Core Task Progress Notes

Team: _____

November 14, 2025

Sanity-Check Run

- **Command.** `OMP_NUM_THREADS=1 python3 pico-llm.py --tinystories_weight 0.0 --input_files 3seqs.txt --block_size 32 --max_steps_per_epoch 1 --kgram_k 2 --embed_size 64 --device_id cpu`
- **Environment tweaks.** Installed the missing `tiktoken` package and pinned OpenMP threads to avoid shared-memory warnings in the sandboxed environment.
- **Outcome.** Training loop completed three abbreviated epochs for the LSTM baseline on `3seqs.txt`, producing loss values ($10.82 \rightarrow 10.80 \rightarrow 10.77$) and sample generations at greedy, $p = 0.95$, and $p = 1.0$ settings. This confirms data loading, tokenization, model forward/backward, and text generation pathways function end-to-end on CPU-only hardware.

Key Observations

- The current nucleus sampling placeholder yields identical outputs across different p values, reinforcing the need to implement true top-p sampling before the presentation.
- LSTM outputs trained on the minimal dataset remain nonsensical, which is expected given the toy corpus and tiny training budget; future sanity checks on TinyStories or richer data should improve qualitative quality.
- Installing dependencies on the target machine (e.g., `tiktoken`) should be part of setup instructions to prevent runtime failures during the live demo.

Sample Interview Questions & Answers

1. **What did you verify in the sanity-check run, and why did you choose the custom dataset?**
   We confirmed that the default training loop, tokenizer, and generation routines run without crashing on CPU. The `3seqs.txt` data ships with the repo, so it avoids the network dependency of downloading TinyStories and shortens iteration time.
2. **Why did the generated samples look incoherent, and is that a concern?**
   The dataset contains synthetic numeric sequences and we limited training to one gradient step per epoch. With minimal data and budget, the LSTM cannot learn meaningful structure; the purpose of this run was functionality, not quality. Larger datasets and more steps will address coherence.

3. **You pinned `OMP_NUM_THREADS` to 1—will that hurt performance later?**
   For this quick CPU smoke test it eliminated shared-memory errors. On a full training environment we can remove or raise the cap once OpenMP shared-memory permissions are available, restoring multithreaded BLAS performance.

4. **What additional checks will you run before the presentation?**
   We plan to repeat the sanity check on TinyStories once Transformer and k-gram models are implemented, capture training curves for the required figures, and verify qualitative outputs under true top-p sampling to demonstrate improved diversity.

KGramMLPSeqModel Design Options

1. **Baseline One-Hot MLP.**
   Structure: Flatten the k one-hot context vectors into a $(k \times |V|)$ input and feed through stacked Linear $\rightarrow$ SiLU blocks ending in a Linear layer to logits.
   Pros: Direct translation of the docstring; simplest to reason about; no extra embeddings needed.
   Cons: Extremely high input dimensionality (tens of thousands) makes each forward/backward pass slow and memory hungry; scales poorly with larger vocab or k.
   Performance expectations: Works for tiny vocab toy data but may be impractical for full TinyStories unless chunking is aggressive.

2. **Token Embedding + MLP (Shared Projection).**
   Structure: Map each context token through a learned embedding matrix ($|V| \rightarrow d$), concatenate the k embeddings (size $k \times d$), then apply a compact MLP that projects back to vocabulary logits.
   Pros: Input dimensionality drops dramatically; parameters scale with d instead of $|V|$; embeddings can be reused or initialized from LSTM/Transformer values.
   Cons: Slightly more complex; must ensure embeddings are trained jointly; loses the "pure" one-hot formulation.
   Performance expectations: Much faster and fits GPU memory; still expressive enough for k-gram patterns when d is moderate.

3. **Embedding + Depthwise 1D Convolution Hybrid.**
   Structure: Embed tokens as above, stack along temporal dimension, then use 1D convolution(s) over the k context positions followed by a projection to logits.
   Pros: Weight sharing across positions reduces parameters; convolutional kernels capture local order efficiently; fits well with chunked inference.
   Cons: Deviates from strict MLP requirement; requires reshaping logic in the provided forward loop; debugging convolutional behavior may be harder.
   Performance expectations: Efficient on GPU and handles longer k gracefully; offers inductive bias similar to CNNs for n-gram features.

Action Plan: Implementing KGramMLPSeqModel

1. **Select architecture.** Choose between one-hot MLP, embedding-based MLP, or convolution hybrid based on desired trade-offs.

2. **Define network layers.** Instantiate embedding (if needed), hidden Linear layers with activation, dropout if desired, and final Linear layer to $|V|$ logits.

3. **Integrate into forward loop.** Replace the `self.net = None` stub and ensure tensors are moved to the correct device, handling chunked processing without altering the provided outer loops.

4. **Add safety checks.** Validate input shapes, raise informative errors if `self.net` is misconfigured, and confirm gradients flow through embeddings.

5. **Test in notebook.** Run the reduced configuration from the sanity check, compare loss trajectories, and profile runtime to confirm expected improvements.

Forward Pass Rationale

- **Responsibility.** The `forward()` method ingests a tensor of token IDs with shape (sequence length, batch size), walks through the sequence in micro-batches, extracts the preceding k tokens for each position, pads when history is short, one-hot encodes that context, and ships the flattened vector through the chosen sub-network to obtain logits.

- **Chunking.** Iterating in blocks of size `chunk_size` lowers peak memory usage, making the naive Python loop feasible even for long sequences.

- **Why it matters.** Without this logic, the k-gram model would never transform context into next-token distributions—no logits means no loss, no gradients, and no generation. This function therefore defines the core behavior we train and evaluate.

- **Postconditions.** Concatenating the outputs from each chunk restores the expected (seq_len, batch, $|V|$) tensor that downstream code (loss, sampling) depends on.

Action Plan: Implementing Nucleus Sampling

1. **Normalize logits.** Convert logits to probabilities via `softmax` to obtain a stable distribution over the vocabulary.

2. **Rank tokens.** Sort probabilities in descending order while keeping token indices to identify the highest-mass prefix.

3. **Cumulative cutoff.** Compute the running probability mass and retain the smallest slice whose sum exceeds the user-specified p threshold.

4. **Renormalize.** Scale the retained probabilities so they sum to one, ensuring a valid categorical distribution.

5. **Sample token.** Draw the next token from the truncated distribution (fall back to the full distribution if p never triggers).

6. **Surface API.** Expose the sampler through `generate_text` so notebook experiments can vary p without extra wiring.

Nucleus Sampling Observations

- **Setup.** Used the quick LSTM sanity-check run (one optimization step on the synthetic corpus) and generated 20 new tokens from the prompt "Once upon a" with different top-p thresholds.
- **Greedy (p = None).** *"Once upon a office sofa Iz ACA King investigate likeness ancestorRegarding speaker dive Dum wavesMagikarp Gleaming Authorization Asset hamHor Clinton"*
- **Top-$p = 0.8$.** *"Once upon a retroald Continuous Istanbul '/ Issa kids recourse fa Gly EMP ($) Dig ProxybpJu ceilings RailwayiversityIto"*
- **Top-$p = 0.95$.** *"Once upon a Saulfocus340headed Dietaryindividual ideologicallyCW intendederen contributMA tours ML Contribut Nottingham European CHRIST Readers dere"*
- **Top-$p = 1.0$.** *"Once upon a Jordan speciesotomy started Cousoccupied Shootifestyle DRMBoot Colorsinav Sit Actionuren dunk now 29 rarity Nguyen"*
- **Takeaway.** Higher p values admit a larger tail of low-probability tokens, yielding more diverse (but noisier) completions. Greedy decoding sticks to the modal choice and repeats motifs, while $p = 1.0$ allows maximum randomness.

Optional Task: Custom Training Data

- **Dataset.** Authored five corpora of hand-written micro-stories in `data/custom_corpus/field_notes.txt`, `mini_stories.txt`, `observatory_journal.txt`, `city_archive.txt`, and the new `harbor_diary.txt`. Combined, the directory now holds 182 unique lines, tripling the original custom pool with metro, nocturnal, aerial, and harbor vignettes.
- **Loader Updates.** Added `--input_dir` (recursive `*.txt` discovery) and `--limit_custom_examples` flags so we can point the trainer at an entire folder or trim the number of custom lines for quick tests.
- **Usage.** Example command: `OMP_NUM_THREADS=1 python3 pico-llm.py --tinystories_weight 0.25 --input_dir data/custom_corpus --block_size 64 --max_steps_per_epoch 1 --enable_transforme gptoss`.
- **Result Snapshot.** A quick smoke test with `--tinystories_weight 0.25` + the custom corpus loads 182 additional sequences and trains without errors. Even with the one-step-per-epoch cap the models start echoing the new terms ("drone", "lighthouse", "meteorite") instead of the numeric toy lines, confirming that the loader now ingests richer data sources.
- **Loss Sweep.** Recorded short runs at four corpus sizes (3, 63, 123, 185 sequences) with `--custom_sweep_log artifacts/custom_data_sweep.json`. The notebook plots the final epoch loss for LSTM and GPT-oss versus corpus size, showing monotonic loss reduction for LSTM and a gentler decline for GPT-oss as more custom data is added.
- **Training Budget Sweep.** Logged fixed-corpus runs with increasing steps per epoch (1, 3, 6) under a 60/40 train/test split. Stored metrics in `artifacts/training_budget_sweep.json`; the notebook visualises how additional gradient steps lower both train and eval losses, clarifying that the earlier loss rise was due to under-training.

- **Next Steps.** Curate a larger custom directory (possibly user transcripts or domain notes) and run longer schedules while logging per-source sampling weights. Compose qualitative comparisons between TinyStories-only and blended corpora for the presentation.

Performance Comparison Strategy

- **Feasibility.** It is practical to compare all three architectures visually: log training loss and wall-clock time per step for each variant while running identical hyperparameters, then plot the metrics using `matplotlib` inside the standalone notebook. GPU execution is optional but accelerates experimentation.
- **Instrumentation plan.**
  - Wrap the training loop to record (loss, step, elapsed seconds) tuples for each KGramMLPSeqModel option via separate helper functions.
  - Store results in a shared list or `pandas` DataFrame, noting architecture label and configuration (e.g., embedding dimension).
  - After each run, generate line plots for loss vs. step and bar charts for tokens/sec to highlight efficiency differences.
- **Caveats.** Full TinyStories runs may be slow in Colab CPU mode; expect to subsample the dataset or limit training steps for a fair yet tractable comparison. Ensure random seeds are fixed for reproducibility.

Overfitting Exploration Plan

- **Goal.** Demonstrate and study overfitting by training on a deliberately small subset of the custom corpus while monitoring both training and testing loss.
- **Subset Selection.** Sample 20–30 sequences from the custom directory, maintain a 60/40 train/test split so the held-out portion remains informative.
- **Training Schedule.** Run extended schedules (e.g., up to 100 gradient steps) with periodic checkpoints. Capture training/testing loss pairs as well as qualitative generations to observe memorisation.
- **Architectures.** Repeat the procedure for both LSTM and GPT-oss to compare how quickly each model memorises the small dataset.
- **Artifacts.** Log metrics to a new JSON file (e.g., `artifacts/overfitting_sweep.json`) so the notebook can add a dedicated plot once runs are complete.
- **Initial Run.** Executed a 30-epoch sweep (batch size 4) on a 43-sequence subset; metrics recorded in `artifacts/overfitting_sweep.json` now feed the notebook plot highlighting the divergence between training and testing losses.

Optional Task: General Hyperparameter Study (Plan)

- **Objective.** Systematically explore non-learned settings (embedding size, block size, number of LSTM layers, Transformer heads/blocks, learning rate, batch size) to see how they affect convergence speed, loss, and sample quality.
- **Experimental Grid.**
  - **Scope.** Focus on a manageable subset: e.g., embedding sizes $\{256, 512, 1024\}$, block sizes $\{32, 64, 128\}$, LSTM layers $\{1, 2\}$, Transformer heads $\{4, 8\}$, learning rates $\{5 times 10^{-4}, 1 times 10^{-3}\}$.
  - **Models.** Run both LSTM and GPT-oss for each hyperparameter combo.
  - **Budget.** Fix a moderate training budget (e.g., 10 epochs, batch size 8) to keep runtimes tractable while still reflecting differences.
- **Metrics.** Log training/testing loss, tokens-per-second, and representative generations. Store results in `artifacts/hparam_sweep.json` for notebook visualisation (e.g., heatmaps or spider plots).
- **Notebook Integration.** Add a "General Hyperparameter Study" section that pulls the JSON results and highlights the best-performing configurations along with trade-offs (quality vs. speed).
- **Deliverables.** Produce plots comparing loss vs. embedding size, loss vs. learning rate, etc., summarise key findings, and note recommendations for the final presentation.
- **Initial Runs.** Logged three configurations (`emb256/block32/lr5e-4`, `emb512/block64/lr1e-3`, `emb1024/block128/lr2e-3`) to `artifacts/hparam_sweep.json`; the notebook now visualises their training/testing losses side by side.

Empirical Results (Synthetic Corpus)

- **Setup.** CPU only; k=2, embedding size 64, two inner MLP layers, Adam with learning rate 1 $times 10^{-3}$, 20 mini-batches per variant (batch size 32) over the `3seqs.txt` dataset.
- **Metrics.**
  - One-hot MLP: average loss $approx 10.63$, throughput $approx 17.6$ tokens/s, runtime $approx 1161$ seconds.
  - Embedding MLP: average loss $approx 9.49$, throughput $approx 42.3$ tokens/s, runtime $approx 485$ seconds.
  - Conv Hybrid: average loss $approx 10.23$, throughput $approx 32.3$ tokens/s, runtime $approx 634$ seconds.
- **Decision.** The embedding-based MLP achieves the best trade-off (lowest loss and highest throughput), so we will proceed with this architecture for the full implementation and upcoming experiments.

TransformerModel Implementation Outline

1. **Surface Configuration.**
   - Expose `d_model`, `n_heads`, `n_blocks` (capped at ten), feed-forward width multiplier, dropout rates, and rotary-frequency toggles so we can sweep sizes without rewriting layers.
   - Share a `transformer_variant` flag to map onto the presets listed below (GPT2, GPT-oss, Llama3-lite, minGPT).
2. **Token and Positional Embeddings.**
   - Reuse the shared token embedding matrix for the output projection (weight tying).
   - Implement learned positional embeddings by default; include a rotary positional embedding helper so GPT-oss and Llama-style presets can opt in.
3. **RMSNorm and Block Skeleton.**
   - Finish the RMSNorm stub: compute the mean square per hidden vector, scale by `weight`, and broadcast the epsilon guard.
   - Each Transformer block follows Pre-Norm layout: `RMSNorm` $\rightarrow$ Multi-Head Attention + residual $\rightarrow$ `RMSNorm` $\rightarrow$ Feed-Forward + residual.
4. **Multi-Head Attention.**
   - Compute query, key, value tensors via shared Linear projections; reshape to `(seq, batch, heads, head_dim)`.
   - Apply rotary or additive position bias if enabled, mask with causal upper-triangular mask, scale by $\frac{1}{\sqrt{\text{head\_dim}}}$, and reuse the cache-friendly loop already provided in the stub comments.
5. **Feed-Forward Network.**
   - Default to GELU two-layer MLP with width multiplier $4\times$; allow toggling to SwiGLU for the Llama3-lite preset.
   - Insert dropout hooks that become no-ops during evaluation to match the starter code style.
6. **Forward Pass and Output Head.**
   - Accept input as `(seq, batch)`, emit logits `(seq, batch, vocab)`.
   - Tie the embedding weights for the output linear layer unless a variant explicitly disables sharing.
7. **Initialization and Guard Rails.**
   - Adopt variant-specific initializers (GPT2 uses scaled normal, Llama-style uses Xavier uniform).
   - Assert `n_heads` divides `d_model` and `n_blocks` ≤ 10 to keep runs within the assignment budget.
8. **Testing Hooks.**
   - Add unit checks that random inputs produce finite logits and that the KV-cache path matches the dense attention for small sequences.
   - Benchmark the synthetic dataset with three quick batches to record loss and throughput per variant.

Transformer Variants and Feasibility

- **minGPT-inspired (Feasible).**
  - Architecture: `d_model` 384, `n_heads` 6, `n_blocks` 6, GELU feed-forward width 1536, learned positional embeddings, tied output weights.
  - Implementation status: fully achievable with the outline above; requires no external dependencies.
  - Metrics plan: reuse the synthetic benchmark harness (20 batches, batch size 32) and report loss/tokens-per-second next to the KGram results.
- **GPT2-small style (Feasible with scaling).**
  - Architecture: adapt GPT-2 small to assignment limits by using `d_model` 768, `n_heads` 12, `n_blocks` 8 (reduced from 12 to stay ≤ 10), learned absolute positions, MLP width 3072, GELU activation, tied output head.
  - Implementation status: feasible once RMSNorm is replaced with LayerNorm for this preset or we supply a LayerNorm toggle; compute cost is higher but manageable on the synthetic corpus.
  - Metrics plan: same synthetic harness; for TinyStories we will downscale batch size to keep runtimes under the cap.
- **GPT-oss (Feasible, requires rotary).**
  - Architecture: `d_model` 512, `n_heads` 8, `n_blocks` 8, rotary positional embeddings, SwiGLU feed-forward width 2048, untied output head per reference design.
  - Implementation status: feasible after we add a light-weight rotary embedding helper; no external kernels needed.
  - Metrics plan: record synthetic benchmark figures and optionally a short TinyStories run (max steps per epoch = 1) to stay within time limits.
- **Llama3-lite (Partially feasible).**
  - Architecture: distilled from the published spec with `d_model` 1024, `n_heads` 16, `n_blocks` 8, rotary positions, SwiGLU feed-forward width 8192, RMSNorm everywhere, untied output head.
  - Implementation status: core building blocks (rotary, SwiGLU, RMSNorm, attention) fit our codebase; full Llama3 features (grouped-query attention, sliding window) exceed current scope. We propose implementing a simplified variant without the extra attention optimisations.
  - Metrics plan: we can capture synthetic benchmark numbers; collecting full TinyStories metrics would demand GPU time beyond the assignment budget, so we will document only CPU synthetic runs unless instructed otherwise.
- **Performance Collection Summary.**
  - All small-batch synthetic metrics are attainable once the Transformer is implemented.
  - Full-scale TinyStories training for GPT2-small or Llama3-lite is out of scope without extensive runtime; we will note this limitation in results to keep expectations clear.

- **Recorded Metrics (Synthetic Corpus, 3 batches, batch size 16).**
  - minGPT-style: average loss 234.14, throughput 26.33 tokens/s, elapsed 0.57 s.
  - GPT2-small (trimmed to 8 blocks): average loss 360.30, throughput 9.35 tokens/s, elapsed 1.60 s.
  - GPT-oss (default): average loss 10.93, throughput 12.36 tokens/s, elapsed 1.21 s.
  - Benchmarks are gated behind `--collect_transformer_metrics`; the numbers above match the `RECORDED_TRANSFORMER_BENCHMARK` entries in `pico-llm.py`.

Core Task Interview Questions & Answers

Task 1: Sanity-Check Run

1. **What was the purpose of the sanity-check run, and what did it verify?**
   The sanity-check run validated that the entire training pipeline works end-to-end: data loading, tokenization, model forward/backward passes, loss computation, and text generation. We used the minimal `3seqs.txt` dataset to avoid network dependencies and speed up iteration. The run confirmed that all core components function correctly on CPU-only hardware before implementing more complex models.

2. **Why did you choose `3seqs.txt` instead of TinyStories for the initial check?**
   `3seqs.txt` ships with the repository, eliminating network download time and potential connectivity issues. It's also tiny (3 sequences), making it ideal for quick smoke tests. TinyStories requires downloading from Hugging Face datasets, which adds latency and potential failure points during a live demo. Once the pipeline is verified, we can confidently scale to larger datasets.

3. **The generated text looks nonsensical. Is this expected, and should we be concerned?**
   Yes, this is expected. The dataset contains only 3 synthetic sequences, and we limited training to one gradient step per epoch. With such minimal data and training budget, the model cannot learn meaningful linguistic patterns. The purpose was to verify functionality, not quality. When we scale to TinyStories or larger custom corpora with proper training budgets, the outputs will improve significantly.

4. **Why did you set `OMP_NUM_THREADS=1`, and will this impact performance?**
   This eliminates shared-memory warnings in sandboxed environments where OpenMP lacks proper permissions. For quick CPU smoke tests, single-threaded execution is sufficient. In production training environments with proper OpenMP setup, we can remove this cap to restore multithreaded BLAS performance, which significantly accelerates matrix operations.

Task 2: K-gram MLP Implementation

1. **What are the three K-gram architecture variants you considered, and why did you choose the embedding-based MLP?**
   We evaluated three options: (1) One-hot MLP that flattens k one-hot vectors into a $(k \times |V|)$ input, (2) Embedding-based MLP that maps tokens to learned embeddings then processes through a compact MLP, and (3) Convolution hybrid using 1D convolutions over embedded tokens. We chose the embedding-based MLP because it achieved the best trade-off: lowest loss (9.49 vs. 10.23 for conv, 10.63 for one-hot) and highest throughput (42.3 tokens/s vs. 32.3 for

conv, 17.6 for one-hot) on the synthetic corpus. The one-hot variant suffers from extremely high input dimensionality, while the embedding approach dramatically reduces parameters and memory usage.

2. **How does the K-gram model differ from the LSTM in terms of context handling?**
   The LSTM processes sequences sequentially, maintaining a hidden state that theoretically captures information from all previous tokens. The K-gram MLP uses a fixed sliding window of k preceding tokens as context, with no recurrent state. This makes it more memory-efficient and parallelizable, but limits its ability to capture long-range dependencies beyond the window size k. The K-gram model is essentially a local pattern matcher, while the LSTM can theoretically remember information from much earlier in the sequence.

3. **What is the purpose of chunking in the K-gram forward pass, and how does it work?**
   Chunking processes the sequence in micro-batches of size `chunk_size` to reduce peak memory usage. Instead of processing all positions at once (which could require storing $(\text{seq\_len} \times k \times |V|)$ one-hot vectors), we iterate through blocks, extract the k-token context for each position in the chunk, and process them incrementally. This makes the naive Python loop feasible even for long sequences without exhausting memory, especially important for the one-hot variant with its massive input dimensionality.

4. **How do you handle positions where there are fewer than k preceding tokens?**
   We pad the context with a special padding token (typically token ID 0) when the sequence position is less than k. For example, at position 0, we pad with k padding tokens; at position 1, we use 1 real token and $(k-1)$ padding tokens. This ensures the input to the MLP always has the same dimensionality, allowing batch processing while maintaining the sliding window semantics.

5. **What were the key performance differences between the variants in your benchmarks?**
   On the synthetic corpus (20 batches, batch size 32, CPU-only): The embedding MLP achieved 9.49 average loss and 42.3 tokens/s in 485 seconds. The conv hybrid reached 10.23 loss and 32.3 tokens/s in 634 seconds. The one-hot MLP had 10.63 loss and only 17.6 tokens/s in 1161 seconds. The embedding approach wins on both quality (lower loss) and efficiency (higher throughput, faster runtime), making it the clear choice for production use.

Task 3: Nucleus Sampling

1. **What is nucleus (top-p) sampling, and how does it differ from greedy decoding?**
   Nucleus sampling dynamically truncates the vocabulary to the smallest set of tokens whose cumulative probability mass exceeds a threshold p. We sort tokens by probability, compute cumulative sums, and keep only tokens up to the point where the sum first exceeds p. Then we renormalize and sample from this truncated distribution. Greedy decoding always selects the single most probable token (argmax), which is deterministic and can lead to repetitive outputs. Top-p introduces controlled randomness while filtering out low-probability noise.

2. **How do different p values affect the generated text quality and diversity?**
   Lower p values (e.g., 0.8) restrict sampling to high-probability tokens, producing more conservative but coherent outputs. Higher p values (e.g., 0.95) admit more candidates, increasing diversity but potentially introducing less likely tokens. At $p = 1.0$, we sample from the full distribution, which can produce very diverse but sometimes nonsensical text. Greedy ($p = \text{None}$)

always picks the mode, leading to repetitive patterns. In our experiments, moderate values like 0.8–0.95 provide a good balance between coherence and variety.

3. **What is the computational complexity of nucleus sampling compared to greedy decoding?**
   Greedy decoding is $O(|V|)$ per token: a single argmax over the vocabulary. Nucleus sampling requires sorting probabilities ($O(|V| \log |V|)$), computing cumulative sums ($O(|V|)$), finding the cutoff point ($O(|V|)$), and renormalizing ($O(k)$ where k is the nucleus size). In practice, the sorting dominates, but modern libraries optimize this, and the overhead is usually acceptable for the quality improvement. The complexity is still linear in vocabulary size, just with a logarithmic factor.

4. **Why might nucleus sampling produce better qualitative results than greedy decoding?**
   Greedy decoding can get stuck in repetitive loops because it always chooses the same high-probability continuation. Nucleus sampling introduces controlled randomness that breaks these loops while still avoiding low-probability tokens that would degrade quality. It also allows the model to explore multiple plausible continuations, making the text more natural and less mechanical. In our experiments, greedy outputs repeated motifs like "investigate likeness ancestorRegarding," while top-p sampling produced more varied and engaging text.

5. **How do you handle edge cases in nucleus sampling, such as when all probabilities are very low?**
   If the cumulative probability never reaches p (which shouldn't happen with proper normalization, but could occur with numerical issues), we fall back to sampling from the full distribution. We also ensure that at least one token is always included in the nucleus, even if its probability is below the threshold, to guarantee a valid sampling distribution. Additionally, we add small epsilon values during renormalization to prevent division by zero and handle floating-point precision issues.

Task 4: Transformer Implementation

1. **What Transformer variants did you implement, and why did you choose GPT-oss as the default?**
   We implemented three variants: minGPT-inspired (384 dim, 6 heads, 6 blocks), GPT2-small style (768 dim, 12 heads, 8 blocks), and GPT-oss (512 dim, 8 heads, 8 blocks with rotary embeddings and SwiGLU). We chose GPT-oss as the default because it achieved the best loss (10.93) on the synthetic corpus while maintaining reasonable throughput (12.36 tokens/s). It balances modern architectural choices (rotary positional embeddings, SwiGLU activation) with computational efficiency, making it ideal for quick iterations and experimentation.

2. **What is RMSNorm, and how does it differ from LayerNorm?**
   RMSNorm (Root Mean Square Normalization) normalizes by the root mean square of activations, omitting the mean-centering step of LayerNorm. Specifically, RMSNorm computes $\text{RMS}(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}$ and returns $x/(\text{RMS}(x) + \epsilon) \times \text{weight}$. LayerNorm subtracts the mean first: $(x - \mu)/(\sigma + \epsilon) \times \text{weight}$. RMSNorm is simpler and faster, and has been shown to work well in modern architectures like GPT-oss and LLaMA, though LayerNorm remains standard in many Transformer variants.

3. **What are rotary positional embeddings (RoPE), and why did you use them in GPT-oss?**
   Rotary Positional Embeddings apply a rotation matrix to query and key vectors based on their position in the sequence. The rotation encodes relative position information directly into the attention mechanism, allowing the model to understand token relationships without explicit positional encodings. RoPE has several advantages: it naturally handles variable sequence lengths, provides better extrapolation to longer sequences than learned embeddings, and has been shown to improve performance in models like GPT-oss and LLaMA. We used it in GPT-oss to match the reference architecture and leverage these benefits.

4. **What is SwiGLU activation, and how does it differ from standard GELU?**
   SwiGLU (Swish-Gated Linear Unit) is a gated activation function: $\text{SwiGLU}(x) = \text{Swish}(xW + b) \odot (xV + c)$, where $\odot$ is element-wise multiplication. It combines a Swish-activated branch with a gating branch, effectively allowing the network to learn which parts of the input to pass through. Standard GELU is simply $\text{GELU}(x) = x \cdot \Phi(x)$ where Φ is the CDF of a standard normal. SwiGLU has more parameters (two linear projections instead of one) but often achieves better performance, which is why it's used in modern architectures like GPT-oss and LLaMA.

5. **How does multi-head attention work, and why is it important?**
   Multi-head attention splits the model dimension into h heads, each with dimension d_{model}/h. Each head independently computes attention over a subspace, allowing the model to attend to different types of relationships simultaneously (e.g., syntactic, semantic, positional). The heads are concatenated and projected back to d_{model}. This parallel attention mechanism is more expressive than single-head attention and is a key innovation of the Transformer architecture. In our implementation, we use 8 heads for GPT-oss, each with 64 dimensions (512 / 8).

6. **What is causal masking, and why is it necessary in language modeling?**
   Causal masking ensures that each token can only attend to previous tokens (and itself), preventing the model from "peeking" at future tokens during training. This is implemented as an upper-triangular mask filled with $-\infty$ (which becomes 0 after softmax) applied to the attention scores. Without causal masking, the model could cheat by using future information to predict the current token, which would not work during autoregressive generation. Causal masking enforces the autoregressive property: predictions depend only on past context.

7. **Why did you limit the number of Transformer blocks to 10?**
   This was a constraint from the assignment specification to keep model complexity and training time manageable. With 8 blocks in GPT-oss (our default), we stay well under this limit while still achieving good performance. More blocks generally improve capacity but increase computation quadratically with sequence length (due to attention) and linearly with depth. The 10-block cap ensures experiments remain tractable on CPU and within reasonable time budgets for the course.

8. **What were the performance differences between the Transformer variants in your benchmarks?**
   On the synthetic corpus (3 batches, batch size 16, CPU-only): GPT-oss achieved 10.93 average loss and 12.36 tokens/s. minGPT had 234.14 loss and 26.33 tokens/s (the high loss suggests initialization or architecture issues). GPT2-small reached 360.30 loss and 9.35 tokens/s (also high loss, possibly due to reduced block count from 12 to 8). GPT-oss clearly outperformed the others in terms of loss, making it the best choice for our default configuration.

Task 5: Optional Tasks (Custom Data, Overfitting, Hyperparameters)

1. **How did you implement custom training data loading, and what features did you add?**
   We added two CLI flags: `--input_dir` for recursively loading all `*.txt` files from a directory, and `--limit_custom_examples` to cap the number of custom sequences for quick tests. The loader deduplicates file paths, tokenizes each line up to `block_size`, and mixes custom data with TinyStories based on the `--tinystories_weight` probability. We also implemented train/test splitting with `--test_split_ratio` to support evaluation on held-out data. This allows flexible experimentation with domain-specific corpora while maintaining the ability to blend with TinyStories.

2. **What did your overfitting experiment demonstrate, and how did LSTM and GPT-oss compare?**
   We trained on a small 43-sequence subset (60% train, 40% test) for 30 epochs with batch size 4. Both models showed classic overfitting: training loss continued decreasing (LSTM: 0.21, GPT-oss: 0.22) while testing loss plateaued and began rising (LSTM: 8.84, GPT-oss: 8.56). GPT-oss memorized faster (its training loss dropped more quickly), but both models ultimately saturated on the tiny dataset. This demonstrates that with insufficient data, even powerful architectures will overfit, highlighting the importance of dataset size and regularization.

3. **What hyperparameters did you explore, and what were the key findings?**
   We varied embedding size (256–1024), block size (32–128), learning rate (5e-4–2e-3), and batch size (4–12) across 6 configurations. The results showed that GPT-oss consistently achieved lower training losses than LSTM across all configurations, but testing losses were more similar (around 5.8–7.7), suggesting GPT-oss may be more prone to overfitting. Larger embedding sizes (1024) and block sizes (128) didn't always improve performance, indicating that model capacity must be balanced with training budget and data size.

4. **Why did loss increase when you added more training data in your initial sweep?**
   This was a case of underfitting, not a problem with the data. We kept the training budget fixed (3 epochs) while increasing corpus size. With more data but the same number of gradient steps, the model couldn't learn the larger dataset effectively. When we increased the training budget (more steps per epoch) in a follow-up experiment, losses decreased as expected. This highlights the importance of scaling training budget with dataset size: more data requires more training steps to learn effectively.

5. **How did you structure your experimental logging and visualization?**
   We implemented JSON-based logging via `--custom_sweep_log` that records hyperparameters, model metrics (training/testing losses, steps, elapsed time), and dataset statistics. The notebook reads these JSON files and generates plots without re-running training, keeping the notebook computation-free. This separation of concerns allows us to run expensive training offline and visualize results interactively, making it easy to compare configurations and share results with instructors.