ECE405 Assignment 1

Github: https://github.com/bushuyeu/LLM-from-scratch

Code locations:

- `ece496b_basics/train_bpe.py` — BPE tokenizer training
- `ece496b_basics/tokenizer.py` — Tokenizer encode/decode
- `ece496b_basics/nn.py` — All neural network modules + generate()
- `ece496b_basics/optimizer.py` — AdamW optimizer, cosine LR schedule, gradient clipping
- `ece496b_basics/data.py` — get_batch()
- `ece496b_basics/checkpointing.py` — save/load checkpoint
- `train.py` — Training script
- `tests/adapters.py` — Test adapter glue code

Section 2: Byte-Pair Encoding (BPE) Tokenizer

Problem (unicode1): Understanding Unicode (1 pt)

(a) What Unicode character does `chr(0)` return?

`chr(0)` returns the **null character** (U+0000), also called NUL — the character with code point zero.

(b) How does this character's string representation (`__repr__()`) differ from its printed representation?

`repr(chr(0))` shows the escape sequence `'\x00'`; `print(chr(0))` produces no visible output.

(c) What happens when this character occurs in text?

The null character is invisible when printed but is still present in the string; it can cause strings to appear truncated in C-style languages or terminals that treat NUL as a string terminator.

Problem (unicode2): Unicode Encodings (3 pts)

(a) Why prefer training on UTF-8 encoded bytes rather than UTF-16 or UTF-32?

UTF-8 gives a small, fixed initial vocabulary of 256 byte values. UTF-16 would start with ~65,536 entries and UTF-32 would need over a million, making BPE training impractical. UTF-8 is also the most compact encoding for English text (1 byte per ASCII character vs 2+ for alternatives).

(b) Why is the following function incorrect?

```
def decode_utf8_bytes_to_str_wrong(bytestring: bytes):
    return "".join([bytes([b]).decode("utf-8") for b in bytestring])
```

This function decodes each byte independently, but multi-byte UTF-8 characters (e.g., non-ASCII characters like "こ" = 3 bytes) require multiple bytes decoded together. For example, `"こんにちは".encode("utf-8")` produces multi-byte sequences; decoding each byte alone raises `UnicodeDecodeError` because individual bytes of a multi-byte sequence are not valid UTF-8 on their own.

(c) Give a two-byte sequence that does not decode to any Unicode character(s).

`bytes([0xc0, 0x80])` — this is an "overlong encoding" of the null character, which is explicitly forbidden by the UTF-8 specification. Valid UTF-8 requires the shortest possible encoding for each code point.

Problem (train_bpe): BPE Tokenizer Training (15 pts)

Deliverable: Code implementation.

File: `ece496b_basics/train_bpe.py`

The `train_bpe()` function takes `input_path`, `vocab_size`, and `special_tokens` as parameters. It returns `(vocab, merges)` where vocab is `dict[int, bytes]` and merges is `list[tuple[bytes, bytes]]`. Implementation includes:

- Vocabulary initialization with 256 byte values + special tokens
- Parallel pre-tokenization using the GPT-2 regex pattern via the `regex` package
- Iterative merge loop finding the most frequent adjacent pair (ties broken lexicographically)
- Optimized pair counting with incremental updates

Problem (train_bpe_tinystories): BPE Training on TinyStories (2 pts)

(a) Training a 10K-vocab BPE tokenizer on TinyStories took ~17 minutes with a peak memory of 0.08 GB. The longest tokens in the vocabulary are 15 bytes each: `' accomplishment'`, `' disappointment'`, `' responsibility'` — frequent whole words (with leading space from GPT-2 pre-tokenization) from simple children's stories. This makes sense given the dataset's constrained vocabulary.

(b) Profiling shows the BPE merge loop itself is not the bottleneck. The `cProfile` trace reveals most wall-clock time is spent in `time.sleep`, `selectors.select`, and `SemLock.acquire` — the parallel pre-tokenization pipeline and multiprocessing coordination overheads.

Problem (train_bpe_expts_owt): BPE Training on OpenWebText (2 pts)

(a) Trained a 32K-vocab BPE tokenizer on OpenWebText. The longest token is `'ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ'` — an artifact from web-scraped data. BPE merges frequent repeated patterns, so encoding garbage becomes a single token. This makes sense as web crawls contain encoding artifacts.

(b) The TinyStories tokenizer learns clean English words (children's vocabulary), while the OWT tokenizer reflects its diverse source: URL fragments (`http`, `www`), political terms (`' unconstitutional'`), technical jargon (`' cryptocurrencies'`), and encoding artifacts.

Problem (tokenizer): Implementing the Tokenizer (15 pts)

Deliverable: Code implementation.

File: `ece496b_basics/tokenizer.py`

The `Tokenizer` class implements:

- `__init__(self, vocab, merges, special_tokens=None)` — constructs tokenizer from vocabulary and merges
- `from_files(cls, vocab_filepath, merges_filepath, special_tokens=None)` — class method to load from serialized files
- `encode(self, text: str) -> list[int]` — encodes text to token IDs (pre-tokenize, then apply merges in order)

- `encode_iterable(self, iterable) -> Iterator[int]` — memory-efficient streaming tokenization
- `decode(self, ids: list[int]) -> str` — decodes token IDs back to text (with `errors='replace'` for invalid UTF-8)

Problem (tokenizer_experiments): Experiments with Tokenizers (4 pts)

(a) Compression ratios (bytes/token):

- TinyStories tokenizer on TinyStories: **4.25 bytes/token**
- OpenWebText tokenizer on OpenWebText: **4.54 bytes/token**

The OWT tokenizer compresses slightly better because its larger vocabulary (32K vs 10K) produces longer tokens.

(b) Using the TinyStories tokenizer on OpenWebText drops the compression ratio to **3.22 bytes/token**. The TinyStories tokenizer lacks vocabulary entries for URLs, technical terms, and diverse punctuation found in web text.

(c) Single-threaded encoding throughput is approximately **0.67 MB/s**. At that rate, tokenizing The Pile (825 GB) would take roughly **342 hours** with a single core.

(d) Token IDs are stored as `uint16` NumPy arrays. This is appropriate because both vocabularies (10K and 32K) are under 65,536 (the uint16 maximum), giving storage savings.

Section 3: Transformer Language Model Architecture

Problem (linear): Implementing the Linear Module (1 pt)

File: `ece496b_basics/nn.py` — class `Linear`

Custom `nn.Module` subclass performing (no bias). Stores weight as `nn.Parameter` of shape `(out_features, in_features)`. Initialized with truncated normal: , truncated at .

Problem (embedding): Implement the Embedding Module (1 pt)

File: `ece496b_basics/nn.py` — class `Embedding`

Custom `nn.Module` subclass. Stores embedding matrix as `nn.Parameter` of shape `(num_embeddings, embedding_dim)`. Forward method indexes into the matrix using input token IDs. Initialized with truncated normal , truncated at .

Problem (rmsnorm): Root Mean Square Layer Normalization (1 pt)

File: `ece496b_basics/nn.py` — class `RMSNorm`

Implements , where . Learnable gain parameter initialized to ones.

Problem (positionwise_feedforward): Implement the Position-wise Feed-Forward Network (2 pts)

File: `ece496b_basics/nn.py` — class `SwiGLU`

Implementswhere . Three weight matrices:, . set to , rounded up to the nearest multiple of 64.

Problem (rope): Implement RoPE (2 pts)

File: `ece496b_basics/nn.py` — class `RotaryPositionalEmbedding`

Implements Rotary Position Embeddings. Applies pairwise rotation to query and key vectors based on token positions.

Problem (softmax): Implement Softmax (1 pt)

File: `ece496b_basics/nn.py` — function `softmax(tensor, dim)`

Subtracts the maximum value along the specified dimension before exponentiating.

Problem (scaled_dot_product_attention): Implement Scaled Dot Product Attention (5 pts)

File: `ece496b_basics/nn.py` — function `scaled_dot_product_attention(Q, K, V, mask=None)`

Computes. Supports optional boolean mask (True = attend, False = mask with).

Problem (multihead_self_attention): Implement Causal Multi-Head Self-Attention (5 pts)

File: `ece496b_basics/nn.py` — class `MultiHeadSelfAttention`

Implements multi-head self-attention with masking. Parameters:and . Projects input, splits into heads, applies RoPE to Q and K, applies mask, computes attention, concatenates heads, and projects output.

Problem (transformer_block): Implement the Transformer Block (3 pts)

File: `ece496b_basics/nn.py` — class `TransformerBlock`

Pre-norm architecture: , then . Contains two RMSNorm layers, one MultiHeadSelfAttention, and one SwiGLU FFN.

Problem (transformer_lm): Implementing the Transformer LM (3 pts)

File: `ece496b_basics/nn.py` — class `TransformerLM`

Full decoder-only transformer: token embedding → num_layers TransformerBlocks → final RMSNorm → LM head (Linear). Returns logits of shape (batch_size, seq_len, vocab_size).

Problem (transformer_accounting): Transformer LM Resource Accounting (5 pts)

(a) The model has **2,127,057,600 (~2.13 billion) trainable parameters**. At 4 bytes per float32 parameter, loading requires **8.51 GB**.

Component	Formula	Parameters
Token embedding		80,411,200
Per transformer block:		

Component	Formula	Parameters
— RMSNorm (×2)		3,200
— Q/K/V projections		7,680,000
— Output projection		2,560,000
— SwiGLU ⟮		30,720,000
— **Block subtotal**		**40,963,200**
48 blocks total		1,966,233,600
Final RMSNorm		1,600
LM head		80,411,200
Grand total		**2,127,057,600**

Memory: bytes = **8.51 GB**.

(b) Total forward pass FLOPs: **~4,513 GFLOPs**. (, , , , , ,)

Per transformer block (×48 blocks):

Matrix multiply	Dimensions ()	FLOPs ()	Value
Q projection			5.24B
K projection			5.24B
V projection			5.24B
attention scores			3.36B
Attention × V			3.36B
Output projection			5.24B
SwiGLU			20.97B
SwiGLU			20.97B
SwiGLU			20.97B
Per block total			**90.59B**

Non-block operations:

Matrix multiply	FLOPs	Value
LM head (output embedding)		164.02B

Grand total: = 4,512.3 GFLOPs 4,513 GFLOPs

Note: Embedding lookup is an index operation (no multiply), so it contributes zero FLOPs.

(c) The **FFN (SwiGLU)** requires the most FLOPs — approximately 67% of total for GPT-2 XL.

(d) Configs: Small (12L, D=768, 12h, d_ff=3,072), Medium (24L, D=1,024, 16h, d_ff=4,096), Large (36L, D=1,280, 20h, d_ff=5,120), XL (48L, D=1,600, 25h, d_ff=6,400). All use S=1,024, V=50,257.

Absolute FLOPs (GFLOPs):

Component	Small	Medium	Large	XL
Attn projections (QKV+O)	58.0	206.2	483.3	1,007.6
scores	19.3	51.5	96.5	161.1
Attention × V	19.3	51.5	96.5	161.1
FFN (W_1 + W_2 + W_3)	174.1	618.5	1,449.8	3,022.8
LM head	79.0	105.3	131.6	164.0
Total	**349.7**	**1,033.0**	**2,257.7**	**4,516.6**

Proportional breakdown (%):

Component	Small	Medium	Large	XL
Attn projections (QKV+O)	16.6%	20.0%	21.4%	22.3%
scores	5.5%	5.0%	4.3%	3.6%
Attention × V	5.5%	5.0%	4.3%	3.6%
FFN (SwiGLU)	49.8%	59.9%	64.2%	66.9%
LM head	22.6%	10.2%	5.8%	3.6%

As models grow, the **FFN** takes up proportionally more FLOPs (50% → 67%) because its cost scales as , which grows with both depth and width. The **LM head** shrinks dramatically (23% → 4%) because its cost () is independent of num_layers — it is a fixed overhead that gets amortized over more block computation. The **attention score computations** (, attn×V) also shrink proportionally (11% → 7%) because they scale as , only linear in , while the projection and FFN costs are quadratic in .

(e) At context length 16,384 for GPT-2 XL:

Component	S=1,024	S=16,384
Attn projections	22.3%	10.8%
+ Attn×V (combined)	7.2%	55.2%
FFN (SwiGLU)	66.9%	32.3%
LM head	3.6%	1.8%
Total	**4.5 TFLOPs**	**149.5 TFLOPs**

Total FLOPs jump ~33x due to quadratic attention scaling ( ). Attention goes from 7% to 55% of total compute. This is why efficient attention mechanisms (FlashAttention, sparse attention) are essential for long-context models.

Section 4: Training a Transformer LM

Problem (cross_entropy): Implement Cross Entropy

File: `ece496b_basics/nn.py` — function `cross_entropy(logits, targets)`

Problem (learning_rate_tuning): Tuning the Learning Rate (1 pt)

Using decaying SGD on a toy problem (minimize):

Step	lr=10	lr=100	lr=1000
0	24.17	24.17	24.17
1	15.47	24.17	8,725.10
2	11.40	4.15	1,506,962.38
3	8.92	0.10	167,633,472
4	7.23	0.00	13.6B
9	3.25	0.00	

At lr=10 the loss decays slowly; at lr=100 it converges rapidly to zero within 4 steps; at lr=1000 the loss diverges exponentially to . The learning rate has a narrow sweet spot — too low wastes compute, too high causes catastrophic divergence.

Code & output: see `notebooks/archive/ECE405_Homework1_ipynb_Pavel_Bushuyeu.ipynb`, *cell-33.*

Problem (adamw): Implement AdamW (2 pts)

File: `ece496b_basics/optimizer.py` — class `AdamW`

Problem (adamwAccounting): Resource Accounting for Training with AdamW (2 pts)

(a) Let = batch*size, = context_length, = num_layers, $D = d*{\text{model}} = num*heads, = vocab_size, $d*{ff} = 4D = total parameters.

Parameters: bytes

Gradients: bytes (same shape as parameters)

Optimizer state: bytes (first moment + second moment , each same shape as parameters)

Activations (4 bytes per float32 element, stored for backprop):

Component	Saved activation	Elements per block
RMSNorm input (×2)	input to each RMSNorm	
QKV projections	Q, K, V outputs	
scores	attention score matrix	
Softmax output	attention weights	
Attention × V	weighted values	
Output projection input	concat of heads	
FFN output	pre-SiLU activation	
SiLU output	post-activation	
FFN input	gated values	

Per block total:

Non-block activations:

- Final RMSNorm input:

- LM head output (logits):
- Cross-entropy (stores logits):

Non-block total:

Activation memory: bytes

Total peak memory:

(b) Substituting GPT-2 XL values (, , , , ,):

- Fixed cost (): bytes 34.03 GB
- Per-sample activation: bytes 16.46 GB

For 80 GB: → max batch size = **2**.

(c) FLOPs per optimizer step, where = number of parameters.

Per parameter, AdamW performs these operations each step:

Operation	**FLOPs**
	2 (multiply + add)
	3 (square + multiply + add)
	1 (divide)
	1 (divide)
	4 (sqrt + add + divide + multiply)
(weight decay)	3 (multiply + multiply + subtract)
Update	1 (subtract)
Total per parameter	**~15**

For GPT-2 XL: GFLOPs per step — negligible compared to the forward/backward pass (~4,500+ GFLOPs).

(d) Training GPT-2 XL for 400K steps, batch size 1024, single A100 at 50% MFU:

- A100 peak: 19.5 TFLOP/s → effective at 50% MFU: TFLOP/s
- Forward FLOPs per step: TFLOPs (from part b)
- Backward FLOPs per step: TFLOPs
- Total per step: TFLOPs
- Time per step: seconds

- **Total: seconds days 18 years**

GPT-2 XL was trained on hundreds of GPUs in parallel — a single A100 would need ~18 years at 50% utilization.

Problem (learning_rate_schedule): Implement Cosine Learning Rate Schedule with Warmup

File: `ece496b_basics/optimizer.py` — function `get_lr_cosine_schedule(it, max_learning_rate, min_learning_rate, warmup_iters, cosine_cycle_iters)`

Implements three-phase schedule:

1. **Warmup** ():
2. **Cosine annealing** ():
3. **Post-annealing** ():

Problem (gradient_clipping): Implement Gradient Clipping (1 pt)

File: `ece496b_basics/optimizer.py` — function `gradient_clipping(parameters, max_l2_norm)`

Computes global L2 norm across all parameter gradients. If the norm exceeds the maximum, scales all gradients by where . Modifies gradients in place.

Section 5: Training Loop

Problem (data_loading): Implement Data Loading (2 pts)

File: `ece496b_basics/data.py` — function `get_batch(dataset, batch_size, context_length, device)`

Takes a numpy array of token IDs, samples random starting indices, and returns (inputs, targets) tensor pairs of shape (batch_size, context_length). Inputs and targets are offset by one position. Uses `np.memmap` for memory-efficient loading of large datasets.

Problem (checkpointing): Implement Model Checkpointing (1 pt)

File: `ece496b_basics/checkpointing.py`

- `save_checkpoint(model, optimizer, iteration, out)` — saves model state_dict, optimizer state_dict, and iteration number via `torch.save`
- `load_checkpoint(src, model, optimizer)` — restores model and optimizer states via `load_state_dict`, returns the saved iteration number

---

Problem (training_together): Put It Together (4 pts)

File: `train.py`

Full training script with:

- CLI argument parsing (model architecture, optimizer hyperparameters, training config)
- Memory-efficient data loading via `np.memmap`
- Checkpoint saving/loading for training resumption
- W&B logging of train/val loss, perplexity, learning rate, throughput
- Cosine LR schedule with warmup
- Gradient clipping (max_norm=1.0)
- Sample text generation during training
- Ablation flags: `--no_rope`, `--no_rmsnorm`, `--post_norm`, `--ffn_silu`, `--weight_tying`
- bf16 mixed precision (`--use_amp`), flash attention (`--use_flash`), gradient accumulation (`--grad_accum_steps`)

Section 6: Generating Text

Problem (decoding): Decoding (3 pts)

File: `ece496b_basics/nn.py` — function `generate()`

Implements autoregressive decoding with:

- User-provided prompt (prefix token IDs)
- Maximum token count control
- **Temperature scaling:** divides logits by before softmax (lower → sharper/greedier, higher → more random)

- **Top-p (nucleus) sampling:** sorts probabilities descending, keeps the smallest set whose cumulative probability exceeds , zeros out the rest, renormalizes, and samples
- Context window truncation when generation exceeds model's context length
- Stops on `<|endoftext|>` token

Section 7: Experiments

Problem (experiment_log): Experiment Logging (3 pts)

Logging infrastructure: W&B integration in `train.py`. Logs training loss, validation loss, perplexity, learning rate, and throughput (tokens/sec) at configurable intervals. All experiments tracked under the `ece405-assignment1` W&B project.

Experiment log: All experiments ran on a Lambda Labs 1x GH200 (96 GB VRAM, ARM64 Grace CPU). 39 total runs (27 finished, 6 failed, 5 crashed). Full log: `notebooks/experiment_log.md`. Analysis & plots: `notebooks/experiments_analysis.ipynb`.

Problem (learning_rate): Tune the Learning Rate (3 pts)

(a) Learning rate sweep on TinyStories (36M params, bs=128, 5000 steps):

Learning Rate	Val Loss	Perplexity
3e-4	1.529	4.61
5e-4	1.443	4.23
1e-3	1.377	3.96
2e-3	**1.349**	**3.85**
3e-3	1.350	3.86
5e-3	2.290	9.88
1e-2	2.529	12.54

Best learning rate: 2e-3, with 1e-3 through 3e-3 forming a near-optimal performance. Val loss 1.349 < 1.45 target.
Full log: `notebooks/experiment_log.md`

(b) The edge of stability lies between 3e-3 and 5e-3. At 3e-3 training is perfectly stable (loss 1.350), but at 5e-3 it oscillates and settles at 2.290. The optimal LR sits at about **40% of the instability threshold**, consistent with the folk wisdom that the best learning rate is "at the edge of stability."

Problem (batch_size_experiment): Batch Size Variations (1 pt)

Swept batch sizes from 1 to 512, with both fixed LR (2e-3) and tuned LR (, capped at 3e-3 for bs=512):

BS	Fixed LR (2e-3)	Tuned LR	Tuned Val Loss
1	3.597	1.8e-4	2.834
8	2.464	5e-4	1.971
32	1.824	1e-3	1.638
128	1.349	2e-3	1.400
256	1.282	2.8e-3	1.271
512	1.228	3e-3	1.226

Key findings: (1) Larger batch sizes achieve lower loss per step but similar token-efficiency when plotted against total tokens processed. (2) Small batch sizes need proportionally lower LRs — at bs=1, tuning LR improves loss by 21%. (3) Very small batch sizes (1, 8) remain less token-efficient even with tuned LRs.

Problem (generate): Generate Text (1 pt)

The 36M-param TinyStories model (val loss 1.349) generates coherent children's stories:

Sample (temperature=0.8, top-p=0.9, 144 tokens):

> Once upon a time, in a big, big forest, there was a lazy cat named Tom. Tom liked to sleep all day and never wanted to play with his friends. One day, Tom saw a little bird named Tim. Tim was sad because he had no friends. Tom wanted to help Tim, so he said, "I will invite you to my house, and we can be friends." Tim was happy and said, "Yes, I will come!" At the house, Tom found a big, round ball. He told Tim, "Let's play with this ball!" Tim and Tom played with the ball and had lots of fun. They became good friends and played together every day.

Fluency: Lower temperature (0.5) gives less artistic but more coherent text; higher temperature (1.0) adds variety but introduces occasional logical lapses.

Problem (layer_norm_ablation): Remove RMSNorm and Train (1 pt)

Replaced all RMSNorm layers with identity functions. Tested at three learning rates (2e-3, 5e-4, 1e-4): the model produces NaN loss at all three. Without normalization, activations grow exponentially.

Problem (pre_norm_ablation): Implement Post-norm and Train (1 pt)

Post-norm (normalizing after the sub-layer instead of before) **converges but is significantly worse**: val loss 1.972 vs baseline 1.349 (+46%). In pre-norm, the residual path is clean — the input passes through without transformation. In post-norm, normalization happens after the residual addition, so gradients must flow through high-variance unnormalized activations, creating a bottleneck that slows optimization.

See `notebooks/experiments_analysis.ipynb`, cell 14 — ablation val loss curves vs baseline.

Problem (no_pos_emb): Implement NoPE (1 pt)

Removing all positional encoding (RoPE) only hurts by 3% (val loss 1.390 vs baseline 1.349). The gap would likely widen with longer sequences, but at context length 256 the model manages well without explicit positional encoding.

See `notebooks/experiments_analysis.ipynb`, cell 14 — ablation val loss curves vs baseline.

Problem (swiglu_ablation): SwiGLU vs. SiLU (1 pt)

SwiGLU: 3 matrices, d_ff=1,408, 2.16M params/block;
SiLU: 2 matrices, d_ff=2,048, 2.10M params/block:

SwiGLU achieves val loss 1.349 vs SiLU's 1.359 — 0.7% advantage.

The gating mechanism provides marginal benefit at this scale; it likely compounds at larger scales.

See `notebooks/experiments_analysis.ipynb`, cell 14 — ablation val loss curves vs baseline.

Problem (main_experiment): Experiment on OWT (2 pts)

Same architecture with 32K vocabulary, trained for 10K steps on OpenWebText:

Metric	TinyStories (5K steps)	OWT (10K steps)
Val Loss	1.349	3.799
Perplexity	3.85	44.66

The higher loss — web text is more diverse and complex. The model generates grammatically correct text with appropriate register (news/editorial tone) but not very coherence:

> "The reality of U.S. anti-government efforts is that the ability to profit from the continued and unregulated nature of U.S. anti-national forces has been nothing less than a sort of perseverance..."

Plausible-sounding phrases without structure.

Learning curves: see `notebooks/experiments_analysis.ipynb`, cell 18 — OWT train/val loss.

Problem (leaderboard): Leaderboard (6 pts)

Best result: val loss 3.738 in 0.65 hours. Optimizations implemented:

1. **bf16 mixed precision** (`train.py:330`): `torch.amp.autocast("cuda", dtype=torch.bfloat16)` — halves memory per activation, enables bf16 tensor core math. Throughput: 328K → 491K tok/s (+50%)
2. **Flash Attention** (`nn.py:269`): `torch.nn.functional.scaled_dot_product_attention(Q, K, V, is_causal=True)` — PyTorch dispatches to FlashAttention-2 on CUDA. memory instead of for attention
3. **Weight tying** (`train.py:266`): `model.lm_head.weight = model.token_embeddings.weight` — shares embedding and output projection, saving

parameters (58M → 42M)

4. **torch.compile** (`train.py:277`): `torch.compile(model)` with `torch.set_float32_matmul_precision("high")` — kernel fusion + TF32 matmuls. Combined with bf16+flash: 560K tok/s (+70% vs baseline)
5. **Gradient accumulation** (`train.py:327`): accumulate gradients over multiple micro-batches before stepping — enables larger effective batch sizes without proportional memory cost

Key experiment results:

Experiment	Config	Params	BS	Steps	Val Loss	Time	Tokens
V2-1	**d=512, 8L, 8h + weight tying**	**42M**	**128**	**40K**	**3.738**	**+0.65h**	**1.31B**
V2-2	d=640, 10L, 10h + weight tying	~70M	128	20K	4.744	0.43h	655M
V2-3	d=512, 8L, 8h + weight tying	42M	256	20K	3.816	0.60h	1.31B

- The 42M model trained on 1.31B tokens (31 tokens/param) outperforms the 70M model trained on 655M tokens (9 tokens/param). Within a fixed compute budget, data beats parameters.
- More frequent smaller updates (V2-1: bs=128, 40K steps) beat less frequent larger updates (V2-3: bs=256, 20K steps)

Throughput on GH200 (small model, bf16 + flash + torch.compile): 556K tok/s (bs=128), 602K tok/s (bs=256). Baseline fp32 without flash: 329K tok/s.

Learning curves: see `notebooks/experiments_analysis.ipynb`, cells 21–22 — leaderboard val loss vs tokens and summary table.