Homework 1

Build a Language Model

11-711 Advanced Natural Language Processing, Fall 2026

Released: September 1 **Due:** September 24, 11:59 PM

Late submissions close: October 14, 11:59 PM

Individual assignment 200 points 10% of your grade

In this homework you will build a language model end to end. You will train a tokenizer, implement the Llama-2 architecture, pretrain a model under a fixed compute budget, evaluate what it learned, and adapt it to a downstream task.

There are two Gradescope submissions. Upload `submission.zip` (built by `scripts/zip-submission.sh`) to **Assignment1: Code** and `report.pdf` to **Assignment1: Report**. Coding questions are graded only by the autograder, written questions only from the report; nothing is submitted twice. Use the provided LaTeX template, and when uploading the report, mark the pages that answer each question.

Late policy. Per the syllabus, each day (or part of a day) a submission is late costs 5% of your grade for this homework; both submissions must be in by October 14, after which Gradescope closes. Requests for excused late days go to the course email at least 7 days before the deadline.

On the use of AI. Per the syllabus, you may use AI to help you write *code*. You may **not** use it to write any part of your report. Question 6 asks you to describe how you used AI; per the syllabus, using AI without making an honest attempt at that question is itself an academic integrity violation.

Roughly half the points are written analysis, and those questions ask about *your* runs: your loss curves, your search table, your generations. Numbers that do not match your submitted checkpoint will not receive credit.

Problem 1: Tokenization (35 points)

A tokenizer maps a string $s \in \mathcal{A}^*$ over an alphabet $\mathcal{A}$ to a sequence of tokens $\mathbf{x} \in \mathcal{V}^*$. Tokenizers are complete in their alphabet: any string over $\mathcal{A}$ can be tokenized. Everything downstream, what the model can represent, how much context a document consumes, how much it costs to serve, inherits from this choice.

[Question 1.1] (***Writing, 12 points***) We might reasonably expect a tokenizer to satisfy the following:

- **Injective**: for all $s \neq s'$, $\text{tokenize}(s) \neq \text{tokenize}(s')$.
- **Invertible**: for any string s, $s = \text{detokenize}(\text{tokenize}(s))$.
- **Preserves concatenation**: $\text{tokenize}(s + s') = \text{tokenize}(s) + \text{tokenize}(s')$.

All three are violated by the tokenizers of widely used language models.

DELIVERABLES FOR Q1.1

A. Find counterexamples showing that a popular, publicly available tokenizer fails each property. **Use `google-bert/bert-base-cased`.** The autograder has it cached and may have no network access, so another choice risks failing there even if it works locally. Write code in `tests/test_tokenizer.py` so that the assertions in `test_not_injective()`, `test_not_invertible()` and `test_not_preserving_concat()` all pass. In your report, describe in at most two sentences per test what violation you found.

B. Explain why it is essentially impossible to build a non-trivial tokenizer that preserves concatenation.[a]

[a] A trivial tokenizer is one whose vocabulary is the alphabet itself.

[Question 1.2] (***Coding, 15 points***) Byte-pair encoding (BPE) *learns* a tokenizer from a corpus. It starts with the alphabet as its vocabulary and repeatedly merges the most frequent adjacent pair of tokens into a new token. Each iteration adds one token while keeping every existing one.

DELIVERABLES FOR Q1.2

A. Implement `ASCIIBPETokenizer.merge()` in `src/tokenizer/bpe.py`. It must pass `test_one_merge()`, `test_two_merges()` and `test_100_merges()` in `tests/test_tokenizer.py`. *You are given helpers* in the same file: `string_to_ascii()`, `compute_bigram_statistics()` and `replace_bigram()`. Read them first, you should not need to write your own.

B. Implement `encode()` and `decode()`. These must pass `test_encode()` and `test_decode()`.

Additional hidden tests may be run after the deadline.

[Question 1.3] (***Writing, 4 points***) We have provided a Unicode BPE tokenizer trained only on English text (`data/english-tokenizer.json`). Run `python -m src.tokenizer.visualize` to load it and inspect the tokens it learned.

DELIVERABLES FOR Q1.3

Answer in at most two sentences each.

A. What is the longest token you see, and what does that tell you about the training data?

B. How can BPE training compromise the privacy of the corpus it was trained on?

[Question 1.4] (***Writing, 4 points***) A tokenizer trained on one distribution spends more tokens per unit of meaning on anything else. Using the Llama SentencePiece tokenizer (`src/llama/sp_tokenizer.py`), measure how many tokens it takes to encode comparable amounts of content across different kinds of text. `python -m src.tokenizer.inspect_tokenizer` runs the core measurements: it prints how sample digits tokenize and tokens per character for sample texts in English, source code and a non-Latin script; `--text` covers anything else, including the exact number ranges in A.

DELIVERABLES FOR Q1.4

Answer in at most three sentences each.

A. **Digits.** Tokenize the numbers 1–20, then 1000–1020. How are multi-digit numbers split? Does the same number of digits always produce the same number of tokens? Explain what this tokenization choice means for a model learning arithmetic, compared to a tokenizer that merges frequent digit sequences into single tokens.

B. **Domains.** Compare tokens-per-character for three texts of similar length: English prose, source code, and a passage in a non-Latin script. Which is most expensive, and why?

C. **Who pays.** API pricing and context windows are both denominated in tokens. State one concrete consequence of your answer to (B) for users writing in the most expensive of the three.

Problem 2: Implementing Llama-2 (37 points)

You will implement the core of the Llama-2 architecture. All relevant code is in `src/llama/`. **Do not modify the classes, functions or their arguments**, and do not import new dependencies; both will break the autograder. Aim for an efficient implementation: call PyTorch operations over whole tensors rather than looping. You may not use layers that make the exercise trivial (e.g. `torch.nn.TransformerDecoder` or `F.scaled_dot_product_attention`); if you are unsure whether something is allowed, ask on Piazza.

[Question 2.1] (***Coding, 24 points***) Implement the following. Each is marked `# todo` in the source.

- `RMSNorm._norm`: root-mean-square normalization.
- `Attention.compute_query_key_value_scores`: scaled dot-product attention with a causal mask.
- `LlamaLayer.forward`: one transformer block.
- `Llama.generate`: temperature and top-p (nucleus) sampling.
- `apply_rotary_emb` in `src/llama/rope.py`: rotary position embeddings. This one is fiddly; `tests/test_llama.py` tests it in isolation.

If you get stuck, Attention Is All You Need (Vaswani et al., 2017) and The Annotated Transformer are useful references for understanding the mechanics of attention and transformer blocks in code.

DELIVERABLES FOR Q2.1

Upload your code to Gradescope and make sure the tests in `tests/test_llama.py` pass.

The test that matters most is `test_llama_forward_matches_reference`. It loads real pre-trained weights (`stories42M.pt`, an 8-layer, 42M-parameter Llama trained on TinyStories) and checks your model reproduces their activations. Shape-correct but subtly wrong code fails it. If the reported maximum difference is around 10^0, suspect your normalization or attention; around 10^{-4} is ordinary floating-point noise.

[Question 2.2] (***Writing, 6 points***) Press and Wolf (2017) propose tying the input embedding and the output projection.

DELIVERABLES FOR Q2.2

A. In at most three sentences, explain what weight tying does and why it helps.

B. Let d be the hidden size, v the vocabulary size, b the batch size and s the sequence length. Given hidden states $h \in \mathbb{R}^{b\times s\times d}$ and token embeddings $E \in \mathbb{R}^{v\times d}$, write one line of PyTorch that computes the token logits using weight tying.

[Question 2.3] (***Writing, 7 points***) Llama-2 uses grouped-query attention: several query heads share one key/value head. During generation the model caches keys and values for every position it has already seen.

DELIVERABLES FOR Q2.3

A. Write an expression for the number of bytes the KV cache occupies, in terms of the number of layers L, key/value heads n_{kv}, head dimension d_h, sequence length s, batch size b, and bytes per element.

B. Using the `configs/llama-base.yaml` values, compute the cache size for batch size 1 at $s = 256$ and, hypothetically, at $s = 8192$, in `float16`.

C. GQA reduces n_{kv} while leaving the number of query heads unchanged. Given your expression, what does that buy, and what is the cost? Why is this a bigger deal at inference time than during training?

Problem 3: Pretraining under a compute budget (68 points)

You will now pretrain your model on TinyStories, a corpus of machine-generated children's stories. We provide the corpus pre-tokenized, so you do not need to run your Part 1 tokenizer over it. Training is budgeted in **FLOPs rather than wall-clock time**, so the assignment is fair across hardware. `src/train.py` prints the projected cost before training begins and warns if you exceed the final 10^{17} budget; for Q3.5's per-configuration budget, check the printed projection against 10^{15} yourself. Total cost is

$$\begin{aligned}\text{FLOPs} = {} & \texttt{flops\_per\_token} \times \texttt{num\_training\_steps} \\ & \times \texttt{grad\_accumulation\_steps} \times \texttt{batch\_size} \times \texttt{seq\_len},\end{aligned}$$

where $\texttt{flops\_per\_token} = 6N + 12Ld\,s_{\max}$ for a model with N parameters, L layers and hidden size d. Every hyperparameter moves this: a larger model buys you fewer steps.

[Question 3.1] (***Writing, 5 points***) `tests/test_train.py` contains `test_train_sanity_check`, which trains on a sequence of uniformly random tokens drawn from $\{0, \ldots, 7\}$ and *asserts the loss stays above* $\log(8) \approx 2.08$.

DELIVERABLES FOR Q3.1

A. Why is log(8) the floor for this data? Show the reasoning.

B. Describe a specific bug in the attention implementation that would let a model score *below* this floor, and explain the mechanism.

C. This test is worth zero points but failing it invalidates Questions 3.4–3.6. Why would those results be meaningless?

[Question 3.2] (***Coding, 28 points***) Implement the following in `src/train.py`: `random_batch_sampler`, `sequential_batch_sampler`, `cosine_lr_schedule` and `compute_language_modeling_loss`.

Also implement `AdamW.step` in `src/optim/adamw.py`, with decoupled weight decay and gradient clipping. Use the efficient bias correction from the end of Section 2 of Kingma & Ba (2014) rather than forming $\hat{m}$ and $\hat{v}$, and fold the learning rate into the weight decay update. Your training runs in Part 3 use *your* optimizer, not `torch.optim.AdamW`.

DELIVERABLES FOR Q3.2

Upload your code and make sure `tests/test_train.py` and `tests/test_optimizer.py` pass, including `test_train_sanity_check` (see Q3.1).

[Question 3.3] (***Writing, 6 points***) Cosine annealing with warmup has two phases. During warmup, $t \in [0, a)$, the learning rate rises linearly from 0 to $\mathrm{lr}_{\max}$. During annealing, $t \in [a, b)$, it decays from $\mathrm{lr}_{\max}$ to $\mathrm{lr}_{\min}$ along a half-cosine. For $t \geq b$ it stays at $\mathrm{lr}_{\min}$.

DELIVERABLES FOR Q3.3

A. Write the expression for the learning rate during warmup, and during annealing.

B. In at most two sentences: why warm up at all, and what does annealing buy over a constant learning rate?

[Question 3.4] (***Writing, 5 points***) Train the smoke-test configuration: `python -m src.train configs/llama-tiny.yaml`. It takes about a minute and is not meant to produce a good model.

DELIVERABLES FOR Q3.4

A. Report the final validation loss.

B. Roughly how many steps before the training loss stops improving?

C. Include the training curve (a screenshot from Weights & Biases is fine).

[Question 3.5] (***Writing, 14 points***) Now search for a good configuration. **Each configuration you try must use at most 10^{15} FLOPs** — about four minutes each on a Google Colab T4 instance. Within that budget you may change anything that affects cost: `dim`, `n_layers`, `n_heads`,

`n_kv_heads`, `max_seq_len`, `batch_size`, `seq_len`, `grad_accumulation_steps`, `num_training_steps`, and the learning rates. Evaluate using validation perplexity.

DELIVERABLES FOR Q3.5

Report **at most three** configurations. For each, state what you changed and the validation perplexity you got.

A. Describe your experimental procedure in one paragraph. What did you vary, and why those knobs rather than others?

B. Describe your results in one paragraph. Which change mattered most?

C. Paste the YAML of your best configuration.

D. You held compute fixed and traded model size against training steps. What does your data suggest about that trade-off, and how does it relate to what you know about scaling laws?

[Question 3.6] (***Coding, 10 points***) Train your final model using the configuration you chose. You may spend up to 10^{17} FLOPs, 100× the search budget, so budget roughly six hours on a Google Colab T4 instance.

Keep the model architecture you reported in Q3.5: `dim`, `n_layers`, `n_heads`, `n_kv_heads` and `max_seq_len`. Everything else is yours to change in order to use the larger budget: `num_training_steps`, `grad_accumulation_steps`, `batch_size`, `seq_len` (up to `max_seq_len`), and the learning rates. In practice you will mostly be raising `num_training_steps`.

DELIVERABLES FOR Q3.6

Report your final validation loss and perplexity, and submit `model.pt` and `config.yaml` as instructed in the README. A hidden test measures perplexity on a held-out split you have not seen: **perplexity ≤ 5.5 earns full credit, ≤ 8.0 earns half**.
Your own validation perplexity is a good proxy for this, so you do not need to guess. The held-out split is drawn from the same distribution as the validation set you can see, and in our experience the two agree closely. If your validation perplexity is comfortably under 5.5, you are done and can stop tuning. For scale, the smoke-test configuration `llama-tiny` reaches roughly 128. Gradescope caps uploads at 100MB; `train.py` warns you if your checkpoint is too large.

Problem 4: Generation and perplexity (35 points)

[Question 4.1] (***Coding, 12 points***) Implement `softmax_with_temperature` in `src/generate.py`, and the top-p sampling branch of `Llama.generate` if you have not already (Q2.1).

DELIVERABLES FOR Q4.1

Upload your code and make sure `tests/test_generate.py` passes.

[Question 4.2] (***Writing, 11 points***) Compare your Part 3 model against `stories42M`. Both are Llamas trained on TinyStories, so the architecture family is held fixed; what differs is model size and how much training went in. Choose three prefixes, from `data/prefixes.jsonl` or your own, and generate at least 32 tokens from each model, at several temperatures. `python -m src.compare_generations --config <your config> --checkpoint <your model.pt>` does this side by side for both models (temperatures 0.0, 0.7, 1.5 by default) and writes every generation to a jsonl file.

DELIVERABLES FOR Q4.2

A. Report your prefixes, the temperatures and `max_new_tokens` you used, and paste the generations from each model separately.

B. What happens to the generations as temperature approaches zero, and as it grows large? Answer in at most two sentences.

C. At a fixed temperature, compare the quality and perplexity of the two models' output. Report what you find and, in at most three sentences, why.

D. Your model and `stories42M` differ in both size and training tokens. What can you *not* conclude from this comparison, and what experiment would separate the two?

[Question 4.3] (***Writing, 12 points***) Now measure perplexity directly. `src/evaluate_perplexity.py` scores every document in a jsonl file and reports perplexity (its docstring has more detail):

```
python -m src.evaluate_perplexity --checkpoint data/stories42M.pt
    --documents data/ppl_examples.jsonl
```

`data/ppl_examples.jsonl` already contains an in-domain TinyStories passage and a Wikipedia passage. Pass `--strip-word the` to delete a word before scoring, and `--config <your config> --checkpoint <your model.pt>` to score with your own Part 3 model instead.

DELIVERABLES FOR Q4.3

A. **Function words.** Compute your model's perplexity on a held-out TinyStories sample. Then delete every instance of "the" and recompute. Report both numbers and explain the change in at most three sentences.

B. **Distribution shift.** Compute your model's perplexity on a few paragraphs of English Wikipedia. Compare it to (A). Explain the gap, and state what that implies about reporting a single perplexity number for a model.

Problem 5: Adapting the model (25 points)

Pretraining gives you a model that continues text. Most of what people want from language models is something else. Here you adapt the pretrained model to sentiment classification, on SST-5 (5-way) and CFIMDB (binary).

[Question 5.1] (***Coding, 10 points***) Implement `LlamaEmbeddingClassifier.forward` in `src/adapt.py`: pool a single vector out of the sequence, apply dropout, pass it through the classifier head, and return log-probabilities.

DELIVERABLES FOR Q5.1

Upload your code and make sure `tests/test_adapt.py` passes.

Note the padding-invariance tests. Batches are right-padded to their longest sequence, so `hidden_states[:, -1]` is a *padding* position for every sequence that is not the longest in its batch. Your pooling must find the last real token.

[Question 5.2] (***Writing, 8 points***) Run the model in two settings against `stories42M`: zero-shot prompting (`--option prompt`) and finetuning with a classification head (`--option finetune`), on both datasets.

DELIVERABLES FOR Q5.2

A. Report dev accuracy for all four runs (two settings × two datasets). Report dev only: your copy of `cfimdb-test.txt` carries placeholder labels (every label is 0), so a locally computed test accuracy is meaningless there, and your runs print `n/a` for it. Every run writes its prediction files to `outputs/preds/`; they are included in your code submission, and we score the test split against the real held-out labels ourselves.

B. Chance is 0.200 for 5-way SST-5 and 0.500 for binary CFIMDB, but chance is only an honest floor if a constant predictor cannot beat it. Compute the majority-class baseline of each dev set: the accuracy of always predicting its most common label. Then look at what your prompting run actually predicts. The first field of each line of the dev output file is the predicted label, so `cut -d'|' -f1 <dev-out> | sort | uniq -c` prints the whole distribution. Report both baselines and both distributions, and say in at most four sentences what they imply about whether zero-shot prompting is doing sentiment classification here. Give one reason about the model and one about the single-token scoring setup.

C. Finetuning does much better while starting from the same weights. What has changed, given that the pretrained model has seen no sentiment labels at all?

[Question 5.3] (***Writing, 7 points***) Now finetune *your own* Part 3 model on SST-5, using the same procedure and passing `--backbone-config` with the config your model was trained with. (SST-5 only, to keep the runs short.)

DELIVERABLES FOR Q5.3

A. Report dev accuracy for your model and for `stories42M`, finetuned identically.

B. Is the difference between the two larger than run-to-run variation? Finetune at least twice with different `--seed` values to find out. Then, in at most three sentences, say what your answer implies about what the extra pretraining compute behind `stories42M` bought on this task. **"The difference is within noise" is a perfectly good answer** if that is what you measure.

C. Both models were pretrained only on children's stories, yet both can be finetuned for movie review sentiment. What does that tell you about what pretraining actually provides?

Question 6: Use of AI (required, ungraded)

DELIVERABLES FOR Q6

Describe how you used AI tools on this assignment: which parts, and how. If you did not use any, say so. Per the syllabus this question is required, and failing to make an honest attempt at it is itself an academic integrity violation.

Bonus (up to 10 points)

Optional. Pick **one** and write it up in an appendix to your report: what you did, what you measured, and what you concluded. Include plots and training logs.

- **The tiniest adding Llama.** Pretrain a model to do 4-digit addition (`bonus/`). Find the smallest architecture reaching 100% test accuracy within the `--capacity` budget of `bonus/addition_run.py`; the todos live in `bonus/addition_data_generation.py` and `bonus/addition_run.py`. Report at least five ablations, including at least two that fail.
- **LoRA.** Implement `LoRALayer.forward` in `bonus/lora.py` and repeat Q5.2 with it. Report the fraction of parameters you trained (see `count_lora_parameters`) alongside the accuracy, and compare against full finetuning. Note that the update is scaled by α/r; explain what you chose and why.
- **Something else.** Continued pretraining for domain adaptation; an alternative positional encoding; a different attention variant; alternative decoding strategies. Clear it on Piazza first.

Acknowledgements. This code is based on llama2.c by Andrej Karpathy (MIT, see `LICENSE`). Parts of the code are also from the `transformers` library (Apache License 2.0), and it contains adaptations from nanoGPT and PyTorch (see `copyright/`).