2023.10.19 - homework 1

October 19, 2023

0.0.1 2023.10.19 - Introduction to Transformers | Homework 1

In this exercise, you will implement your own character-based Tokenizer as well as an Embedding Layer from scratch. Base your code on the following skeleton code that we provide:

0.0.2 Exercise 1 - Character-based Tokenizer:

- Initialize your vocabulary with a list of unique characters. Consider alphabetic letters, common punctuation and numbers for a start. Your initial vocabulary should at least include lowercase English letters (a-z), digits (0-9), and common punctuation marks (e.g., ., !, ?).
- Implement a basic character-based tokenizer. Ensure to include a special $< UNK >$ ("unknown") token to handle characters outside your vocabulary.
  - The tokenizer should be capable of:
    - Parsing a string into a list of characters.
    - Encoding a list of characters into their corresponding indices in the vocabulary.
    - Decoding a list of indices back into a string.
    - When encoding, return the token ID for $< UNK >$ for any character not in the vocabulary. Similarly, when decoding, return the $< UNK >$ token for any unknown token ID.

[ ]:

```python
# Importing necessary dependencies
from typing import List
```

[ ]:

```python
vocab = list('') # define your vocabulary here
```

[ ]:

```python
class Tokenizer:
    def __init__(self, vocab: List[str]):
        # Add <UNK> token if it's not already in the vocabulary
        pass # Replace "pass" with your own code

    def parse(self, input: str) -> List[str]:
        """Convert a string to a list of characters."""
        pass # Replace "pass" with your own code

    def encode(self, tokens: List[str]) -> List[int]:
        """ Encode a list of tokens into their corresponding indices."""
        pass # Replace "pass" with your own code

    def decode(self, indices: List[int]) -> str:
        """Decode a list of indices back into a string."""
        pass # Replace "pass" with your own code
```

0.0.3 Run Exercise 1

Run this cell to evaluate your implementation.

[ ]:

```
"""
Expected Output:
============ Tokenizer
tokenizer.parse: ['c', 'a', 't', 'e', 'r', 'p', 'i', 'l', 'l', 'a', 'r', '!']
tokenizer.encode: [28, 26, 45, 30, 43, 41, 34, 37, 37, 26, 43, 53] # these␣
 ↪numbers will be different
tokenizer.decode: caterpillar!
tokenizer.encode/decode unknown: <UNK> # This will be different if you choose␣
 ↪to use a different <UNK> token
tokenizer.decode out of bounds: <UNK>  # This will be different if you choose␣
 ↪to use a different <UNK> token
"""

tokenizer = Tokenizer(vocab)

print("============ Tokenizer")
# Test parsing
tokens = tokenizer.parse('caterpillar!')
print(f"tokenizer.parse: {tokens}")

# Test encoding
token_ids = tokenizer.encode(tokens)
print(f"tokenizer.encode: {token_ids}")

# Test decoding
print(f"tokenizer.decode: {tokenizer.decode(token_ids)}")

# Test <UNK>
print(f"tokenizer.encode/decode unknown: {tokenizer.decode(tokenizer.
 ↪encode(['$']))}")
print(f"tokenizer.decode out of bounds: {tokenizer.decode([100])}\n")
```

0.0.4 Excercise 2 - Embedding Layer:

- Implement an embedding layer from scratch. This layer should be able to:
  - Initialize an embedding table with random values.
  - Look up and return embeddings for a given list of indices.
  - Handle potential out-of-bounds errors when looking up embeddings.

[ ]:

```
import torch
from torch import Tensor
```

[ ]:

```
class Embedding:
    def __init__(self, n_embd: int, d_embd: int):
        pass # Replace "pass" with your own code. You might use torch.randn

    def forward(self, input: Tensor) -> Tensor:
        """Perform a lookup for the given indices in the embedding table."""
        pass # Replace "pass" with your own code

    def __call__(self, input: Tensor) -> Tensor:
        # This function lets you call a class instance as a function e.g.␣
 ↪Embedding(n_embd, d_emdb)(x)
        # https://docs.python.org/3/reference/datamodel.html#object.__call__
        return self.forward(input)
```

0.0.5 Run Exercise 2

Run these cells to evaluate your implementation.

[ ]:

```
# A helper function to assert a function call throws an exception
def assert_raises(fn, *args, **kwargs):
    try:
        fn(*args, **kwargs)
    except Exception as e:
        print(f"Expected error occurred: {type(e).__name__} - {e}")
        return
    raise AssertionError("Expected error did not occur")
```

[ ]:

```
"""
Expected Output:

============ Embedding Layer
input (torch.Size([11])):
tensor([28, 26, 45, 30, 43, 41, 34, 37, 37, 26, 43]) # these numbers will be␣
 ↪different

embedding_layer result (torch.Size([11, 3])):  # these numbers will be␣
 ↪different, depending on your vocab size
tensor([[-0.0748,  0.5664, -0.6240], # all of the following numbers will be␣
 ↪different and also different per run
        [-1.9658, -0.7646, -0.4583],
        [ 1.1624,  0.8075, -0.5995],
        [ 0.4513, -0.0109,  0.2278],
        [-1.2602, -0.8705, -0.0846],
        [ 0.3563,  0.4905,  0.5740],
        [ 0.5596,  0.3183, -2.2232],
        [-0.2117, -0.0676,  1.6243],
        [-0.2117, -0.0676,  1.6243],
        [-1.9658, -0.7646, -0.4583],
        [-1.2602, -0.8705, -0.0846]])
Expected error occurred: ValueError - Input tensor contains invalid indices for lookup table. # The string will depend on which error you throw
"""
print("============ Embedding Layer")
n_embd = len(vocab)
d_embd = 3
embedding_layer = Embedding(n_embd, d_embd)

input_tensor = torch.tensor(tokenizer.encode(tokenizer.parse('caterpillar')))
result = embedding_layer(input_tensor)

print(f"input ({input_tensor.size()}):\n{input_tensor}\n")
print(f"embedding_layer result ({result.size()}):\n{result}")

# Assure layer throws exception on invalid index
assert_raises(embedding_layer, torch.tensor([n_embd]))
```