The Embedding Layer

Hunter Phillips · Following

11 min read · May 9

This article is the first in The Implemented Transformer series. It introduces embeddings on a small-scale to build intuition. This is followed by the transformers usage of the embedding layer.

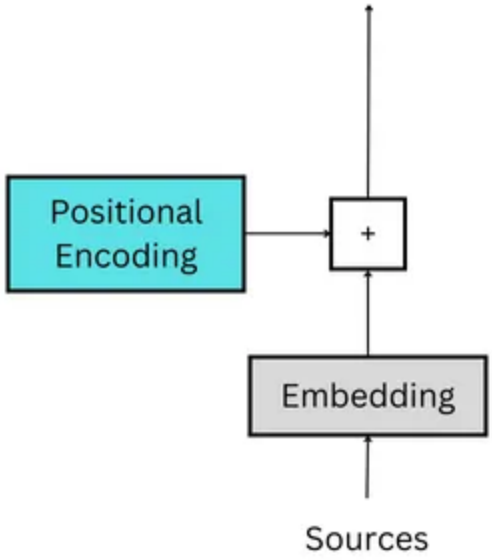


Image by Author

Background

The goal of an embedding layer is to enable a model to learn more about the relationships between words, tokens, or other inputs. This embedding layer can be viewed as transforming data from a higher-dimension space to a lower-dimension space, or it could be viewed as mapping data from a lower-dimension space to a higher-dimension space.

From One-Hot Vectors to Embedding Vectors

In natural language processing, tokens are derived from a corpus of data that may contain chapters, paragraphs, or sentences. These are broken into smaller pieces in various ways, but the most common tokenization method

is by word. All of the unique words from the corpus are known as the vocabulary.

Each word in the vocabulary is assigned an integer since it is easier for computers to process. There are various ways to assign these integers, but once again, the simplest method is to assign them alphabetically.

The image below demonstrates this process of breaking down a larger corpus into its components and assigning integers to each. Please note that the punctuation was stripped, and the text was set to lowercase for simplicity.

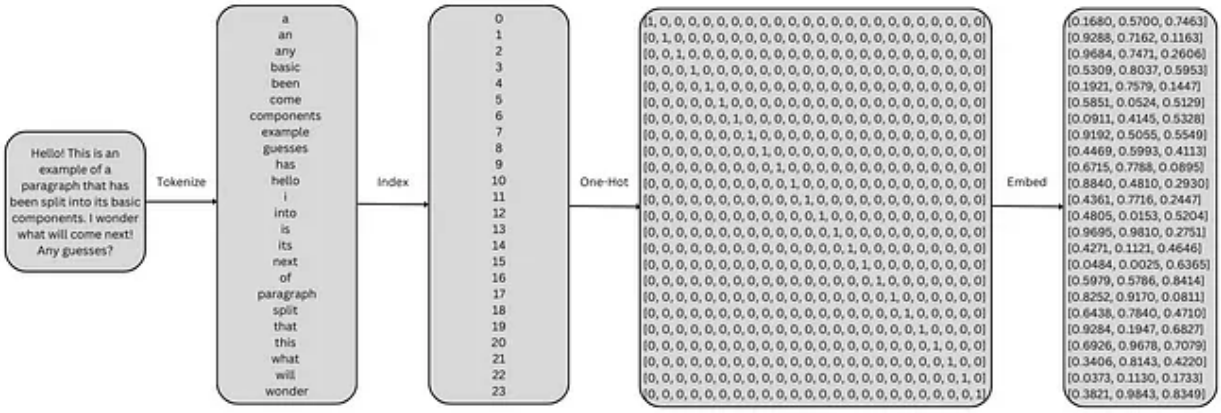


Image by Author

The numerical ordering created by assigning each word an index implies a relationship. Since this is not the intent, the indices are often used to create a one-hot encoded vector for each word. A one-hot vector has the same length as the vocabulary. In this case, each vector has 24-elements. It is called a “one-hot” vector because only one element is “turned on” or set to 1; all other tokens are “off” or set to 0. The index of the 1 corresponds to the

integer value assigned to the word. Typically, a model learns to predict the highest probability for a given index in the vector.

One-hot encoded vectors are often a convenient representation when there is only a dozen tokens or classes for a model to predict. However, a large corpus can have hundreds of thousands of tokens. Instead of using sparse vectors full of zeros that do not convey much meaning, an embedding layer is used to map the vectors to smaller dimensions. These embedded vectors can be trained to convey more information about each word and its relationship to other words.

Essentially, each word is represented by a ***d_model***-dimensional vector, where ***d_model*** can be any number. It simply indicates the number of embedding dimensions. If ***d_model*** is 2 or 3, then it is possible to visualize the relationship between each word, but it common to use values of 256, 512, and 1024 depending on the task.

An example of optimized embeddings can be seen below, where books of similar genres are embedded near each other:

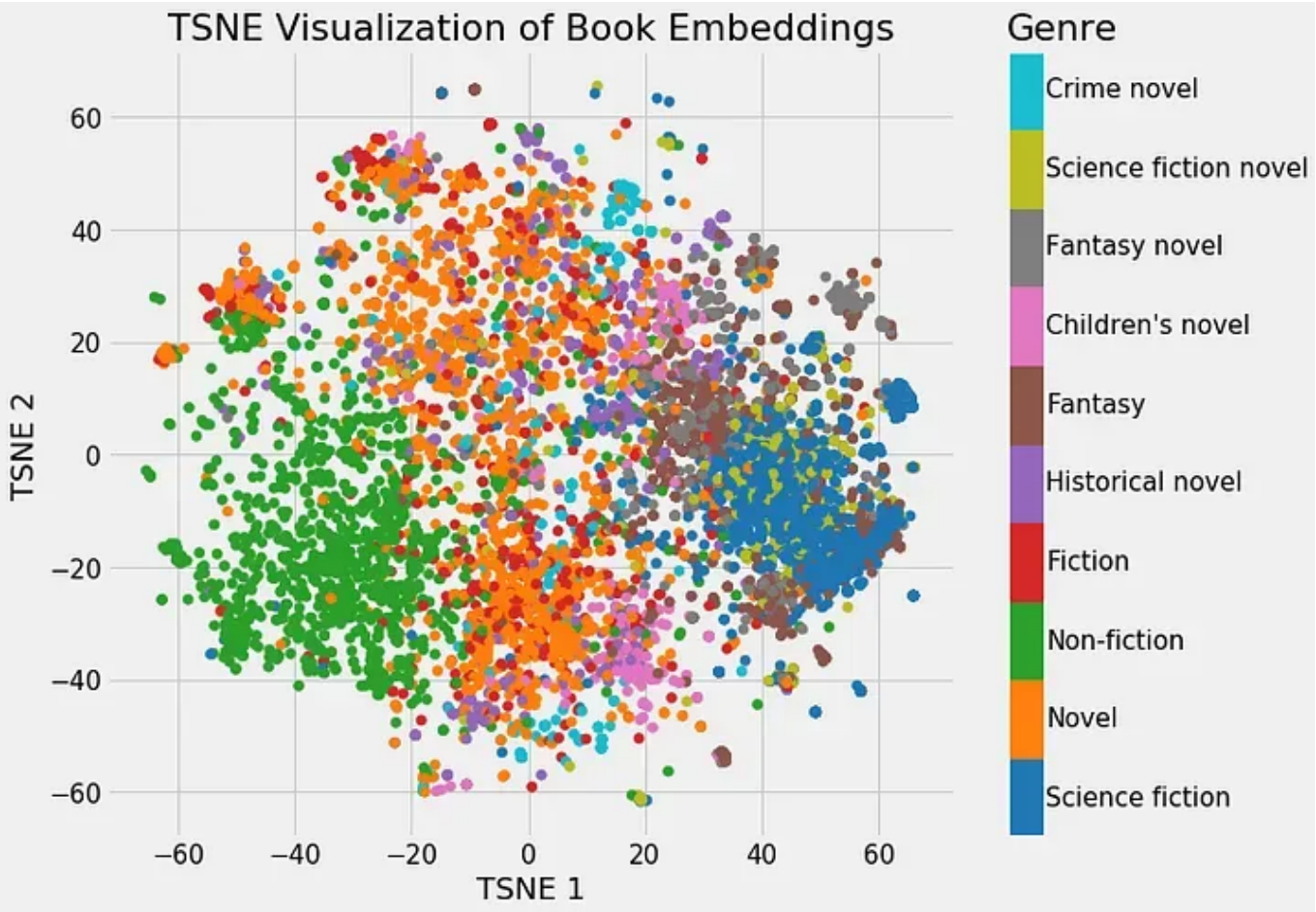


Image by Will Koehrsen

Embedding Vectors

The embedding matrix has a size of *(vocab_size, d_model)*. This allows for a matrix of one-hot vectors with a size of *(seq_length, vocab_size)* to be multiplied against it to acquire a new embedded representation. The sequence length is represented by *seq_length*, which is the number of tokens in a sequence. Keep in mind that the “sequence” in the visualizations thus far have been the entire vocabulary. In practice, a subset of the vocabulary would be used, such as `”a basic paragraph”`. This sequence would be tokenized, indexed, and converted to a matrix of one-hot encoded vectors. These one-hot encoded vectors would then be able to be multiplied against the embedding matrix.

An embedded sequence would have a size of *(seq_length, vocab_size)* **x** *(vocab_size, d_model)* = *(seq_length, d_model)*. This means each word in a sentence is now represented by a *d_model*-dimensional vector instead of a *vocab_size*-element one-hot encoded vector. An example of this matrix multiplication can be seen below. The indexed sequence has a shape of *(3,24)*, and the embedding matrix has a shape of *(24, 3)*. Once they are multiplied, the output is a *(3,3)* matrix. Each word is represented by its 3-element embedding vector.

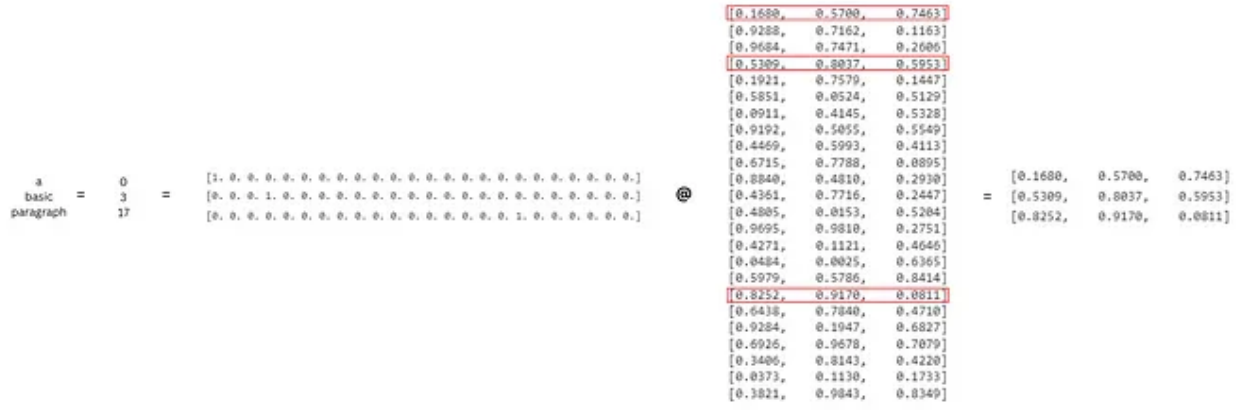


Image by Author

When a one-hot encoded matrix is multiplied with an embedding layer, the corresponding vectors of the embedding layer are returned without any changes. Below is matrix multiplication between the entire vocabulary of one-hot encoded vectors and the embedding matrix. The output is the embedding matrix.

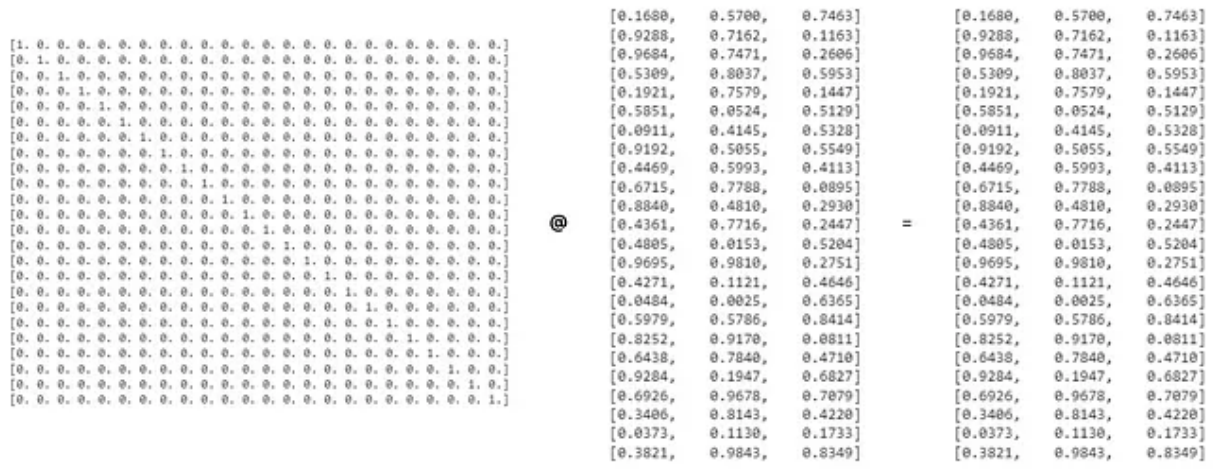


Image by Author

This indicates there is an easier way to acquire these same values without using matrix multiplication, which can be resource intensive. Instead of going from a one-hot encoded vector to an ***d_model***-dimensional embedding, which is from a larger dimension to a smaller dimension, the integer assigned to each word can be used to directly index the embedding matrix. This is like going from one-dimension to ***d_model***-dimensions that provide more information about the token.

The diagram below shows how the exact same result is obtained without multiplication:

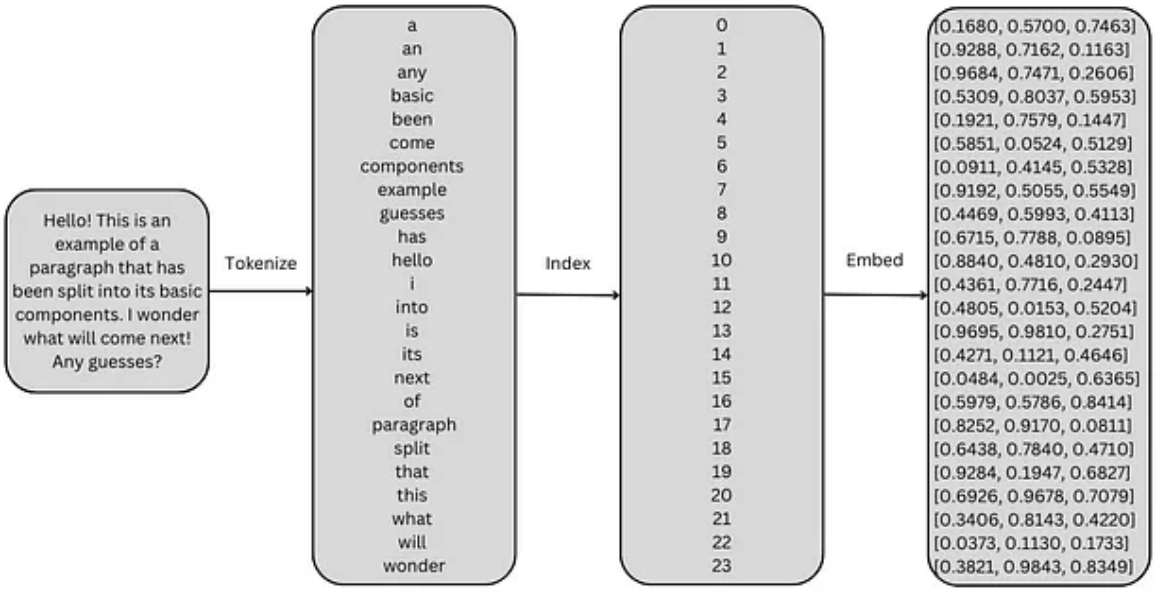


Image by Author

Embeddings from Scratch

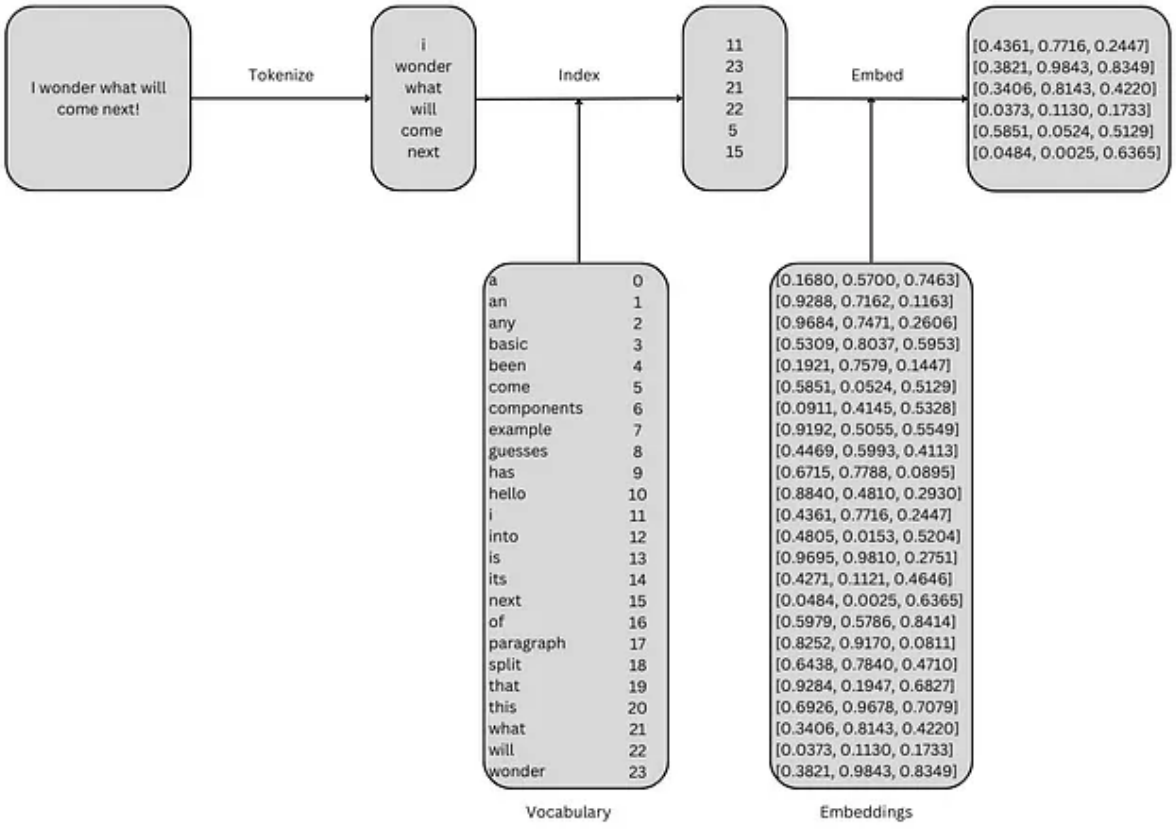


Image by Author

A simple implementation of the above diagram can be created in Python. Embedding a sequence requires a tokenizer, a vocabulary of words and their indices, and a three-dimensional embedding for each word in the vocabulary. A tokenizer splits a sequence into its tokens, which are lowercase words in this example. The simple function below removes punctuation from the sequence, splits it into its tokens, and lowercases them.

```
# importing required libraries
import math
import copy
import numpy as np
```

```
# torch packages
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import Tensor

# visualization packages
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt

example = "Hello! This is an example of a paragraph that has been split into its

def tokenize(sequence):
  # remove punctuation
  for punc in ["!", ".", "?"]:
    sequence = sequence.replace(punc, "")

  # split the sequence on spaces and lowercase each token
  return [token.lower() for token in sequence.split(" ")]

tokenize(example)
```

```
['hello', 'this', 'is', 'an', 'example', 'of', 'a', 'paragraph', 'that',
'has', 'been', 'split', 'into', 'its', 'basic', 'components', 'i',
'wonder', 'what', 'will', 'come', 'next', 'any', 'guesses']
```

With the tokenizer created, the vocabulary can be created for the example. The vocabulary contains the unique list of words that make up the data. While there are not duplicates in the example, they should still be removed. A simple example would be the following sentence: "***i am cool because i am short.***" The vocabulary would be "***i, am, cool, because, short***". These words would then be placed in alphabetical order: "***am, because, cool, i, short***". Finally, they would each be assigned an integer: "***am: 0, because: 1, cool: 2, i: 3, short: 4***". This process is implemented in the function below.

```
def build_vocab(data):
  # tokenize the data and remove duplicates
  vocab = list(set(tokenize(data)))

  # sort the vocabulary
  vocab.sort()

  # assign an integer to each word
  stoi = {word:i for i, word in enumerate(vocab)}

  return stoi

# build the vocab
stoi = build_vocab(example)

stoi
```

```
{'a': 0,
 'an': 1,
 'any': 2,
 'basic': 3,
 'been': 4,
 'come': 5,
 'components': 6,
 'example': 7,
 'guesses': 8,
 'has': 9,
 'hello': 10,
 'i': 11,
 'into': 12,
 'is': 13,
 'its': 14,
 'next': 15,
 'of': 16,
 'paragraph': 17,
 'split': 18,
 'that': 19,
 'this': 20,
 'what': 21,
 'will': 22,
 'wonder': 23}
```

This vocabulary can now be used to convert any sequence of tokens into its integer representation.

```
sequence = [stoi[word] for word in tokenize("I wonder what will come next!")]
sequence
```

```
[11, 23, 21, 22, 5, 15]
```

The next step is to create the embedding layer, which is nothing more than a matrix of random values with a size of *(vocab_size, d_model)*. These values can be generated using *torch.rand*.

```
# vocab size
vocab_size = len(stoi)

# embedding dimensions
d_model = 3

# generate the embedding layer
embeddings = torch.rand(vocab_size, d_model) # matrix of size (24, 3)
embeddings
```

```
tensor([[0.7629, 0.1146, 0.1228],
        [0.3628, 0.5717, 0.0095],
        [0.0256, 0.1148, 0.1023],
        [0.4993, 0.9580, 0.1113],
        [0.9696, 0.7463, 0.3762],
        [0.5697, 0.5022, 0.9080],
        [0.2689, 0.6162, 0.6816],
        [0.3899, 0.2993, 0.4746],
        [0.1197, 0.1217, 0.6917],
```

```
        [0.8282, 0.8638, 0.4286],
        [0.2029, 0.4938, 0.5037],
        [0.7110, 0.5633, 0.6537],
        [0.5508, 0.4678, 0.0812],
        [0.6104, 0.4849, 0.2318],
        [0.7710, 0.8821, 0.3744],
        [0.6914, 0.9462, 0.6869],
        [0.5444, 0.0155, 0.7039],
        [0.9441, 0.8959, 0.8529],
        [0.6763, 0.5171, 0.9406],
        [0.1294, 0.6113, 0.5955],
        [0.3806, 0.7946, 0.3526],
        [0.2259, 0.4360, 0.6901],
        [0.6300, 0.2691, 0.9785],
        [0.2094, 0.9159, 0.7973]])
```

With the embeddings created, the indexed sequence can be used to select the appropriate embedding for each token. The original sequence has a shape of *(6,)* and values of *[11, 23, 21, 22, 5, 15]*.

```
# embed the sequence
embedded_sequence = embeddings[sequence]

embedded_sequence
```

```
tensor([[0.7110, 0.5633, 0.6537],
        [0.2094, 0.9159, 0.7973],
        [0.2259, 0.4360, 0.6901],
        [0.6300, 0.2691, 0.9785],
        [0.5697, 0.5022, 0.9080],
        [0.6914, 0.9462, 0.6869]])
```

Now, each of the six tokens is replaced by a 3-element vector; the new shape is *(6, 3)*.

Since each of these tokens has three components, they can be mapped in three dimensions. While this plot shows an untrained embedding matrix, a trained one would map similar words near each other like the aforementioned book example.

```
# visualize the embeddings in 3 dimensions
x, y, z = embedded_sequences[:, 0], embedded_sequences[:, 1], embedded_sequences
ax = plt.axes(projection='3d')
ax.scatter3D(x, y, z)
```

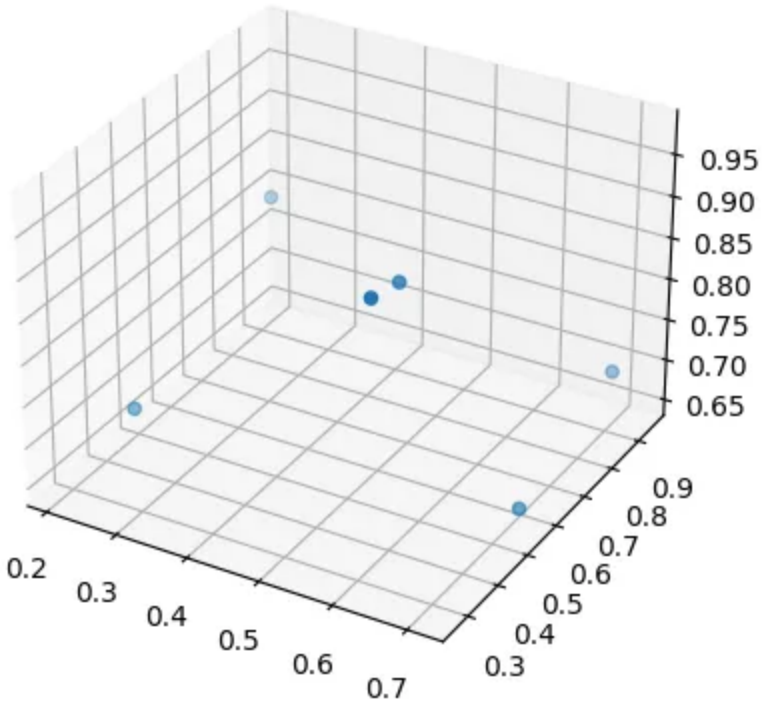


Image by Author

Embeddings Using the PyTorch Module

Since PyTorch will be used to implement the transformer, the ***nn.Embedding*** module can be analyzed. PyTorch defines it as:

> *A simple lookup table that stores embeddings of a fixed dictionary and size.*
>
> *This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.*

This describes exactly what was done in the previous example when using indices instead of one-hot vectors.

At a minimum, ***nn.Embedding*** requires the ***vocab_size*** and the embedding dimension, which will continue to be notated as ***d_model*** moving forward. As a reminder, this is short for the dimension of the model.

The code below creates an embedding matrix with a shape of *(24, 3)*.

```
# vocab size
vocab_size = len(stoi) # 24

# embedding dimensions
d_model = 3

# create the embeddings
lut = nn.Embedding(vocab_size, d_model) # look-up table (lut)

# view the embeddings
lut.state_dict()['weight']
```

```
tensor([[-0.3959,  0.8495,  1.4687],
        [ 0.2437, -0.3289, -0.5475],
        [ 0.9787,  0.7395,  2.0918],
```

```
[-0.4663,  0.4056,  1.2655],
[-1.0054,  1.4883, -0.1254],
[-0.1028, -1.1913,  0.0523],
[-0.2654, -1.0150,  0.4967],
[-0.4653, -1.9941, -1.7128],
[ 0.3894, -0.9368,  1.5543],
[-1.1358, -0.2493,  0.6290],
[-1.4935,  1.1509, -1.8723],
[-0.0421,  1.2857, -0.4009],
[-0.2699, -0.8918, -1.0352],
[-1.3443,  0.4688,  0.1536],
[ 0.3638,  0.1003, -0.2809],
[ 1.4208, -0.0393,  0.7823],
[-0.4473, -0.4605,  1.2681],
[ 1.1315, -1.4704,  0.2809],
[ 0.4270, -0.2067, -0.7951],
[-1.0129,  0.0706, -0.3417],
[ 1.4999, -0.2527,  0.4287],
[-1.9280, -0.6485,  0.4660],
[ 0.0670, -0.5822,  0.0996],
[-0.7058,  0.2849,  1.1725]], grad_fn=<EmbeddingBackward0>)
```

If the same sequence of indices as before, *[11, 23, 21, 22, 5, 15]*, is passed to it, the output will be a *(6, 3)* matrix, where each token is represented by its 3-dimensional embedding vector. The indices must be in the form of a tensor with a data type of either integer or long.

```
indices = torch.Tensor(sequence).long()

embeddings = lut(indices)

embeddings
```

The output would be:

```
tensor([[ 0.7584,  0.2332, -1.2062],
        [-0.2906, -1.2168, -0.2106],
        [ 0.1837, -0.9425, -1.9011],
        [-0.7708, -1.1671,  0.2051],
        [ 1.5548,  1.0912,  0.2006],
        [-0.8765,  0.8829, -1.3169]], grad_fn=<EmbeddingBackward0>)
```

The Embedding Layer in Transformers

In the original paper, the embedding layer is used in the encoder and decoder. The only addition to the ***nn.Embedding*** module is a scalar. The embedding weights are multipled by ***√(d_model)***. This helps preserve the underlying meaning when the embedding is added to the positional encoding in the next step. This essentially makes the positional encoding relatively smaller and decreases its impact on the embeddings. This Stack Overflow thread discusses it more.

To implement this, a class can be created; it will be called ***Embeddings*** and take advantage of PyTorch's ***nn.Embedding*** module. This implementation is based on that of The Annotated Transformer.

```
class Embeddings(nn.Module):
  def __init__(self, vocab_size: int, d_model: int):
    """
    Args:
      vocab_size:     size of vocabulary
      d_model:        dimension of embeddings
    """
    # inherit from nn.Module
    super().__init__()

    # embedding look-up table (lut)
```

```
        self.lut = nn.Embedding(vocab_size, d_model)

        # dimension of embeddings
        self.d_model = d_model

    def forward(self, x: Tensor):
        """
        Args:
            x:              input Tensor (batch_size, seq_length)

        Returns:
                            embedding vector
        """
        # embeddings by constant sqrt(d_model)
        return self.lut(x) * math.sqrt(self.d_model)
```

Forward Pass

This ***Embeddings*** class works the same way as ***nn.Embedding***. The code below demonstrates its usage with the single sequence used in the previous examples.

```
lut = Embeddings(vocab_size, d_model)

lut(indices)
```

```
tensor([[-1.1189,  0.7290,  1.0581],
        [ 1.7204,  0.2048,  0.2926],
        [-0.5726, -2.6856,  2.4975],
        [-0.7735, -0.7224, -2.9520],
        [ 0.2181,  1.1492, -1.2247],
        [ 0.1742, -0.8531, -1.7319]], grad_fn=<MulBackward0>)
```

Up until this point, only a single sequence has been used with in every embedding. However, a model is usually trained with a batch of sequences. This is essentially a list of sequences that are converted to their indices and then embedded. This can be seen in the image below.

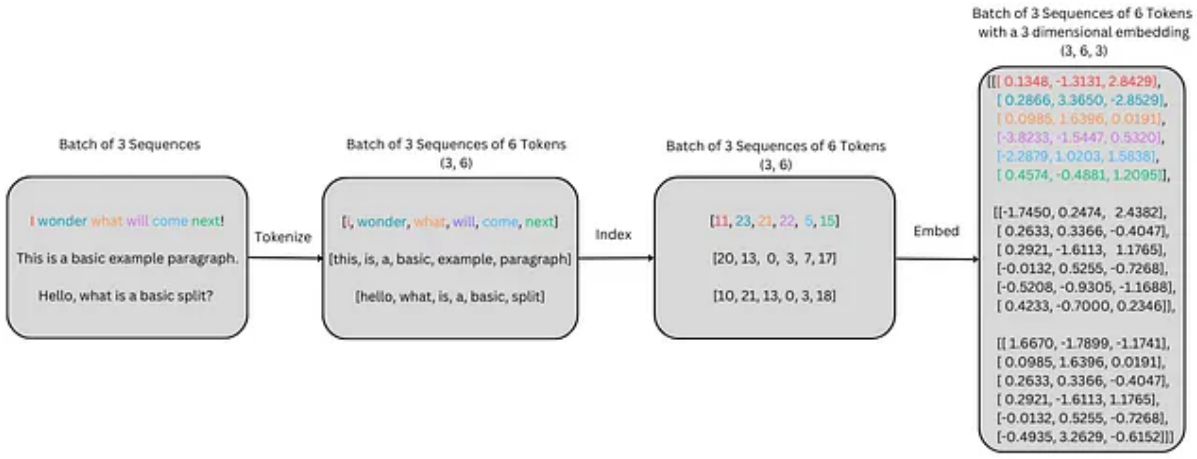


Image by Author

```
# list of sequences (3, )
sequences = ["I wonder what will come next!",
             "This is a basic example paragraph.",
             "Hello, what is a basic split?"]
```

While the previous example is rudimentary, it generalizes for batches of sequences. The example shown in the image above is a batch with three sequences; after tokenization, each sequence is represented by six tokens. The tokenized sequences have a shape of *(3, 6)*, which correlates to *(batch_size, seq_length)*. Essentially, three, six-word sentences.

```
# tokenize the sequences
tokenized_sequences = [tokenize(seq) for seq in sequences]
tokenized_sequences
```

```
[['i', 'wonder', 'what', 'will', 'come', 'next'],
 ['this', 'is', 'a', 'basic', 'example', 'paragraph'],
 ['hello', 'what', 'is', 'a', 'basic', 'split']]
```

These tokenized sequences can then be converted to their indexed representations using the vocabulary.

```
# index the sequences
indexed_sequences = [[stoi[word] for word in seq] for seq in tokenized_sequences

indexed_sequences
```

```
[[11, 23, 21, 22, 5, 15],
 [20, 13,  0,  3, 7, 17],
 [10, 21, 13,  0, 3, 18]]
```

Finally, these indexed sequences can be converted to a tensor that can be passed through the embedding layer.

```
# convert the sequences to a tensor
tensor_sequences = torch.tensor(indexed_sequences).long()
```

```
lut(tensor_sequences)
```

```
tensor([[[ 0.1348, -1.3131,  2.8429],
         [ 0.2866,  3.3650, -2.8529],
         [ 0.0985,  1.6396,  0.0191],
         [-3.8233, -1.5447,  0.5320],
         [-2.2879,  1.0203,  1.5838],
         [ 0.4574, -0.4881,  1.2095]],

        [[-1.7450,  0.2474,  2.4382],
         [ 0.2633,  0.3366, -0.4047],
         [ 0.2921, -1.6113,  1.1765],
         [-0.0132,  0.5255, -0.7268],
         [-0.5208, -0.9305, -1.1688],
         [ 0.4233, -0.7000,  0.2346]],

        [[ 1.6670, -1.7899, -1.1741],
         [ 0.0985,  1.6396,  0.0191],
         [ 0.2633,  0.3366, -0.4047],
         [ 0.2921, -1.6113,  1.1765],
         [-0.0132,  0.5255, -0.7268],
         [-0.4935,  3.2629, -0.6152]]], grad_fn=<MulBackward0>)
```

The output will be a *(3, 6, 3)* matrix, which correlates to ***(batch_size, seq_length, d_model)***. Essentially, each indexed token is replaced by its corresponding 3-dimensional embedding vector.

Before moving to the next sections, it is extremely important to understand the shape of this data, ***(batch_size, seq_length, d_model)***:

- ***batch_size*** correlates to the number of sequences provided at a time, which is normally 16, 32, or 64.
- ***seq_length*** correlates to the number of words or tokens in each sequence after tokenization.

- *d_model* correlates to the size of the model after each token has been embedded.

The article for the Positional Encodings is next in the series.

Please don't forget to like and follow for more! :)

References

1. Image by Will Koehrsen
2. PyTorch's Embedding Module
3. Stack Overflow Discussion
4. The Annotated Transformer
5. Transformers from scratch

Embedding Layer | Transformers | Translation | Machine Learning | Artificial Intelligence

More from the list: "NLP"

Curated by Himanshu Birla

Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

· 11 min read · Sep 4, 2021

Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

· 6 min read · Sep 3, 2021

Jon Gi... in

The Word2ve

· 15 min rea

View list

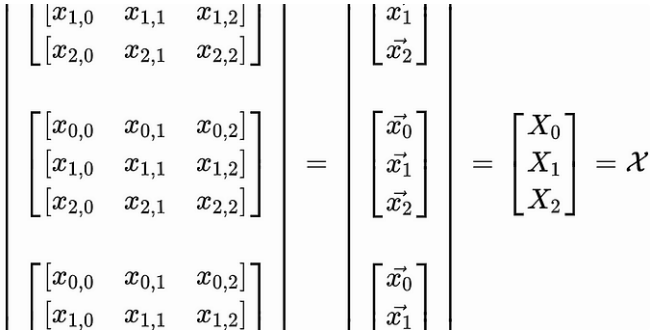
Written by Hunter Phillips

Following

219 Followers

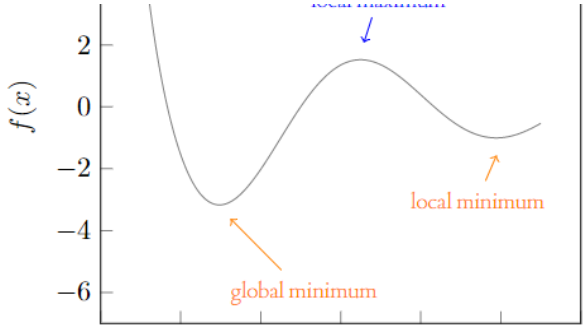
Machine Learning Engineer and Data Scientist

More from Hunter Phillips



Hunter Phillips

A Simple Introduction to Tensors



Hunter Phillips

A Simple Introduction to Gradient Descent

A tensor is a generalization of vectors and matrices to n dimensions. Understanding ho...

11 min read · May 10

273 5

Gradient descent is one of the most common optimization algorithms in machine learning....

10 min read · May 18

108

Hunter Phillips

What is an RDD in PySpark?

This article covers the basic uses of resilient distributed datasets in PySpark. It includes...

7 min read · Jun 10

100

Hunter Phillips

An Introduction to Machine Learning in Python: The Normal...

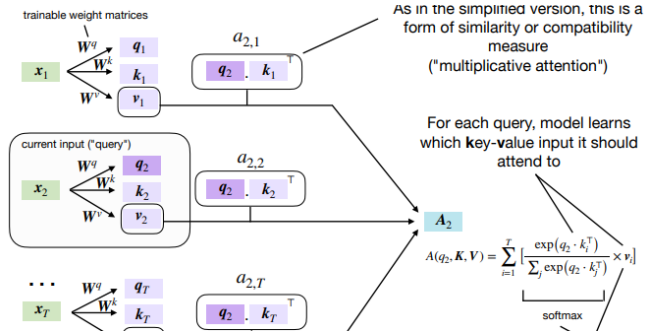
The Normal Equation is a closed-form solution for minimizing a cost function and...

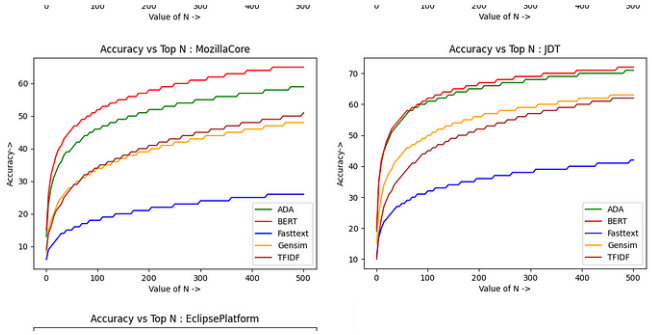
4 min read · May 22

175

See all from Hunter Phillips

Recommended from Medium





Zain ul Abideen

Attention Is All You Need: The Core Idea of the Transformer

An overview of the Transformer model and its key components.

6 min read · Jun 26

144

Avinash Patil

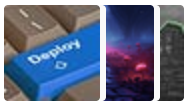
Embeddings: BERT better than ChatGPT4?

In this study, we compared the effectiveness of semantic textual similarity methods for...

4 min read · Sep 19

3 1

Lists



Predictive Modeling w/ Python

20 stories · 452 saves

Natural Language Processing

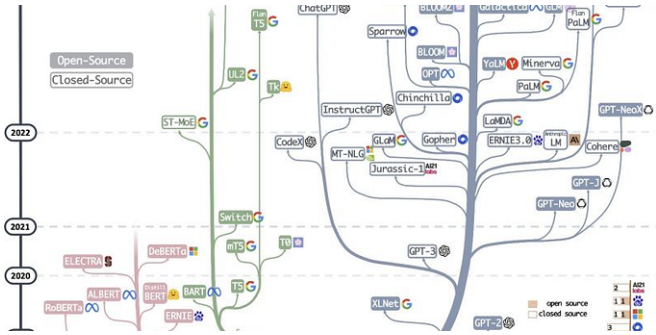
669 stories · 283 saves

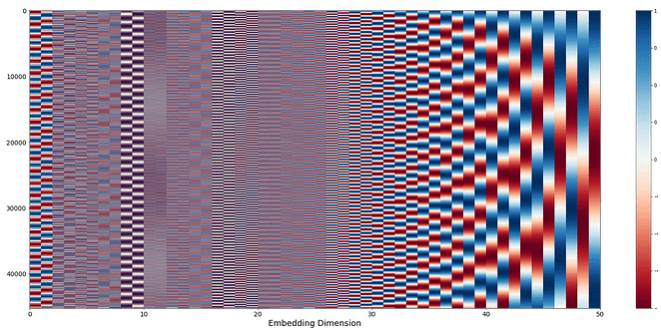
AI Regulation

6 stories · 138 saves

ChatGPT prompts

24 stories · 459 saves





Haifeng Li

A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's...

15 min read · Sep 14

372

Eugene Ku

Transformer Architecture (Part 1—Positional Encoding)

Nowadays, arguably the most popular and influential model behind the hypes of deep...

4 min read · Aug 22

15

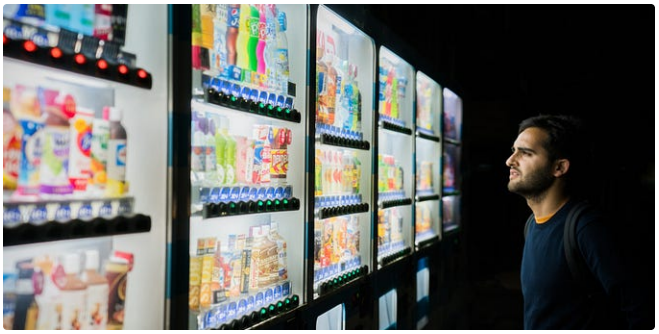
Abby Morgan in Generative AI

Explainable AI: Visualizing Attention in Transformers

And logging the results in an experiment tracking tool

13 min read · Jul 17

1.5K 7



Ovbude Ehi

Recommendation Engines

Practical Techniques: Content-Based Filtering, Collaborative Filtering and...

11 min read · May 8

5

See more recommendations