10-315 Notes
Word Embeddings
Version: 0.1

Carnegie Mellon University
Machine Learning Department

Contents

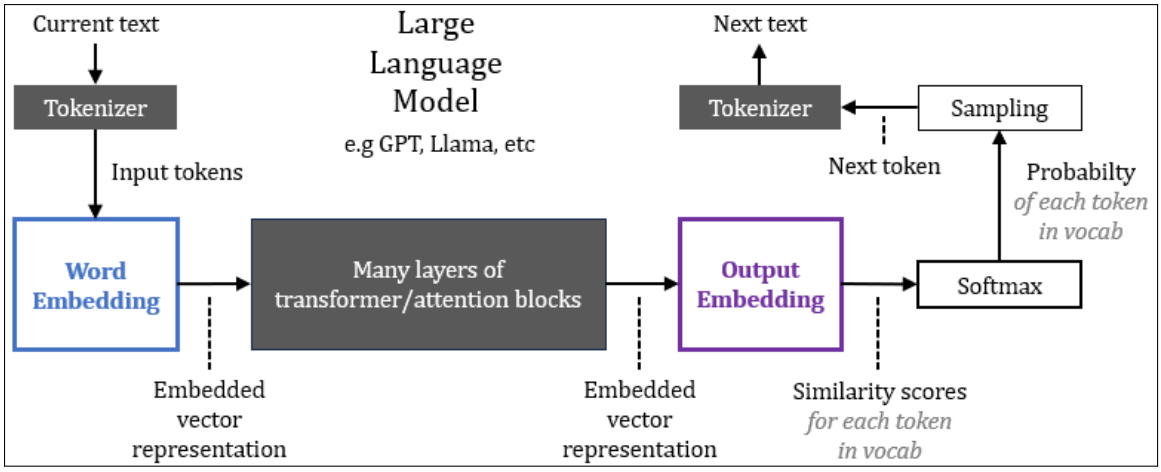
1 Word embeddings

When processing text, we need to convert words into numbers. One way to do this is to use a one-hot encoding, where each word in the vocabulary is represented by a vector of zeros with a single one at the index corresponding to that word. For example, if our vocabulary is {dog, ran, the}, then the one-hot encoding for "ran" would be $[0, 1, 0]$. However, this representation has some limitations. For example, it does not capture any information about the relationships between words. To address this, we can use **word embeddings**, which are dense vector representations of words that capture their meanings and relationships. Word embeddings are typically learned from large corpora of text using neural networks. The idea is to train a model to predict the next word in a sentence given the previous words. During this process, the model learns to represent words that appear in similar contexts with similar vectors.

Word embeddings allow us to **embed** words (or more genearlly, tokens) into vectors of real numbers. They also help us to **unembed** vectors, converting them back into words. *Note:* while we commonly refer to "word embeddings", it would be more accurate to refer to them as "token embeddings", since they can be used for any token, not just words. For example, we can use them for punctuation marks, numbers, portions of words, and even emojis and other character symbols.

Word embedding models are sometimes trained independently of the task at hand, and the resulting embeddings are then used to engineer features for other NLP tasks, such as sentiment analysis or named entity recognition.

Alternatively, word embeddings can be trained as part of a larger model that is specifically designed for a particular task. In this case, the embeddings are learned in conjunction with the other parameters of the model, and they are optimized to perform well on that task. This is the approach used in many modern large language models, such as GPT and LLaMA. Word embeddings are one of the first and last layers in these models.



2 Running example

Through these notes, we'll use the following training **corpus** as a running example. A corpus is just a body of text, often much, much bigger than this small example. Our training corpus is **tokenized** into words and puctuation **tokens**. The resulting **vocabulary** is the set of unique tokens in the corpus and in this case is only 8 tokens long.

Corpus:

```
The dog ran.
The dog ate.
The dog ran the zoo.
The cat ate the dog!
The cat ran the zoo.
```

(Simple) Tokenized Corpus:

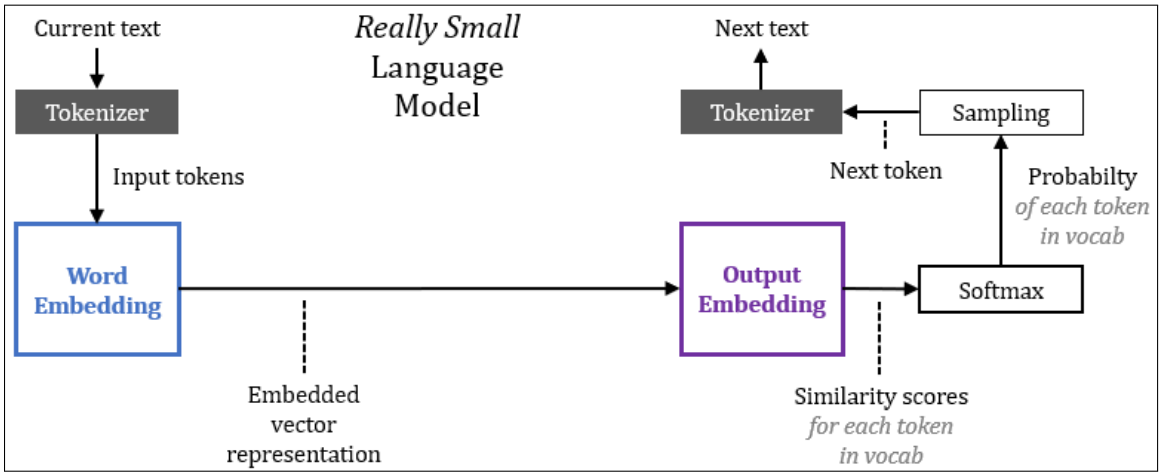
```
[ 'the', 'dog', 'ran', '.', 'the',
  'dog', 'ate', '.', 'the', 'dog',
  'ran', 'the', 'zoo', '.', 'the',
  'cat', 'ate', 'the', 'dog', '!',
  'the', 'cat', 'ran', 'the',
  'zoo', '.' ]
```

Vocabulary:

```
[ '!',
  '.',
  'ate',
  'cat',
  'dog',
  'ran',
  'the',
  'zoo' ]
```

3 Word Embedding Language Model

We are going to implement a really small **language model** (LM) that predicts the next word in a sentence given the previous word. We are essentially striping out all of the attention/transformer layers in a large language model (LLM) and just using the embedding layers. This is a very simple model, but it will help us to understand the critical concept of vector representation of words and how they are used in LLMs, including the attention mechanism.



3.1 Task: Text Generation

The task for our language model is to predict the next token in a sentence given the previous token. This is a common task in natural language processing (NLP) and is often used as a benchmark for evaluating the performance of language models. In our simple language model, we'll only use one previous token as our input **context**. In a more complex model, we could use multiple previous tokens as input, but for our purposes, we'll keep it simple.

This is equivalent to a Markov chain of order 1, where the next state (token) depends only on the current state (token). In our case, the current state is the previous token, and the next state is the token we want to predict:

$$p(\mathbf{x}_{t+1} \mid \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t) = p(\mathbf{x}_{t+1} \mid \mathbf{x}_t)$$

Thus, our task is to learn the conditional probability of the next token given the previous token:

$$\hat{\mathbf{x}}_{t+1} = h(\mathbf{x}_t) = p(\mathbf{x}_{t+1} \mid \mathbf{x}_t)$$

3.2 Model

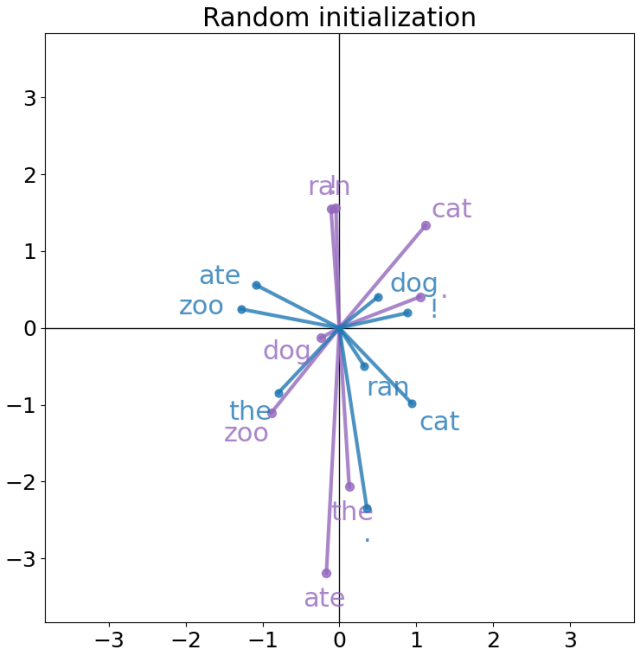
Let K be the the hyperparameter that defines the length of the embedded vectors and let N_{vocab} be the size of the vocabulary. Our model will have two sets of K-dimensional vectors for each token in the vocabulary: one for the previous token and one for the next token. Theses vectors are parameters; they will be initialized randomly and then updated as we train our model to better predict the next token.

We will stack these two sets of N_{vocab} vectors into the rows of two $N_{vocab} \times K$ matrices, V and U.

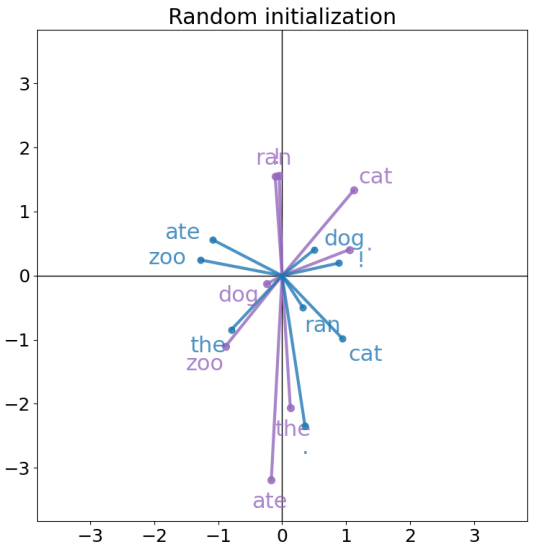
Below are examples of V and U for a for our example dataset with embedded dimension $K = 2$. These are the randomly initialized vectors for each token in the vocabulary. Conveniently, $K = 2$, so we can visualize the vectors in the 2D embedded space.

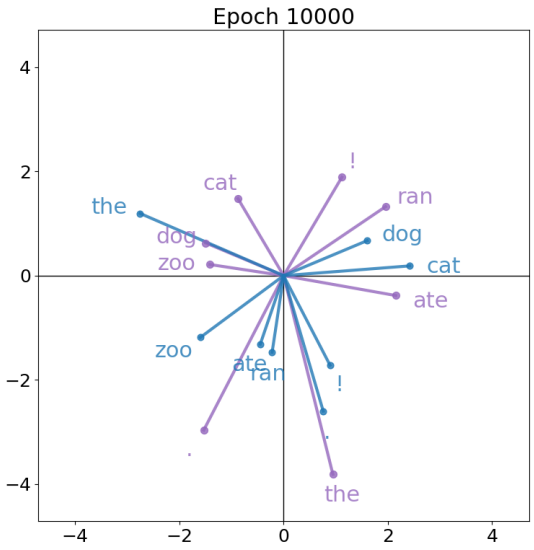
```
V:
!:     0.884,   0.196
.:     0.358,  -2.343
ate:  -1.085,   0.560
cat:   0.939,  -0.978
dog:   0.503,   0.406
ran:   0.323,  -0.493
the:  -0.792,  -0.842
zoo:  -1.280,   0.246

U:
!:    -0.044,   1.568
.:     1.051,   0.406
ate:  -0.169,  -3.190
cat:   1.120,   1.333
dog:  -0.243,  -0.130
ran:  -0.109,   1.556
the:   0.129,  -2.067
zoo:  -0.885,  -1.105
```



In the figure below, we can see the V and U vectors before and after training for 10,000 epochs. At first glance, they both look pretty random, but as we learn more about sampling and training our model, we'll be able to see some organization in the trained vectors.





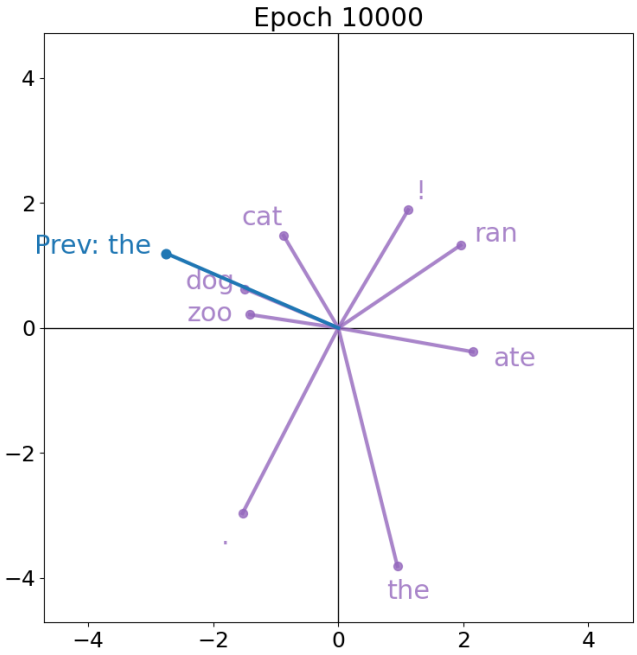
3.3 Cosine Similarity

The way that our model works is that we take the vector associated with the previous token, $\mathbf{v}_j$ and compare it to all of the next token vectors in U.

Process:

1. Look up the index i for the vocab token 'the'
2. Access the i-th row of V, $\mathbf{v}_i$
3. Compare $\mathbf{v}_i$ to all of the rows of U

For example, if the previous token is 'the', then we focus only on the vector for 'the' from V and compare it to all of the vectors in U. Given the plots below for previous token 'the', what do you think the model will predict as the next token? We would probably expect the model to predict 'dog' as the next token, as the dog vector in U seems to be the most similar to the 'the' vector in V.



To make this more rigorous, we need to define a similarity metric to compare two vectors. We've seen Euclidean distance before $d(\mathbf{u}, \mathbf{v}) = \|\mathbf{u}-\mathbf{v}\|_2$, but a more common metric for comparing vectors is within neural networks is **dot product similarity** or **(unnormalized) cosine similarity**.

Cosine similarity is defined as the cosine of the angle between two vectors. It is a measure of how similar two vectors are, regardless of their magnitude. The cosine similarity between two vectors $\mathbf{u}$ and $\mathbf{v}$ is defined as:

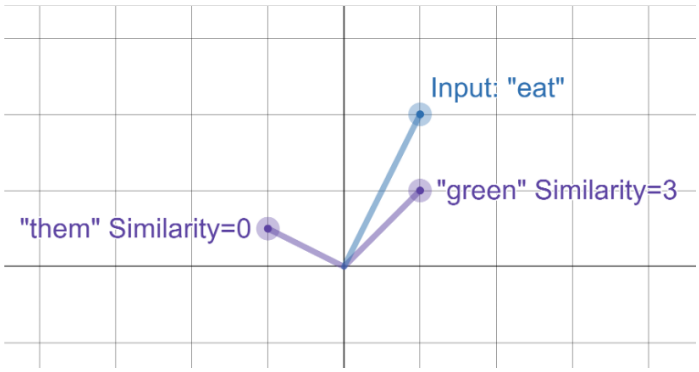
$$f(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u}^\top \mathbf{v}}{\|\mathbf{u}\|_2 \|\mathbf{v}\|_2}$$

This is equivalent to the dot product of the two vectors divided by the product of their magnitudes. The cosine similarity ranges from -1 to 1, where 1 indicates that the two vectors are identical, 0 indicates that they are orthogonal (i.e., not similar), and -1 indicates that they are opposite.

Rather than using the cosine similarity, we use the much more computationally efficient dot product similarity. The dot product simliarity just drops the normalization aspect of cosine similarity, so we can just use the dot product of the two vectors:

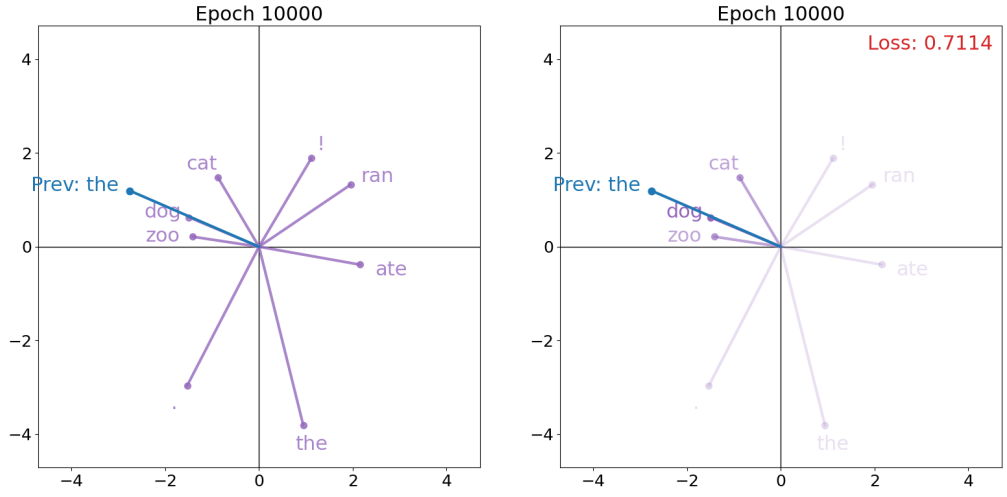
$$f(\mathbf{u}, \mathbf{v}) = \mathbf{u}^\top \mathbf{v}$$

See our Desmos demo for (unnormalized) cosine similarity. Moving these vectors around, you can see how both the angle and the magnitude of the vectors affect the similarity score. Rather than having values range from -1 to 1, the dot product similarity can take on any value from $-\infty$ to ∞.



3.4 Sampling

Now that we have similarity scores we can use them to choose the next token. We could simply take the argmax of the similarity scores. However, given that this is really a classification problem (where the classes are the tokens in the vocabulary), we can would much rather have a probability distribution over next tokens. Just like in logistic regression, we can use the softmax function to convert the similarity scores into a probability distribution over the next tokens.

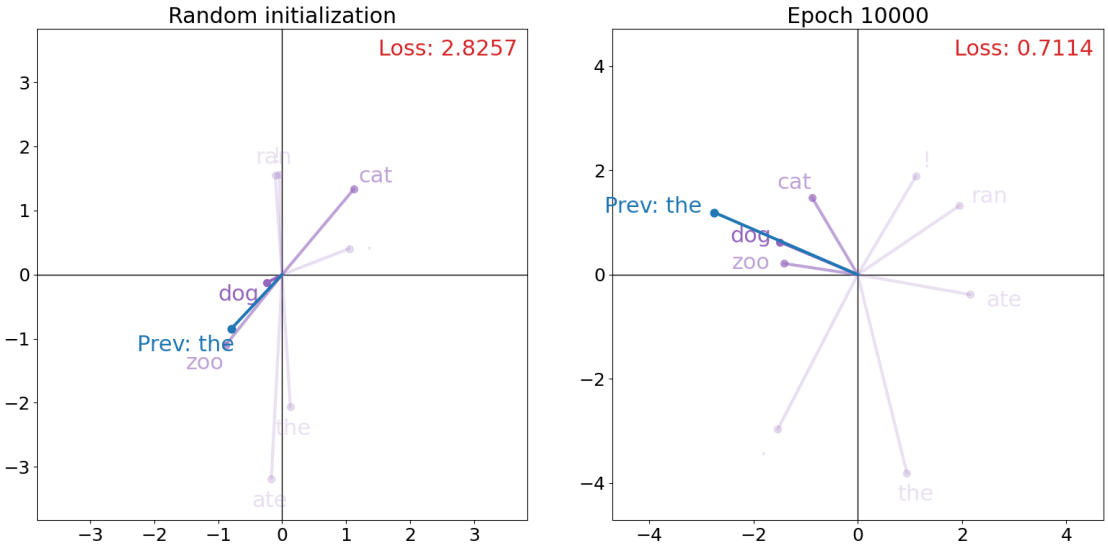


For previous token 'the' and next token vectors shown above, we can compute the similarity scores all once with $U^\top \mathbf{v}_j$ and then apply the softmax function to get the probability distribution over the next tokens.

```
Similarity scores U@v:    Probabilities softmax(U@v):
    the !   -0.823            the !   0.002
    the .    0.683            the .   0.008
    the ate -6.376            the ate 0.000
    the cat  4.174            the cat 0.250
    the dog  4.860            the dog 0.496
    the ran -3.803            the ran 0.000
    the the -7.149            the the 0.000
    the zoo  4.152            the zoo 0.245
```

3.5 Training

Vectors before and after training, weighted by $\hat{\mathbf{x}}_{t+1} = p(\mathbf{x}_{t+1} \mid x_t = \texttt{`the'})$



20 generated tokens from **untrained** (randomly initialized) model starting with context "the dog":

the dog cat ate ran zoo zoo zoo dog dog cat cat ate ran . ate . ate ate ! the ate

20 generated tokens from **trained** model starting with context "the dog":

the dog ate . the dog ! the zoo . the dog ran the zoo . the dog ate the dog !

References

Y. Bengio, R. Ducharme, P. Vincent. A neural probabilistic language model. Journal of Machine Learning Research, 3:1137-1155, 2003. https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf

T. Mikolov, K. Chen, G. Corrado, J. Dean. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781, 2013. https://arxiv.org/pdf/1301.3781