Assignment 5: Transformers and Natural Language Processing (v1)

Aspen Morgan

Collaborated with: Jamison Talley

Netid: 790907699

Note: this assignment falls under collaboration Mode 2: Individual Assignment – Collaboration Permitted. Please refer to the syllabus for additional information.

Attributions: Portions of this assignment are adapted from the Understanding Deep Learning textbook by Prince link here

Introduction

In this assignment you impelement some operations of transformers such as self-attention, and then investigate their use for natural language processing.

Problem 1: Implement Self-Attention (30 points) Below is an implementation of the self-attention process, which is utilized within transformers. Please fill in the missing portions of the code, where it says "TO DO". I have provided part of the correct solution at the bottom to help you determine if you are likely to be correct, but I encourage you to understand the operations well and check your code carefully to be sure! In some cases I have added comments to help explain the operations.

NOTE: you will primarily be graded upon the correctness of the output of the final tokenization routine that you need to build (i.e., the output of the last cell).

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

The self-attention mechanism maps N inputs $\mathbf{x}_n \in \mathbb{R}^D$ and returns N outputs $\mathbf{x}'_n \in \mathbb{R}^D$.

In [ ]:
```
# Set seed so we get the same random numbers
# DO NOT CHANGE THE RANDOM SEED!
np.random.seed(2)
# Number of inputs
N = 3
# Number of dimensions of each input
D = 4
```

```
# Create an empty list
all_x = []
# Create elements x_n and append to list
for n in range(N):
  all_x.append(np.random.normal(size=(D,1)))
# Print out the list
print(all_x)
```

```
[array([[-0.41675785],
       [-0.05626683],
       [-2.1361961 ],
       [ 1.64027081]]), array([[-1.79343559],
       [-0.84174737],
       [ 0.50288142],
       [-1.24528809]]), array([[-1.05795222],
       [-0.90900761],
       [ 0.55145404],
       [ 2.29220801]])]
```

We'll also need the weights and biases for the keys, queries, and values (equations 12.2 and 12.4)

In [ ]:
```
# Set seed so we get the same random numbers
# DO NOT CHANGE THE RANDOM SEED!
np.random.seed(1)

# Choose random values for the parameters
omega_q = np.random.normal(size=(D,D))
omega_k = np.random.normal(size=(D,D))
omega_v = np.random.normal(size=(D,D))
beta_q = np.random.normal(size=(D,1))
beta_k = np.random.normal(size=(D,1))
beta_v = np.random.normal(size=(D,1))
```

Now let's compute the queries, keys, and values for each input

In [ ]:
```
# Make three lists to store queries, keys, and values
all_queries = []
all_keys = []
all_values = []
# For every input
for x in all_x:
  query = np.dot(omega_q, x) + beta_q
  key = np.dot(omega_k, x) + beta_k
  value = np.dot(omega_v, x) + beta_v

  all_queries.append(query)
  all_keys.append(key)
  all_values.append(value)
```

We'll need a softmax function (equation 12.5) -- here, it will take a list of arbitrary numbers and return a list where the elements are non-negative and sum to one

In [ ]:
```
def softmax(items_in):
  return np.exp(items_in)/np.sum(np.exp(items_in));
```

Now compute the self attention values:

In [ ]:
```
# Create emptymlist for output
all_x_prime = []

# For each output
for n in range(N):
  # Create list for dot products of query N with all keys
  all_km_qn = []
  # Compute the dot products
  for key in all_keys:

    # compute the appropriate dot product
    dot_product = np.dot(np.transpose(key), all_queries[n])

    # Store dot product
    all_km_qn.append(dot_product)

  # Compute dot product
  attention = softmax(all_km_qn)
  # Print result (should be positive sum to one)
  print("Attentions for output ", n)
  print(attention)

  # Compute a weighted sum of all of the values according to the attention
  # (equation 12.3 in UDL text)
  x_prime = np.matmul(np.transpose(all_values), attention.flatten())
  all_x_prime.append(x_prime)


# Print out true values to check you have it correct
# Note that I have omitted two true values for the 0th solution;
#  you will be graded primarily upon correctness of these two entires
print("x_prime_0_calculated:", all_x_prime[0].transpose())
print("x_prime_0_true: [[ 1.95291253 4.36675277  ...   ...]]")
print("x_prime_1_calculated:", all_x_prime[1].transpose())
print("x_prime_1_true: [[ 2.46106441  2.64767206 -1.10557715 -1.1883621 ]]")
print("x_prime_2_calculated:", all_x_prime[2].transpose())
print("x_prime_2_true: [[ 1.94330266 4.45582258 4.74359057 6.28117285]]")
```

```
Attentions for output  0
[[[2.73836084e-02]]

 [[3.06126168e-08]]

 [[9.72616361e-01]]]
Attentions for output  1
[[[1.58139717e-04]]

 [[9.99837609e-01]]

 [[4.25104341e-06]]]
Attentions for output  2
[[[0.00993823]]

 [[0.00332099]]

 [[0.98674078]]]
x_prime_0_calculated: [[1.95291253]
 [4.36675277]
 [4.76351526]
 [6.26439746]]
x_prime_0_true: [[ 1.95291253 4.36675277  ...  ...]]
x_prime_1_calculated: [[ 2.46106441]
 [ 2.64767206]
 [-1.10557715]
 [-1.1883621 ]]
x_prime_1_true: [[ 2.46106441  2.64767206 -1.10557715 -1.1883621 ]]
x_prime_2_calculated: [[1.94330266]
 [4.45582258]
 [4.74359057]
 [6.28117285]]
x_prime_2_true: [[ 1.94330266 4.45582258 4.74359057 6.28117285]]
```

Problem 2 (30 points) In this problem we build a "vocabulary" from a text string, as in figure 12.8 of the UDL book, which can be used to tokenize text for input into a neural network (e.g., transformer). Tokenization is a step shared by nearly all forms of natural language processing (NLP), and this exercise will impart to you some familiarity with the process.

Work through the cells below, running each cell in turn. You will see "TO DO" in the comments in several blocks of code as you go; follow the instructions to complete the code in each of these instances. In some cases I provide detailed comments to help explain some of the operations.

NOTE: you will primarily be graded upon the correctness of the output of the final tokenization routine that you need to build (i.e., the output of the last cell).

In [ ]:
```
import re, collections
```

In [ ]:
```
text = "a sailor went to sea sea sea "+\
                  "to see what he could see see see "+\
```

```
                   "but all that he could see see see "+\
                   "was the bottom of the deep blue sea sea sea"
```

Tokenize the input sentence To begin with the tokens are the individual letters and the whitespace token. So, we represent each word in terms of these tokens with spaces between the tokens to delineate them.

The tokenized text is stored in a structure that represents each word as tokens together with the count of how often that word occurs. We'll call this the *vocabulary*.

In [ ]:
```
def initialize_vocabulary(text):
  # Create a dictionary with keys that are text, and values that are numbere
  # The keys will be our words, and the values will be how many instances of
  # that word that we find.
  vocab = collections.defaultdict(int)

  # Automatically split the text into a list of words, with all extra white
  # space removed
  words = text.strip().split()

  # Add each individual word to the dictionary, and increment count each tim
  # we encounter that word.  We add white space to the individual letters
  # so that later we can apply split again and get the individual letters to
  # create tokens
  for word in words:
      vocab[' '.join(list(word)) + ' _'] += 1
  return vocab
```

In [ ]:
```
vocab = initialize_vocabulary(text)
print('Vocabulary: {}'.format(vocab))
print('Size of vocabulary: {}'.format(len(vocab)))
```

```
Vocabulary: defaultdict(<class 'int'>, {'a _': 1, 's a i l o r _': 1, 'w e n
t _': 1, 't o _': 2, 's e a _': 6, 's e e _': 7, 'w h a t _': 1, 'h e _': 2,
'c o u l d _': 2, 'b u t _': 1, 'a l l _': 1, 't h a t _': 1, 'w a s _': 1,
't h e _': 2, 'b o t t o m _': 1, 'o f _': 1, 'd e e p _': 1, 'b l u e _':
1})
Size of vocabulary: 18
```

Find all the tokens in the current vocabulary and their frequencies

In [ ]:
```
def get_tokens_and_frequencies(vocab):
  tokens = collections.defaultdict(int)
  for word, freq in vocab.items():
      word_tokens = word.split()
      for token in word_tokens:
          # count characters by getting current value of token in dict and a
          tokens[token] = tokens[token] + freq
  return tokens
```

In [ ]:
```
tokens = get_tokens_and_frequencies(vocab)
print('Tokens: {}'.format(tokens))
print('Number of tokens: {}'.format(len(tokens)))
```

```
Tokens: defaultdict(<class 'int'>, {'a': 12, '_': 33, 's': 15, 'i': 1, 'l':
6, 'o': 8, 'r': 1, 'w': 3, 'e': 28, 'n': 1, 't': 11, 'h': 6, 'c': 2, 'u': 4,
'd': 3, 'b': 3, 'm': 1, 'f': 1, 'p': 1})
Number of tokens: 19
```

Find each pair of adjacent tokens in the vocabulary and count them. We will subsequently merge the most frequently occurring pair.

In [ ]:
```
def get_pairs_and_counts(vocab):
    pairs = collections.defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
            # get count for each pair of symbols by getting current value ar
            pairs[symbols[i] + ' ' + symbols[i+1]] = pairs[symbols[i] + ' '
    return pairs
```

In [ ]:
```
pairs = get_pairs_and_counts(vocab)
print('Pairs: {}'.format(pairs))
print('Number of distinct pairs: {}'.format(len(pairs)))

most_frequent_pair = max(pairs, key=pairs.get)
print('Most frequent pair: {}'.format(most_frequent_pair))
```

```
Pairs: defaultdict(<class 'int'>, {'a _': 7, 's a': 1, 'a i': 1, 'i l': 1,
'l o': 1, 'o r': 1, 'r _': 1, 'w e': 1, 'e n': 1, 'n t': 1, 't _': 4, 't o':
3, 'o _': 2, 's e': 13, 'e a': 6, 'e e': 8, 'e _': 12, 'w h': 1, 'h a': 2,
'a t': 2, 'h e': 4, 'c o': 2, 'o u': 2, 'u l': 2, 'l d': 2, 'd _': 2, 'b u':
1, 'u t': 1, 'a l': 1, 'l l': 1, 'l _': 1, 't h': 3, 'w a': 1, 'a s': 1, 's
_': 1, 'b o': 1, 'o t': 1, 't t': 1, 'o m': 1, 'm _': 1, 'o f': 1, 'f _': 1,
'd e': 1, 'e p': 1, 'p _': 1, 'b l': 1, 'l u': 1, 'u e': 1})
Number of distinct pairs: 48
Most frequent pair: s e
```

Merge the instances of the best pair in the vocabulary

Updated merge_pair_in_vocabulary and white space character because </w> is unnecessarily long and the merge_pair_in_vocabulary function does not work once characters are lumped together (e.g. old function can process 's e e *' but not 'se e*'.)

In [ ]:
```
# def merge_pair_in_vocabulary(pair, vocab_in):
#     vocab_out = {}
#     bigram = re.escape(' '.join(pair))
#     p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
#     for word in vocab_in:
#         word_out = p.sub(''.join(pair), word)
#         vocab_out[word_out] = vocab_in[word]
#     return vocab_out
```

In [ ]:
```
def merge_pair_in_vocabulary(pair, vocab_in):
    vocab_out = {}
    char1, char2 = pair.split(' ') # pair has space
    sandwich = char1 + char2 # sandwich does not
```

```
    for word in vocab_in:
      word_out = word.replace(pair, sandwich)
      vocab_out[word_out] = vocab_in[word]
    return vocab_out
```

In [ ]:
```
vocab = merge_pair_in_vocabulary(most_frequent_pair, vocab)
print('Vocabulary: {}'.format(vocab))
print('Size of vocabulary: {}'.format(len(vocab)))
```

```
Vocabulary: {'a _': 1, 's a i l o r _': 1, 'w e n t _': 1, 't o _': 2, 'se a _': 6, 'se e _': 7, 'w h a t _': 1, 'h e _': 2, 'c o u l d _': 2, 'b u t _': 1, 'a l l _': 1, 't h a t _': 1, 'w a s _': 1, 't h e _': 2, 'b o t t o m _': 1, 'o f _': 1, 'd e e p _': 1, 'b l u e _': 1}
Size of vocabulary: 18
```

Update the tokens, which now include the best token 'se'

In [ ]:
```
tokens = get_tokens_and_frequencies(vocab)
print('Tokens: {}'.format(tokens))
print('Number of tokens: {}'.format(len(tokens)))
```

```
Tokens: defaultdict(<class 'int'>, {'a': 12, '_': 33, 's': 2, 'i': 1, 'l': 6, 'o': 8, 'r': 1, 'w': 3, 'e': 15, 'n': 1, 't': 11, 'se': 13, 'h': 6, 'c': 2, 'u': 4, 'd': 3, 'b': 3, 'm': 1, 'f': 1, 'p': 1})
Number of tokens: 20
```

Now let's write the full tokenization routine

In [ ]:
```
def tokenize(text, num_merges):
  # Initialize the vocabulary from the input text
  vocab = initialize_vocabulary(text)

  for i in range(num_merges):
    # Find the tokens and how often they occur in the vocabulary
    tokens = get_tokens_and_frequencies(vocab)

    # Find the pairs of adjacent tokens and their counts
    pairs = get_pairs_and_counts(vocab)

    # Find the most frequent pair
    most_frequent_pair = max(pairs, key=pairs.get)
    print('Most frequent pair: {}'.format(most_frequent_pair))

    # Merge the code in the vocabulary
    vocab = merge_pair_in_vocabulary(most_frequent_pair, vocab)

  # Find the tokens and how often they occur in the vocabulary one last time
  tokens = get_tokens_and_frequencies(vocab)

  return tokens, vocab
```

In [ ]:
```
tokens, vocab = tokenize(text, num_merges=22)
```

```
Most frequent pair: s e
Most frequent pair: e _
Most frequent pair: a _
Most frequent pair: se e_
Most frequent pair: se a_
Most frequent pair: t _
Most frequent pair: h e_
Most frequent pair: t o
Most frequent pair: to _
Most frequent pair: h a
Most frequent pair: ha t_
Most frequent pair: c o
Most frequent pair: co u
Most frequent pair: cou l
Most frequent pair: coul d
Most frequent pair: could _
Most frequent pair: t he_
Most frequent pair: s a
Most frequent pair: sa i
Most frequent pair: sai l
Most frequent pair: sail o
Most frequent pair: sailo r
```

In [ ]:

```
print('Tokens: {}'.format(tokens))
print('Number of tokens: {}'.format(len(tokens)))
print('Vocabulary: {}'.format(vocab))
print('Size of vocabulary: {}'.format(len(vocab)))
```

```
Tokens: defaultdict(<class 'int'>, {'a_': 1, 'sailor': 1, '_': 6, 'w': 3,
'e': 3, 'n': 1, 't_': 2, 'to_': 2, 'sea_': 6, 'see_': 7, 'hat_': 2, 'he_':
2, 'could_': 2, 'b': 3, 'u': 2, 'a': 2, 'l': 3, 't': 2, 's': 1, 'the_': 2,
'o': 2, 'to': 1, 'm': 1, 'f': 1, 'd': 1, 'p': 1, 'e_': 1})
Number of tokens: 27
Vocabulary: {'a_': 1, 'sailor _': 1, 'w e n t_': 1, 'to_': 2, 'sea_': 6, 'se
e_': 7, 'w hat_': 1, 'he_': 2, 'could_': 2, 'b u t_': 1, 'a l l _': 1, 't ha
t_': 1, 'w a s _': 1, 'the_': 2, 'b o t to m _': 1, 'o f _': 1, 'd e e p _':
1, 'b l u e_': 1}
Size of vocabulary: 18
```