Embedding_plus_Positional_encoding

October 31, 2025

1 Transformer Pre-processing

Welcome to Week 4's first ungraded lab. In this notebook you will delve into the pre-processing methods you apply to raw text to before passing it to the encoder and decoder blocks of the transformer architecture.

After this assignment you'll be able to:

- Create visualizations to gain intuition on positional encodings
- Visualize how positional encodings affect word embeddings

1.1 Table of Contents

## Packages

Run the following cell to load the packages you'll need.

```
[ ]: import tensorflow as tf
     import numpy as np
     import matplotlib.pyplot as plt
     import os

     from tensorflow.keras.layers import Embedding
     from tensorflow.keras.preprocessing.text import Tokenizer
     from tensorflow.keras.preprocessing.sequence import pad_sequences
```

## 1 - Positional Encoding

Here are the positional encoding equations that you implemented in the previous assignment. This encoding uses the following formulas:

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$
$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

It is a standard practice in natural language processing tasks to convert sentences into tokens before feeding texts into a language model. Each token is then converted into a numerical vector of fixed length called an embedding, which captures the meaning of the words. In the Transformer architecture, a positional encoding vector is added to the embedding to pass positional information throughout the model.

The meaning of these vectors can be difficult to grasp solely by examining the numerical representations, but visualizations can help give some intuition as to the semantic and positional similarity of the words. As you've seen in previous assignments, when embeddings are reduced to two dimensions and plotted, semantically similar words appear closer together, while dissimilar words are plotted farther apart. A similar exercise can be performed with positional encoding vectors - words that are closer in a sentence should appear closer when plotted on a Cartesian plane, and when farther in a sentence, should appear farther on the plane.

In this notebook, you will create a series of visualizations of word embeddings and positional encoding vectors to gain intuition into how positional encodings affect word embeddings and help transport sequential information through the Transformer architecture.

### 1.1 - Positional encoding visualizations

The following code cell has the `positional_encoding` function which you implemented in the Transformer assignment. Nice work! You will build off that work to create some more visualizations with this function in this notebook.

```
[ ]: def positional_encoding(positions, d):
         """
         Precomputes a matrix with all the positional encodings

         Arguments:
             positions (int) -- Maximum number of positions to be encoded
             d (int) -- Encoding size

         Returns:
             pos_encoding -- (1, position, d_model) A matrix with the positional␣
      ↪encodings
         """

         # initialize a matrix angle_rads of all the angles
         angle_rads = np.arange(positions)[:, np.newaxis] / np.power(10000, (2 * (np.
      ↪arange(d)[np.newaxis, :]//2)) / np.float32(d))
         angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
         angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])

         pos_encoding = angle_rads[np.newaxis, ...]
```

```
    return tf.cast(pos_encoding, dtype=tf.float32)
```

Define the embedding dimension as 100. This value must match the dimensionality of the word embedding. In the "Attention is All You Need" paper, embedding sizes range from 100 to 1024, depending on the task. The authors also use a maximum sequence length ranging from 40 to 512 depending on the task. Define the maximum sequence length to be 100, and the maximum number of words to be 64.

[ ]:
```
EMBEDDING_DIM = 100
MAX_SEQUENCE_LENGTH = 100
MAX_NB_WORDS = 64
pos_encoding = positional_encoding(MAX_SEQUENCE_LENGTH, EMBEDDING_DIM)

plt.pcolormesh(pos_encoding[0], cmap='RdBu')
plt.xlabel('d')
plt.xlim((0, EMBEDDING_DIM))
plt.ylabel('Position')
plt.colorbar()
plt.show()
```

You have already created this visualization in this assignment, but let us dive a little deeper. Notice some interesting properties of the matrix - the first is that the norm of each of the vectors is always a constant. No matter what the value of `pos` is, the norm will always be the same value, which in this case is 7.071068. From this property you can conclude that the dot product of two positional encoding vectors is not affected by the scale of the vector, which has important implications for correlation calculations.

[ ]:
```
pos = 34
tf.norm(pos_encoding[0,pos,:])
```

Another interesting property is that the norm of the difference between 2 vectors separated by `k` positions is also constant. If you keep `k` constant and change `pos`, the difference will be of approximately the same value. This property is important because it demonstrates that the difference does not depend on the positions of each encoding, but rather the relative seperation between encodings. Being able to express positional encodings as linear functions of one another can help the model to learn by focusing on the relative positions of words.

This reflection of the difference in the positions of words with vector encodings is difficult to achieve, especially given that the values of the vector encodings must remain small enough so that they do not distort the word embeddings.

[ ]:
```
pos = 70
k = 2
print(tf.norm(pos_encoding[0,pos,:] -  pos_encoding[0,pos + k,:]))
```

You have observed some interesting properties about the positional encoding vectors - next, you will create some visualizations to see how these properties affect the relationships between encodings and embeddings!

### 1.2 - Comparing positional encodings

1.2.1 - Correlation The positional encoding matrix help to visualize how each vector is unique for every position. However, it is still not clear how these vectors can represent the relative position of the words in a sentence. To illustrate this, you will calculate the correlation between pairs of vectors at every single position. A successful positional encoder will produce a perfectly symmetric matrix in which maximum values are located at the main diagonal - vectors in similar positions should have the highest correlation. Following the same logic, the correlation values should get smaller as they move away from the main diagonal.

```
[ ]: # Positional encoding correlation
     corr = tf.matmul(pos_encoding, pos_encoding, transpose_b=True).numpy()[0]
     plt.pcolormesh(corr, cmap='RdBu')
     plt.xlabel('Position')
     plt.xlim((0, MAX_SEQUENCE_LENGTH))
     plt.ylabel('Position')
     plt.colorbar()
     plt.show()
```

1.2.2 - Euclidean distance You can also use the euclidean distance instead of the correlation for comparing the positional encoding vectors. In this case, your visualization will display a matrix in which the main diagonal is 0, and its off-diagonal values increase as they move away from the main diagonal.

```
[ ]: # Positional encoding euclidean distance
     eu = np.zeros((MAX_SEQUENCE_LENGTH, MAX_SEQUENCE_LENGTH))
     print(eu.shape)
     for a in range(MAX_SEQUENCE_LENGTH):
         for b in range(a + 1, MAX_SEQUENCE_LENGTH):
             eu[a, b] = tf.norm(tf.math.subtract(pos_encoding[0, a], pos_encoding[0,␣
      ↪b]))
             eu[b, a] = eu[a, b]

     plt.pcolormesh(eu, cmap='RdBu')
     plt.xlabel('Position')
     plt.xlim((0, MAX_SEQUENCE_LENGTH))
     plt.ylabel('Position')
     plt.colorbar()
     plt.show()
```

Nice work! You can use these visualizations as checks for any positional encodings you create.

## 2 - Semantic embedding

You have gained insight into the relationship positional encoding vectors have with other vectors at different positions by creating correlation and distance matrices. Similarly, you can gain a stronger intuition as to how positional encodings affect word embeddings by visualizing the sum of these vectors.

### 2.1 - Load pretrained embedding

To combine a pretrained word embedding with the positional encodings you created, start by loading one of the pretrained embeddings from the glove project. You will use the embedding with 100 features.

[ ]:
```
embeddings_index = {}
GLOVE_DIR = "glove"
f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
print('d_model:', embeddings_index['hi'].shape)
```

Note: This embedding is composed of 400,000 words and each word embedding has 100 features.

Consider the following text that only contains two sentences. Wait a minute - these sentences have no meaning! Instead, the sentences are engineered such that: * Each sentence is composed of sets of words, which have some semantic similarities among each groups. * In the first sentence similar terms are consecutive, while in the second sentence, the order is random.

[ ]:
```
texts = ['king queen man woman dog wolf football basketball red green yellow',
         'man queen yellow basketball green dog  woman football  king red wolf']
```

First, run the following code cell to apply the tokenization to the raw text. Don't worry too much about what this step does - it will be explained in detail in later ungraded labs. A quick summary (not crucial to understanding the lab):

- If you feed an array of plain text of different sentence lengths, and it will produce a matrix with one row for each sentence, each of them represented by an array of size `MAX_SEQUENCE_LENGTH`.
- Each value in this array represents each word of the sentence using its corresponding index in a dictionary(`word_index`).
- The sequences shorter than the `MAX_SEQUENCE_LENGTH` are padded with zeros to create uniform length.

Again, this is explained in detail in later ungraded labs, so don't worry about this too much right now!

[ ]:
```
tokenizer = Tokenizer(num_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
```

```
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, padding='post', maxlen=MAX_SEQUENCE_LENGTH)

print(data.shape)

print(data)
```

To simplify your model, you will only need to obtain the embeddings for the different words that appear in the text you are examining. In this case, you will filter out only the 11 words appearing in our sentences. The first vector will be an array of zeros and will codify all the unknown words.

[ ]:
```
embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
print(embedding_matrix.shape)
```

Create an embedding layer using the weights extracted from the pretrained glove embeddings.

[ ]:
```
embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            embeddings_initializer=tf.keras.initializers.
 ↪Constant(embedding_matrix),
                            trainable=False)
```

Transform the input tokenized data to the embedding using the previous layer. Check the shape of the embedding to make sure the last dimension of this matrix contains the embeddings of the words in the sentence.

[ ]:
```
embedding = embedding_layer(data)
print(embedding.shape)
```

### 2.2 - Visualization on a Cartesian plane

Now, you will create a function that allows you to visualize the encoding of our words in a Cartesian plane. You will use PCA to reduce the 100 features of the glove embedding to only 2 components.

[ ]:
```
from sklearn.decomposition import PCA

def plot_words(embedding, sequences, sentence):
    pca = PCA(n_components=2)
    X_pca_train = pca.fit_transform(embedding[sentence,0:
 ↪len(sequences[sentence]),:])



    fig, ax = plt.subplots(figsize=(12, 6))
    plt.rcParams['font.size'] = '12'
    ax.scatter(X_pca_train[:, 0], X_pca_train[:, 1])
    words = list(word_index.keys())
    for i, index in enumerate(sequences[sentence]):
        ax.annotate(words[index-1], (X_pca_train[i, 0], X_pca_train[i, 1]))
```

Nice! Now you can plot the embedding of each of the sentences. Each plot should disply the embeddings of the different words.

[ ]:
```
plot_words(embedding, sequences, 0)
```

Plot the word of embeddings of the second sentence. Recall that the second sentence contains the same words are the first sentence, just in a different order. You can see that the order of the words does not affect the vector representations.

[ ]:
```
plot_words(embedding, sequences, 1)
```

## 3 - Semantic and positional embedding

Next, you will combine the original glove embedding with the positional encoding you calculated earlier. For this exercise, you will use a 1 to 1 weight ratio between the semantic and the positional embedding.

[ ]:
```
embedding2 = embedding * 1.0 + pos_encoding[:,:,:] * 1.0

plot_words(embedding2, sequences, 0)
plot_words(embedding2, sequences, 1)
```

Wow look at the big difference between the plots! Both plots have changed drastically compared to their original counterparts. Notice that in the second image, which corresponds to the sentence in which similar words are not together, very dissimilar words such as `red` and `wolf` appear more close.

Now you can try different relative weights and see how this strongly impacts the vector representation of the words in the sentence.

[ ]:
```
W1 = 1 # Change me
W2 = 10 # Change me
embedding2 = embedding * W1 + pos_encoding[:,:,:] * W2
plot_words(embedding2, sequences, 0)
plot_words(embedding2, sequences, 1)

# For reference
#['king queen man woman dog wolf football basketball red green yellow',
# 'man queen yellow basketball green dog  woman football  king red wolf']
```

If you set `W1 = 1` and `W2 = 10`, you can see how arrangement of the words begins to take on a clockwise or anti-clockwise order depending on the position of the words in the sentence. Under these parameters, the positional encoding vectors have dominated the embedding.

Now try inverting the weights to `W1 = 10` and `W2 = 1`. Observe that under these parameters, the plot resembles the original embedding visualizations and there are only a few changes between the positions of the plotted words.

In the previous Transformer assignment, the word embedding is multiplied by `sqrt(EMBEDDING_DIM)`. In this case, it will be equivalent using `W1 = sqrt(EMBEDDING_DIM) = 10` and `W2 = 1`.

1.1.1 Congratulations!

You've completed this notebook, and have a better sense of the inputs of the Transformer network!

By now, you've:

- Created positional encoding matrices to visualize the relational properties of the vectors
- Plotted embeddings and positional encodings on a Cartesian plane to observe how they affect each other

What you should remember: - Positional encodings can be expressed as linear functions of each other, which allow the model to learn according to the relative positions of words. - Positional encodings can affect the word embeddings, but if the relative weight of the positional encoding is small, the sum will retain the semantic meaning of the words.

```
[ ]:
```