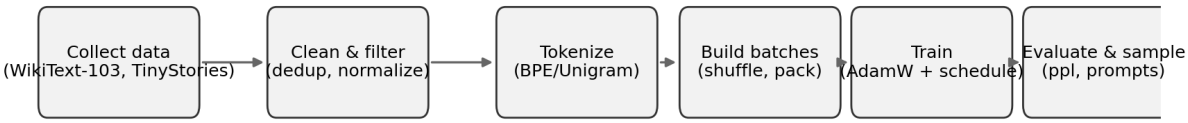
Training a Small LLM From Scratch

A practical, scientific guide to build and scale a decoder-only Transformer model from ~20M parameters to 50M and 100M using WikiText-103 and TinyStories.

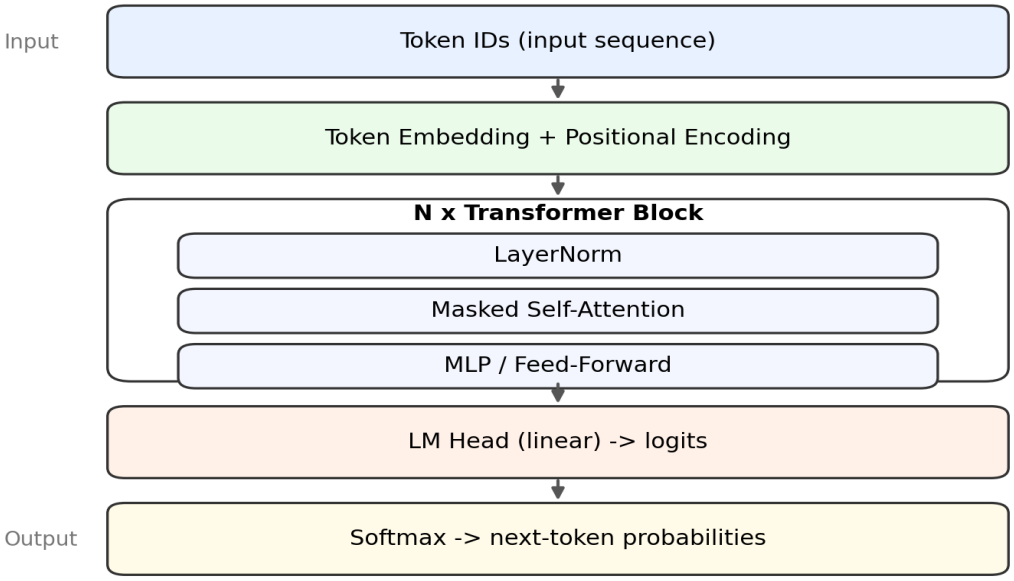
Prepared for Mohamed • February 18, 2026

End-to-end training pipeline



Repeat the loop: train -> evaluate -> adjust data/params -> train more

Decoder-only Transformer (GPT-style)



Contents

1. What you are building (mental model)

You are training an **autoregressive language model**: given a sequence of tokens (subwords/bytes/characters), the model learns to predict the **next token** at every position. The training signal is simple and extremely scalable: minimize next-token prediction error over large text. A practical goal for small models (20M-100M) is to learn strong local grammar, frequent facts, and coherent short-form generation. Quality depends on **data**, **tokenization**, **architecture**, and **training stability**.

> ***Scientific mindset:*** *treat each change as an experiment. Track what changed (data, tokenizer, model size, optimizer, schedule), what metric improved (loss/perplexity), and whether samples actually look better.*

2. Data: collecting, cleaning, and mixing corpora

You mentioned using **WikiText-103** (encyclopedic text) and **TinyStories** (simple narratives). This is a strong combination: WikiText teaches factual style and long sentences, while TinyStories teaches coherent short story structure and dialogue. Most small-model training benefits from a **mixture**.

2.1 Data collection checklist

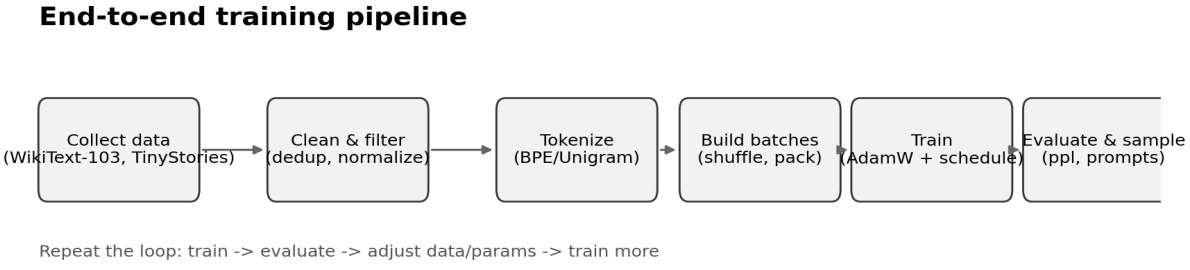
• Use official dataset dumps (or well-known dataset libraries) and keep a record of versions.
• Store raw text separately from processed text to allow re-processing later.
• Split into train/validation early (do not leak validation text into training).
• Keep metadata: source name, language, license, date, document id (if available).

2.2 Cleaning and quality filters (minimum set)

• Normalize whitespace, remove broken encodings, and standardize newlines.
• Remove duplicate documents (exact and near-duplicate) to prevent memorization and repetition.
• Filter extreme length outliers (very short spam or extremely long noisy pages).
• Optional: keep only English if your goal is English generation.

2.3 Mixing datasets (a simple scientific approach)

Start with a baseline mix (example: 70% TinyStories, 30% WikiText). Train a small run, evaluate loss and samples, then adjust the ratio. If the model becomes too 'story-like' and loses factual style, increase WikiText; if it becomes dry and incoherent in narrative, increase TinyStories.



3. Tokenization and vocabulary design

Tokenization converts text into a sequence of discrete symbols. Most modern LLMs use **subword tokenization** (BPE or Unigram) on bytes or Unicode text. Your run shows **Vocab: 4,979** tokens, which is a relatively small subword vocabulary (good for compact models, but it can increase sequence length).

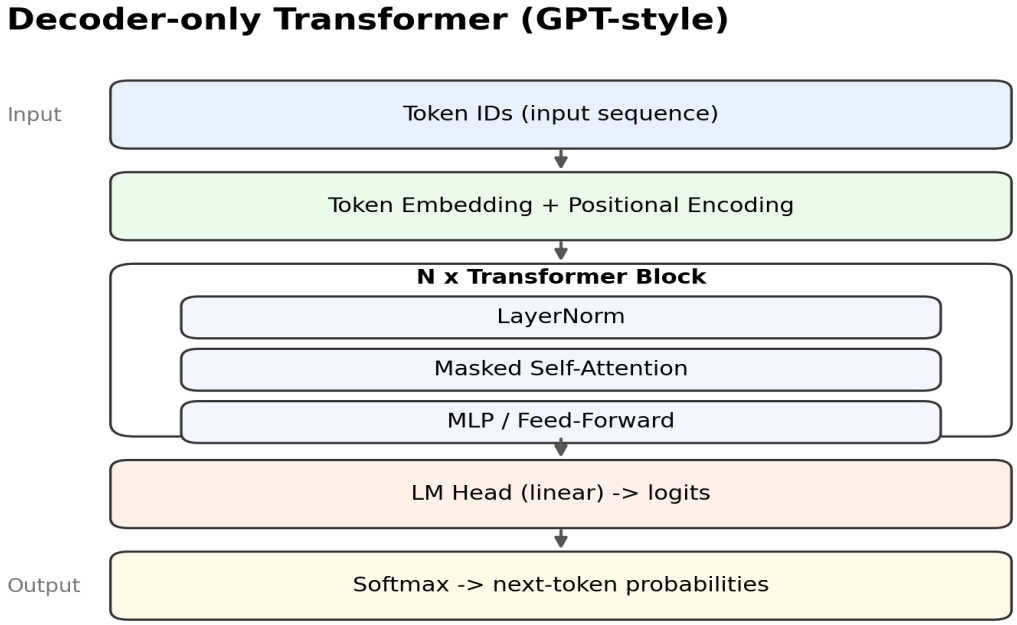
Good tokenization principles

• Train the tokenizer on the same mixture of data you will train the model on.
• Include a clear set of special tokens: *pad*, *eos*, *bos* (optional), *unk* (if needed).
• Check tokenization of common words, punctuation, and numbers (avoid pathological splits).
• For English-only small models, vocab sizes in the 8k-32k range often work well; smaller vocabs can still work but may need longer context.

> *If your generations show lots of repetition, one cause can be **sampling** (temperature/top-k/top-p), but another is insufficient diversity in the dataset or too-small vocab causing awkward token patterns. Treat it like an experiment: increase vocab (e.g., 8k), keep everything else fixed, and compare.*

4. Model architecture (decoder-only Transformer)

For next-token prediction, the standard architecture is a **decoder-only Transformer** (GPT style). It uses **masked self-attention** so each position can attend only to earlier tokens. The model outputs logits over the vocabulary for every time step.



Key design knobs

• **d_model** (embedding width): main capacity knob.
• **n_layers**: depth; more layers usually improves reasoning and long-range structure.
• **n_heads**: attention heads; choose so that d_model is divisible by n_heads.
• **d_ff** (MLP hidden): typically ~4 * d_model.
• **seq_len**: maximum context window; larger needs more compute.

5. The math: cross-entropy, softmax, and attention

5.1 Objective (next-token cross-entropy)

Given tokens x1...xT, the model predicts p(xt | x<t). The training loss is the average negative log-likelihood:

```
L(θ) = -(1/N) * Σ_t log pθ(x_t | x_<t)
```

If the loss uses natural logarithms (common in deep learning), **perplexity** is:

```
Perplexity = exp(L)
```

5.2 Softmax

The model outputs logits z for each token in the vocabulary. Softmax converts logits to probabilities:

```
p_i = exp(z_i) / Σ_j exp(z_j)
```

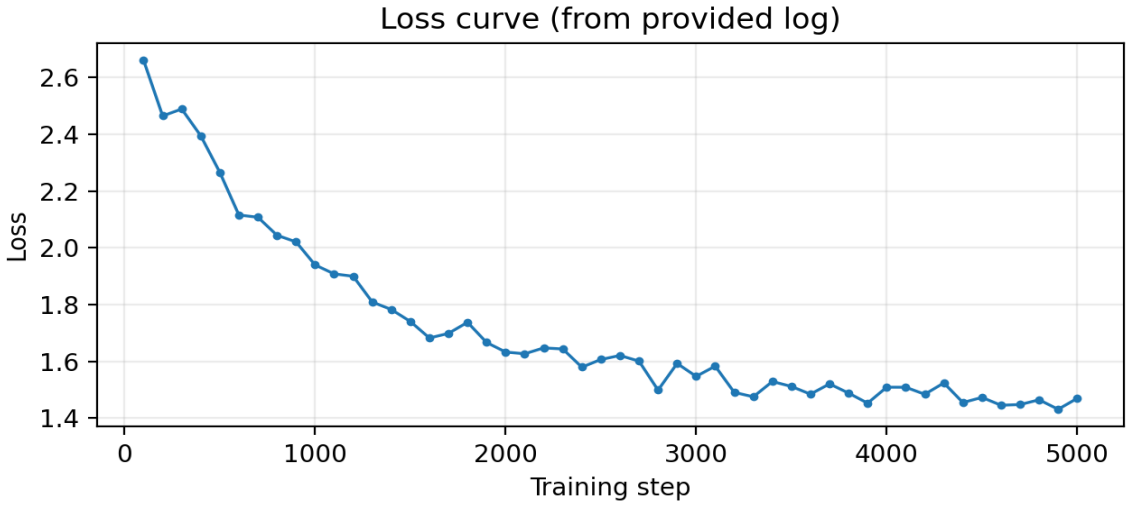
5.3 Scaled dot-product attention (with causal mask)

Attention maps a sequence into a weighted combination of past values. With queries Q, keys K, values V:

```
Attention(Q,K,V) = softmax((Q K^T) / sqrt(d_k) + mask) V
```

The **mask** blocks attention to future positions (causality). This is what makes the model autoregressive.

Your loss curve example (from the log you provided)



In your run, loss decreased from about **2.66** (ppl≈14.3) to about **1.47** (ppl≈4.3) over 5,000 steps. This is a healthy learning signal for an early-stage model.

6. Training engineering: optimizer, schedule, stability

Most small LLM training uses **AdamW** (Adam + decoupled weight decay). Stability and speed depend on the learning-rate schedule, batch size (tokens), precision (fp16/bf16), and gradient clipping.

6.1 Recommended defaults (strong baseline)

Component	Baseline choice	Why it works
Optimizer	AdamW (β1=0.9, β2=0.95)	Stable for Transformers
LR schedule	Warmup (1-5%) then cosine decay	Avoids early divergence, smooth convergence
Weight decay	0.1 (exclude bias/LayerNorm)	Regularizes without hurting normalization
Gradient clipping	1.0	Prevents exploding updates
Precision	bf16/fp16 + GradScaler	Faster and fits larger models on GPU
Dropout	0.0-0.1	Helps generalization for smaller data

6.2 Batch size in tokens (more important than 'batch size')

For language models, track **tokens per update** = batch_size * seq_len. When scaling up (20M -> 50M -> 100M), you often increase tokens per update to improve training efficiency. If GPU memory is limited, use gradient accumulation.

6.3 Sampling settings (to reduce repetition)

- Temperature: 0.8-1.0 (lower = safer, higher = more diverse).
- Top-k: 40-100 (limits to the most likely tokens).
- Top-p (nucleus): 0.9-0.95 (keeps a probability mass).
- Repetition penalty / no-repeat n-gram (optional): reduces loops in small models.

7. Reading your training log (loss, samples, checkpoints)

Your log includes device info, vocabulary size, total characters, step-by-step loss, and periodic text samples. This is exactly the right pattern for iterative development.

Metric	Value (from your run)
Device	cuda
Vocabulary size	4,979 tokens
Total characters in dataset	539,459,362
Training steps shown	5,000
Loss (start -> end)	2.6603 -> 1.4700
Perplexity approx.	14.3 -> 4.3 (if loss is ln-based)
Checkpoint	simple_ckpt.pt (saved multiple times)

Why the samples start repetitive

Early in training, the model has not learned long-range structure, so it collapses to very frequent tokens and short loops (e.g., 'the the the'). As loss decreases, it learns more plausible word sequences. Repetition later can be caused by: (1) small model capacity, (2) limited data diversity, (3) too-greedy decoding (temperature too low, no top-k/top-p), or (4) training not long enough.

Checkpoint discipline

Save checkpoints by: (a) step number, (b) best validation loss, and (c) time. Keep a small JSON/YAML file storing the config (layers, d_model, heads, seq_len, vocab), optimizer settings, and the exact data mix. This makes experiments reproducible.

8. Scaling plan: 20M → 50M → 100M

Scaling up is not just 'more parameters'. It must be matched with enough data, compute, and stable optimization. A good scaling plan changes one main axis at a time, while keeping a known-good baseline.

8.1 Approximate parameter math (rule of thumb)

For a decoder-only Transformer, parameters are dominated by layer weights. A simple approximation is:

```
Params ≈ 12 * n_layers * d_model^2  +  vocab_size * d_model
```

This ignores small terms (LayerNorm, biases). Use it to choose a target size quickly.

8.2 Example configs (compatible with small vocab like ~5k)

Target	n_layers	d_model	n_heads	d_ff	seq_len	Notes
~20M	10	384	6	1536	256-512	Good for fast experiments
~50M	12	512	8	2048	512-1024	Better fluency; needs more compute
~100M	16	640	10	2560	1024+	Stronger coherence; watch memory

8.3 What to scale together

- Increase **training tokens** when you increase parameters (more data/steps).
- Tune learning rate: larger models often need slightly smaller peak LR.
- Increase tokens/update or use gradient accumulation to keep training efficient.
- Always keep a validation set and stop when validation loss stops improving.

9. Roadmap: phases to improve quality step by step

Phase 0 - Minimal baseline

- Train a 10M-20M model end-to-end with a small context (256) and a small tokenizer.
- Confirm the loss decreases smoothly and sampling works.
- Log everything (config + seed + git commit).

Phase 1 - Data quality upgrade

- Deduplicate documents and remove low-quality noise.
- Introduce a stable train/valid split and track validation loss.
- Experiment with dataset mix ratios (TinyStories vs WikiText).

Phase 2 - Tokenization upgrade

- Increase vocab (e.g., 8k) and compare: loss, speed, sample quality.
- Make sure numbers and punctuation tokenize reasonably.
- Keep special tokens consistent across runs.

Phase 3 - Scaling to 50M

- Move to ~50M params using a config like 12 layers, 512 width.
- Use warmup + cosine decay, AdamW, gradient clipping.
- Increase context length if memory allows (512+).

Phase 4 - Scaling to 100M

- Increase to ~100M params, keep training stable with a slightly lower peak LR.
- Use bf16/fp16 + gradient accumulation to fit on GPU.
- Train longer and evaluate on prompt sets; reduce repetition with better sampling.

Phase 5 - Evaluation and specialization

- Build a small benchmark set: prompts for grammar, facts, story coherence.
- Add domain data (your target use case) for fine-tuning.
- Track regressions: do not lose general fluency when specializing.

Appendix A: Example hyperparameter presets

These are safe starting points. You will still need to tune for your GPU memory and dataset size.

Preset	Peak LR	Warmup	Weight decay	Dropout	Clip	Precision
20M fast	3e-4	200 steps	0.1	0.1	1.0	fp16/bf16
50M stable	2e-4	500 steps	0.1	0.05-0.1	1.0	bf16 preferred
100M stable	1.5e-4	1,000 steps	0.1	0.0-0.1	1.0	bf16 preferred

Tip: *If you see training loss improving but samples remain repetitive, try (1) better decoding (temperature + top-k/top-p), (2) a bit more dropout, (3) more diverse data, and (4) longer training.*

Appendix B: Common failure modes and fixes

Loss becomes NaN / diverges

• Lower peak learning rate.
• Add/confirm warmup, and enable gradient clipping.
• Check mixed precision (GradScaler) and overflow handling.

Model outputs repetitive loops

• Use better decoding: temperature 0.8-1.0, top-k 40-100, top-p 0.9-0.95.
• Increase data diversity and remove duplicates.
• Train longer; small models need more steps to learn structure.

Validation loss stops improving early

• Your dataset may be too small or too noisy; improve cleaning.
• Lower dropout if underfitting, increase if overfitting.
• Increase context length gradually; long contexts are harder.

Too slow / GPU OOM

• Use gradient accumulation instead of huge batch.
• Reduce seq_len, then increase gradually.
• Enable activation checkpointing if available.

Short excerpt of your provided log

```
Device: cuda
Vocab: 4979  Total chars: 539459362
step 100/5000 loss=2.6603
...
step 5000/5000 loss=1.4700
Saved checkpoint: simple_ckpt.pt
```