Core Task Progress Notes

Team: ________________________

November 12, 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 → 10.80 → 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.

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.

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.