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`): Implement the Transformer block (3 points)

implemented in `/cs336_basics/transformer.py`

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.