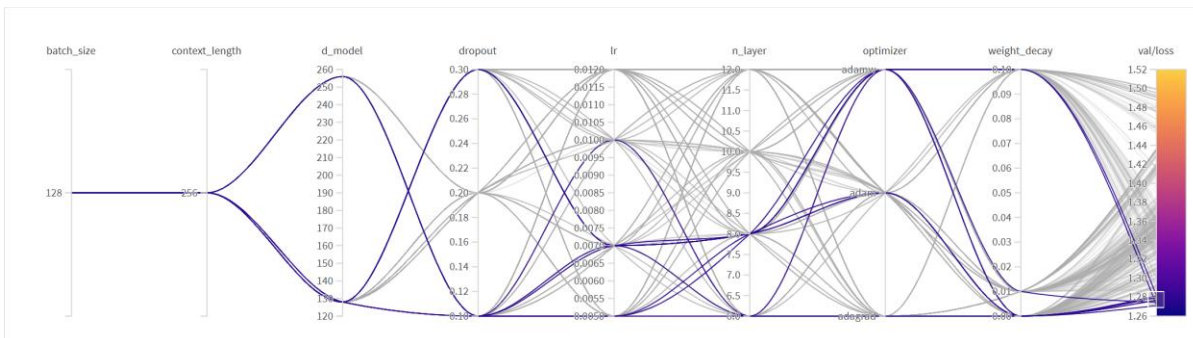


Appendix L

Validation loss distribution



Sweep distribution diagram

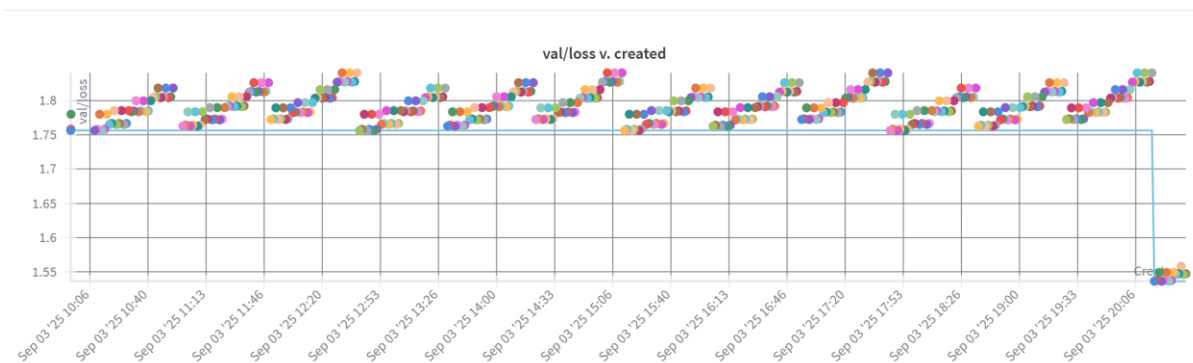


Sweep Configuration

https://github.com/tedasdf/mainrun/blob/main/mainrun/config/sweep_config/sweep_gpt_old.yaml

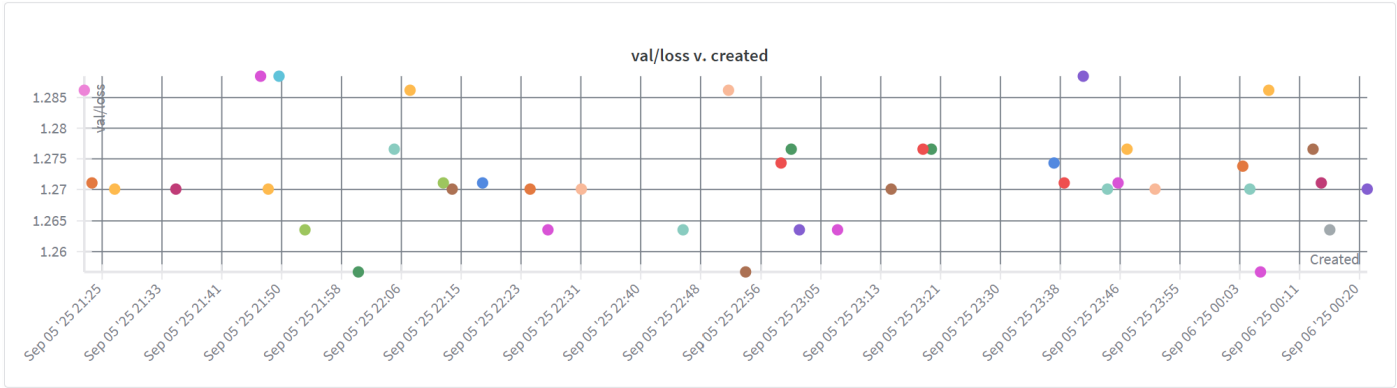
Appendix M

https://github.com/tedasdf/mainrun/blob/main/mainrun/config/sweep_config/sweep_unet_old.yaml



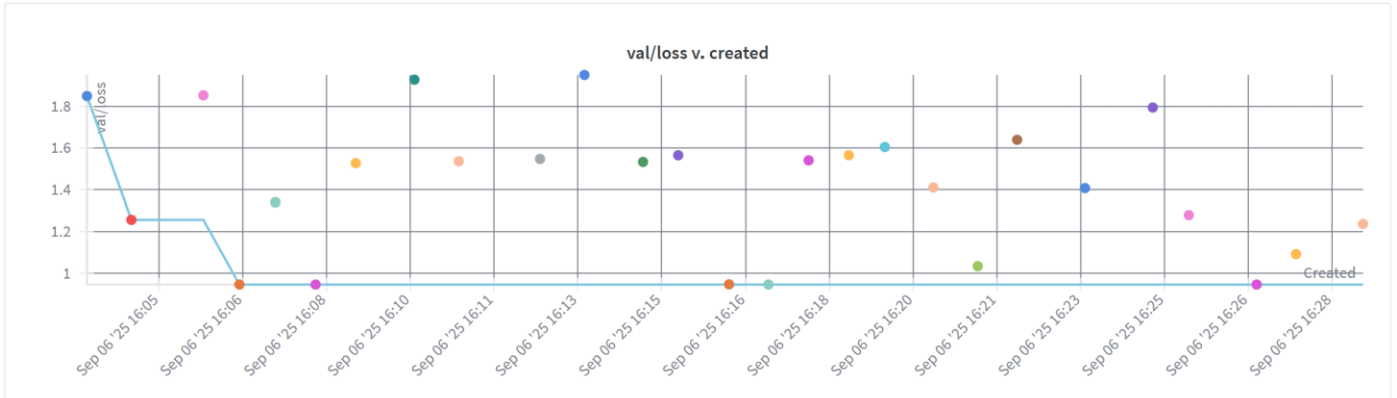
Appendix N

Configuration :https://github.com/tedasdf/mainrun/blob/main/mainrun/config/sweep_config/sweep_gpt_sparse.yamll



Appendix O

Configuration:https://github.com/tedasdf/mainrun/blob/main/mainrun/config/sweep_config/sweep_gpt_new.yamll



Appendix P

Configuration:

<https://github.com/tedasdf/mainrun/blob/main/mainrun/config/training.yamll>

