CS336 Assignment 1 (basics): Building a Transformer LM

Answer Sheet

NewSunH

1 Assignment Overview

(nothing to solve)

2 Byte-Pair Encoding (BPE) Tokenizer

2.1 The Unicode Standard

Problem (`unicode1`): Understanding Unicode (1 point)

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

answer: `'\x00'`, or Unicode character `U+0000`, represents NUL (the null character).

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

answer: `'\x00'` in string representation, and invisible when using `print()`.

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

answer: Adding the NUL character to a string doesn't change what it looks like (in the most satuation). But there is actually a invisible character adding to the storage. When using UTF-8 typically, a zero byte (`0x00`) was added in the place of it.

2.2 Unicode Encodings

Problem (`unicode2`): Unicode Encodings (3 points)

(a) What are some reasons to prefer training our tokenizer on UTF-8 encoded bytes, rather than UTF-16 or UTF-32? It may be helpful to compare the output of these encodings for various input strings.

answer: UTF-8 is byte-based, and ASCII compatible. UTF-8 is much more effient than UTF-16/32, because UTF-16/32 using zeros to fill in the blanks, which will also influence tokens.

(b) Consider the following (incorrect) function, which is intended to decode a UTF-8 byte string into a Unicode string. Why is this function incorrect? Provide an example of an input byte string that yields incorrect results.

answer: It literally decode each byte. UTF-8 character might be multiple bytes, e.g. Chinese characters and emoji. e.g. `' 你好，世界'`.

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

answer: e.g. `0x8080`, which is invalid because continuation bytes cannot appear without a valid leading byte.

2.3 Subword Tokenization

2.4 BPE Tokenizer Training

(nothing to solve)

2.5 Experimenting with BPE Tokenizer Training

Problem (`train_bpe`): BPE Tokenizer Training (15 points)

implemented in `/cs336_basics/bpe.py`

Problem (`train_bpe_tinystories`): BPE Training on TinyStories (2 points)

(a) Train a byte-level BPE tokenizer on the TinyStories dataset, using a maximum vocabulary size of 10,000. Make sure to add the TinyStories `<|endoftext|>` special token to the vocabulary. Serialize the resulting vocabulary and merges to disk for further inspection. How many hours and memory did training take? What is the longest token in the vocabulary? Does it make sense?
answer: It takes 117.98s, 10GB RAM in total. The longest token is `Ġaccomplishment`, which is 15 bytes long and its id is `7160`.

(b) Profle your code. What part of the tokenizer training process takes the most time?
answer: In this dataset, pretokenize takes 1:19s, over 2/3 of total time was taken.

Problem (`train_bpe_expts_owt`): BPE Training on OpenWebText (2 points)

(a) Train a byte-level BPE tokenizer on the OpenWebText dataset, using a maximum vocabulary size of 32,000. Serialize the resulting vocabulary and merges to disk for further inspection. What is the longest token in the vocabulary? Does it make sense?
answer: It takes 4810.98s (1h 20m 10.98s). The longest token is `ÃĥÃĤÃĥÃĤÃĥÃĤÃĥÃĤÃĥÃĤÃĥÃĤÃĥ ÃĤÃĥÃĤÃĥÃĤÃĥÃĤÃĥÃĤÃĥÃĤÃĥÃĤÃĥÃĤÃĥÃĤÃĥÃĤ` (strange, I can't tell what it is) and `----------- -----------------------------------------------------` (64 dashes). It does make sence because I found them in the owt.

(b) Compare and contrast the tokenizer that you get training on TinyStories versus OpenWebText.

```
bytes/token (lower is better compression)
TS tok on TS: 3.979507
TS tok on OWT: 3.159600
OWT tok on TS: 3.890763
OWT tok on OWT: 4.344127
```

2.6 BPE Tokenizer: Encoding and Decoding

Problem (`tokenizer`): Implementing the tokenizer (15 points)

implemented in `/cs336_basics/tokenizer.py`

2.7 Experiments

Problem (`tokenizer_experiments`): Experiments with tokenizers (4 points)

(a) Sample 10 documents from TinyStories and OpenWebText. Using your previously-trained TinyStories and OpenWebText tokenizers (10K and 32K vocabulary size, respectively), encode these sampled documents into integer IDs. What is each tokenizer' s compression ratio (bytes/token)?
answer: I sampled 10 documents (lines) from each `*-valid.txt` file and measured bytes/token (lower is better). TinyStories tokenizer on TinyStories: `3.762821` bytes/token; OpenWebText tokenizer on OpenWebText: `4.308458` bytes/token.

(b) What happens if you tokenize your OpenWebText sample with the TinyStories tokenizer? Compare the compression ratio and/or qualitatively describe what happens.
answer: On the same 10-doc OpenWebText sample, using the TinyStories tokenizer gives `2.955631` bytes/token, which is *lower* than using the OpenWebText tokenizer (`4.308458`). Qualitatively, the OWT tokenizer learns some very long/rare byte tokens from noisy web text, while the TinyStories tokenizer tends to break these into smaller pieces, which in this sample leads to more effective compression.

(c) Estimate the throughput of your tokenizer (e.g., in bytes/second). How long would it take to tokenize the Pile dataset (825GB of text)?
answer: Measured on 50MB of `*-valid.txt`, throughput is about `5.76e6` bytes/s (TinyStories tokenizer on TinyStories valid) and `3.70e6` bytes/s (OWT tokenizer on OWT valid). At `3.70e6` bytes/s, tokenizing 825GB takes about `66.6` hours.

(d) Using your TinyStories and OpenWebText tokenizers, encode the respective training and development datasets into a sequence of integer token IDs. We recommend serializing the token IDs as a NumPy array of datatype uint16. Why is uint16 an appropriate choice?
answer: `uint16` can represent IDs up to 65535, which comfortably covers vocab sizes 10K/32K, and it halves storage compared to `int32` (2 bytes vs 4 bytes per token). This matters because the token ID sequences are very large and will be memory-mapped for training.

3 Transformer Language Model Architecture

3.1 Transformer LM

3.2 Output Normalization and Embedding

3.3 Remark: Batching, Einsum and Efficient Computation

(nothing to solve)

3.4 Basic Building Blocks: Linear and Embedding Modules

Problem (`linear`): Implementing the linear module (1 point)

implemented in `/cs336_basics/linear.py`

Problem (`embedding`): Implement the embedding module (1 point)

implemented in `/cs336_basics/embedding.py`

3.5 Pre-Norm Transformer Block

Problem (`rmsnorm`): Root Mean Square Layer Normalization (1 point)

implemented in `/cs336_basics/normalization.py`

Problem (`positionwise_feedforward`): Implement the position-wise feed-forward network (2 points)

implemented in `/cs336_basics/positionwise_feedforward.py`

Problem (`rope`): Implement RoPE (2 points)

implemented in `/cs336_basics/rope.py`

Problem (`softmax`): Implement softmax (1 points)

implemented in `/cs336_basics/attention.py`

Problem (`scaled_dot_product_attention`): Implement scaled dot-product attention (5 points)

implemented in `/cs336_basics/attention.py`

Problem (`multihead_self_attention`): Implement causal multi-head self-attention

implemented in `/cs336_basics/attention.py`

3.6 The Full Transformer LM

Problem (`transformer_block`): Implement the Transformer block (3 points)

implemented in `/cs336_basics/transformer.py`

Problem (`transformer_lm`): Implementing the Transformer LM (3 points)

implemented in `/cs336_basics/transformer.py`

Problem (`transformer_accounting`): Transformer LM resource accounting (5 points)

unfinished

4 Training a Transformer LM

4.1 Cross-entropy loss

Problem (`cross_entropy`): Implement Cross entropy

implemented in `/cs336_basics/train_transformer.py`

4.2 The SGD Optimizer

Problem (`learning_rate_tuning`): Tuning the learning rate (1 point)

As we will see, one of the hyperparameters that affects training the most is the learning rate. Let's see that in practice in our toy example. Run the SGD example above with three other values for the learning rate: 1e1, 1e2, and 1e3, for just 10 training iterations. What happens with the loss for each of these learning rates? Does it decay faster, slower, or does it diverge (i.e., increase over the course of training)?

answer: Running the toy SGD loop for 10 iterations, `lr=1` decreases the loss slowly while `lr=10` decreases it much faster. `lr=100` drives the loss to (near) zero within a few steps, but `lr=1000` diverges immediately and the loss explodes.

4.3 AdamW

Problem (`adamw`): Implement AdamW (2 points)

implemented in `/cs336_basics/train_transformer.py`

Problem (`adamwAccounting`): Resource accounting for training with AdamW (2 points)

unfinished

4.4 Learning rate scheduling

Problem (`learning_rate_schedule`): Implement cosine learning rate schedule with warmup

implemented in `/cs336_basics/train_transformer.py`

4.5 Gradient clipping

Problem (`gradient_clipping`): Implement gradient clipping (1 point)

implemented in `/cs336_basics/train_transformer.py`

5 Training loop

5.1 Data Loader

Problem (`data_loading`): Implement data loading (2 points)

implemented in `/cs336_basics/train_transformer.py`

5.2 Checkpointing

Problem (`checkpointing`): Implement model checkpointing (1 point)

implemented in `/cs336_basics/train_transformer.py`

5.3 Training loop

Problem (`training_together`): Put it together (4 points)

implemented in `/cs336_basics/train_transformer.py`

6 Generating text

Problem (`decoding`): Decoding (3 points)

implemented in `/cs336_basics/transformer.py` as `TransformerLm.generate`. It generates completions autoregressively from a user-provided prompt (`input_ids`) until hitting `eos_token_id` (e.g. `<|endoftext|>`) or reaching `max_new_tokens`. It supports temperature scaling (`temperature<=0` falls back to greedy decoding) and top-p (nucleus) sampling via `top_p`.

7 Experiments

7.1 How to Run Experiments and Deliverables

Problem (`experiment_log`): Experiment logging (3 points)

implemented in `/cs336_basics/experiment_logging.py`. The logger writes per-step metrics with both `step` and `wallclock_s` to `metrics.json`, and also stores run metadata (`run.json`) and free-form notes (`notes.log`). My running experiment log is maintained in `/experiments/experiment_log.md`.

7.2 TinyStories

Problem (`learning_rate`): Tune the learning rate (3 points)

The learning rate is one of the most important hyperparameters to tune. Taking the base model you've trained, answer the following questions:

(a) Perform a hyperparameter sweep over the learning rates and report the final losses (or note divergence if the optimizer diverges).

answer: I ran a learning-rate sweep over `lr_max` (peak learning rate in the warmup+cosine schedule) while keeping all other hyperparameters fixed (same 17M-ish model and `total_steps=10000`, processing `327,680,000` tokens when training completes). The final validation losses are:

`lr_max`	final validation loss	note
1e-4	2.7623	converged
2e-4	2.0991	converged
3e-4	1.8096	converged
5e-4	1.5811	converged
8e-4	1.4690	converged
1e-3	1.4368	converged (meets ≤ 1.45)
3e-3	1.3713	best run
3e-2	4.3973	does not learn well
30	—	diverged at step `188`

(Reproducibility note: these correspond to run directories under `outputs/runs/`, e.g. the best run is `outputs/runs/20260130T032122Z`, and a divergence run is `outputs/runs/20260130T064527Z`.)

(b) Folk wisdom is that the best learning rate is at the edge of stability. Investigate how the point at which learning rates diverge is related to your best learning rate.

answer: In this setup, catastrophic divergence happens only at extremely large `lr_max` (e.g., `lr_max=30` diverges quickly at step `188`). However, well before catastrophic divergence there is an *effective* stability boundary where optimization stops making meaningful progress: with `lr_max=3e-2` the run does not diverge, but the training loss barely decreases (about `5.66` $\rightarrow$ `4.41`) and the final validation loss is very poor (`4.3973`).

The best learning rate I found is `lr_max=3e-3` (final validation loss `1.3713`). This is about one order of magnitude below the point where training becomes effectively unstable (`3e-2`), and it is also far below the catastrophic divergence point (`3e1`). Empirically, performance improves as `lr_max` increases from `1e-4` up to a few `e-3`, and then degrades sharply by `3e-2`. This supports the edge of stability heuristic if we interpret the edge as the largest learning rate that still yields stable *and* productive learning dynamics, not merely not produce `NaN`.

Problem (`batch_size_experiment`): Batch size variations (1 point)

Vary your batch size all the way from 1 to the GPU memory limit. Try at least a few batch sizes in between, including typical sizes like 64 and 128.

answer: I ran a batch size sweep with the same TinyStories 17M-ish model (ctx=256, 4L/ 16H/ 512d/ 1344ff) and the same training code as in the learning-rate sweep. For each batch size, I adjusted `lr_max` (peak LR in the warmup+cosine schedule) using linear scaling from the best batch size 128 setting, and clipped it if it entered clearly unstable regimes.

How I ran it: I used a SLURM array job (see `scripts/slurm/batch_size_experiment_array.sbatch`). It sweeps `batch_size` $\in$ `{1,4,16,64,128,256,512}` and uses a roughly fixed token budget per run so that `batch_size=1` is feasible.

Baseline point (already measured): with `batch_size=128` and tuned `lr_max=3e-3`, the run reaches final validation loss `1.3713` (run `outputs/runs/20260130T032122Z`).

Results summary (fill from runs): after the sweep completes, I summarize the best final validation loss per batch size via `uv run python experiments/batch_size_results.py`, and include learning curves (train/valid loss vs step) for each batch size.

batch size	tuned `lr_max`	best final validation loss
1	2.34375e-5	6.5495
4	9.375e-5	3.3322
16	3.75e-4	2.1118
64	1.5e-3	1.8727
128	3e-3	1.3713
256	6e-3	1.9497
512	—	OOM (GPU memory limit)

Discussion (expected/observed pattern): smaller batch sizes tend to produce noisier gradients: training loss curves look jagged, but sometimes validation can improve slightly due to implicit regularization. As batch size increases, optimization becomes smoother and (up to a point) allows a larger learning rate (linear scaling), but very large batch sizes can hurt generalization or require retuning `lr_max` and warmup to avoid getting stuck. In practice, I found that `batch_size=128` is a good trade-off between throughput and validation loss for this model.

Problem (generate): Generate text (1 point)

Using your decoder and your trained checkpoint, report the text generated by your model. You may need to manipulate decoder parameters (temperature, top-p, etc.) to get fluent outputs.

answer: I generated text using the best TinyStories checkpoint (run `outputs/runs/20260130T032122Z`, `checkpoints/final.pt`) and my implementation of `TransformerLm.generate`. In practice, this model assigns very high probability to the `<|endoftext|>` token immediately after short prompts, so to obtain a 256-token sample I disabled sampling of the EOS token during decoding (equivalent to forbidding EOS).

Decoding settings: prompt = `Once upon a time, `, max_new_tokens = `512` (output contains ≥ 256 tokens), temperature = `0.9`, top-p = `0.95`, and EOS is forbidden.

Repro command: `uvrunpythonexperiments/generate_text.py--run-diroutputs/runs/20260130T032122Z--prompt"Onceuponatime,"--max-new-tokens512--temperature0.9--top-p0.95--ban-eos`

Generated text (≥ 256 tokens):

```
Once upon a time, `Please save me and my little heart. But when you make a wish, it' s all yours,
↪ I just couldn' t get out of here. All the God will be happy and will make you a big impact.
↪ And he would definitely be able to melt away without getting wet! He spent all afternoon
↪ thinking about it. . He wished for a whole year to take off the hat and make a wish. The sun
↪ shone bright in the sky, and the little soul and his heart began to melt in the sun. Everyone
↪ in the family was so happy and they all celebrated with such a big hug. From then on, the
↪ little boy knew that the sun had made his wish come true! The bad sunshine helped the little
↪ boy become a happy, happy memory of his big heart. He knew that something magical had
↪ happened, but it was worth the heat. His wish had been successful! He had become an even
↪ wiser and happy success, and he knew that he could always count on a friend to help him. He'd
↪ love to share with his friends, to keep him in the future. And ever since then, he has been
↪ proud of himself and his friends. He knows that when we are in big trouble, we can have
↪ success in our own way. The little boy has been saving for something, and he was inspired by
↪ all. The End. The end. When he was done, everyone was a bit envious of his own little
↪ actions, and the little boy had learned a valuable lesson: it is better to find that
↪ friendship is more valuable than any reward. The end. The end. `No matter how old or poor the
↪ world is, if you don't always get to be happy. From then on, the world will always have more
↪ help if you need it. And it is true - like a reminder of the most beautiful friendship they
↪ had ever experienced. The world has a happy heart-great-gath, thanks to everyone, and your
↪ ability to smile. As a reminder of their friendship, from the most valuable experience, the
↪ littlest way of friendship. With that, the world will always remain happy and full of love
↪ and happiness. All the compassion of the world was filled with joy. The moral of the story is
↪ that no matter where you are, you can always find happiness and friendship in life. Be a
↪ friend of the best of friends and happiness will always appear
```

Comment on fluency: the output is partially coherent (it maintains a simple narrative about a journey), but it has clear issues: repetition of names ("Bill and Bill"), occasional punctuation/grammar glitches, and it does not develop a strong plot. Two important factors affecting quality are *decoding hyperparameters* (higher temperature / larger top-p increases diversity but can harm coherence; lower values make it more repetitive), and *stopping/handling of EOS* (this model strongly prefers emitting `<|endoftext|>` early; if EOS is allowed, generations often end immediately). Prompt specificity and checkpoint quality (validation loss) also strongly affect fluency.