

# Session 9 - Neural Embedding

---

**Due** Saturday by 11:59pm **Points** 1,000

**Available** Oct 10 at 10am - Oct 17 at 11:59pm 8 days

---

## Session 9 - Neural Word Embeddings

- [Introduction](#)
- [Understanding n-grams and bag-of-words](#)
- [One-hot encoding of words and characters](#)
- [Character-level one-hot encoding](#)
- [Using word embedding](#)
- [Learning word embeddings with the embedding layer](#)
- [Using Pre-trained word Embeddings](#)
- [Embedding Types](#)
- [Coding Walk-through](#)
- [Assignment](#)

### Introduction





How many of you understood what's written here?

For those who understood, do you agree that it conveys the same emotions that your brain feels while reading it and while listening to it in an orchestra?

This is an attempt by humans to 'write' music.

**Yet, it works!**

Computers understand numbers.

```
[ 0.50451 , 0.68607 , -0.59517 , -0.022801, 0.60046 , -0.13498 , -0.08813 , 0.47377 , -0.61798 , -0.31012 ,
-0.076666, 1.493 , -0.034189, -0.98173 , 0.68229 , 0.81722 , -0.51874 , -0.31503 , -0.55809 , 0.66421 , 0.1961
, -0.13495 , -0.11476 , -0.30344 , 0.41177 , -2.223 , -1.0756 , -1.0783 , -0.34354 , 0.33505 , 1.9927 ,
-0.04234 , -0.64319 , 0.71125 , 0.49159 , 0.16754 , 0.34344 , -0.25663 , -0.8523 , 0.1661 , 0.40102 , 1.1685 ,
-1.0137 , -0.21585 , -0.15155 , 0.78321 , -0.91241 , -1.6106 , -0.64426 , -0.51042 ]
```

**NLP is an attempt by humans for computers to understand human languages by performing powerful mathematical operations and statistics on words.**

[Source \(http://faculty.neu.edu.cn/yury/AAI/Textbook/Deep%20Learning%20with%20Python.pdf\)](http://faculty.neu.edu.cn/yury/AAI/Textbook/Deep%20Learning%20with%20Python.pdf)

Text is one of the most widespread forms of sequence data. It can be understood as either a sequence of characters or a sequence of words, but it's most common to work at the level of words. Though none of the models we would work on would truly understand the text in a human sense; rather, these models can map the statistical structure of written language, which is sufficient to solve many simple to complex textual tasks like document classification, sentiment analysis, author identification, and even question-answering (QA).

Deep Learning for NLP is pattern recognition applied to words, sentences, and paragraphs, in much the same way that computer vision is pattern recognition to pixels.

Like all other neural networks, deep-learning models don't take as input raw text: they only work with numeric tensors. Vectorizing text is the process of transforming text into numeric tensors. This can be done in multiple ways:

- segment text into words, and transform each word into a vector
- segment text into characters, and transform each character into a vector
- extract n-gram of words or characters, and transform each n-gram into a vector. N-grams are overlapping groups of multiple groups of multiple consecutive words or characters.

Collectively, the different units into which you can break down the text (words, characters, or n-grams) are called tokens, and breaking text into such tokens is called tokenization.

All text-vectorization processes consist of applying some tokenization scheme and then associating

numeric vectors with the generated tokens.

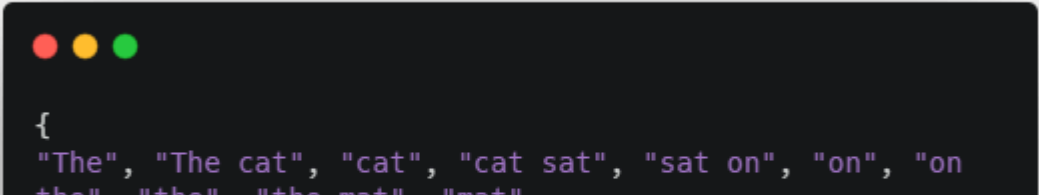
These vectors, packed into sequence tensors, are fed into deep neural networks.

There are multiple ways to associate a vector with a token. In this section, I (François Chollet) will present two major ones: one-hot encoding to tokens and token embedding (typically used exclusively for words and word embedding).

## Understanding n-grams and bag-of-words

Word n-grams are a group of N "or fewer" consecutive words that you can extract from a sentence. The same concept may also be applied to characters instead of words

Here's a simple example. Consider the sentence "The cat sat on the mat." It may be decomposed into the following set of **2-grams**:



```
{  
  "The", "The cat", "cat", "cat sat", "sat on", "on", "on  
  the", "the", "the mat", "mat"
```

```
the , the , the mat , mat  
}
```

It may also be decomposed into the following set of **3-grams**:

```
{  
"The", "The cat", "cat", "cat sat", "The cat sat",  
"sat", "sat on", "on", "cat sat on", "on the", "the",  
"sat on the", "the mat", "mat", "on the mat"  
}
```

Such a set is called a bag-of-2-grams or bag-of-3-grams, respectively. The term bag here refers to the fact that you're dealing with a set of tokens rather than a list of sequences: the tokens have no specific order. This family of tokenization methods is called bag-of-words.

Because bags-of-words isn't an order-preserving tokenization method (the tokens generated are understood as a set, not a sequence, and the general structure of the sentences is lost), it tends to be used in **shallow language-processing** models rather than in deep-learning models.

Extracting n-grams is a form of feature engineering, and deep learning does away with this kind of rigid, brittle approach, replacing it with hierarchical feature learning. We'll not cover n-grams further.

## One-hot encoding of words and characters

One-hot encoding is the most common, most basics way to turn a token into a vector. It consists of associating a unique integer index with every word and then turning this integer index  $i$  into a binary vector of size  $N$  (the size of the vocabulary); the vector is all zeros except for the  $i^{th}$  entry, which is 1.

Let's look at an example:

```
import numpy as np
```

```
#Initial data: one entry per sample (in this example, a sample is a sentence, but it co
```

```

document)

samples = ['The cat sat on the mat.', 'The dog are my breakfast.']

# builds an index of all tokens in the data

token_index = {}

for sample in samples:
    # Tokensizes the samples via the split method. In real life, you'd also strip punctuation
    # characters from the samples.
    for word in sample.split():
        if word not in token_index:
            token_index[word] = len(token_index) + 1
        # assigns a unique index to each unique word. Note that you don't attribute index 0 to
        # Vectorizes the samples. You'll only consider the first max_length words in each sample
    max_length = 10

    # This is where you store the results.
    results = np.zeros(shape=(len(samples),
                              max_length,
                              max(token_index.values()) + 1))

    for i, sample in enumerate(samples):
        for j, word in list(enumerate(sample.split()))[:max_length]:
            index = token_index.get(word)
            results[i, j, index] = 1

```

## Character-level one-hot encoding (toy example)

```

import string

```

```

samples = ['The cat sat on the mat.', 'The dog ate my

```

```

samples = ['The cat sat on the mat.', 'The dog ate my homework']

# All printable ASCII characters

characters = string.printable
token_index = dict(zip(range(1, len(characters) + 1),
                        characters))

max_length = 50
results = np.zeros((len(samples), max_length,
                    max(token_index.keys()) + 1))
for i, sample in enumerate(samples):
    for j, character in enumerate(sample):
        index = token_index.get(character)
        results[i, j, index] = 1

```

Always use built-in utilities (but TorchText is tough, so prefer PyTorch-NLP) for doing one-hot encoding of text at the word level or character level, starting from raw text data. You should use these utilities because they take care of a number of important features such as stripping special characters from strings and only taking into account the N most common words in your dataset(a common restriction, to avoid dealing with very large input vector spaces).

A variant of one-hot encoding is the so-called one-hot hashing trick, which you can use when the number of unique tokens in your vocabulary is too large to handle explicitly. Instead of explicitly assigning an index to each word and keeping a reference of these indices in a dictionary, you can hash words into vectors of fixed size.

```
samples = ['The cat sat on the mat.', 'The dog ate my homework.']
```



```
#stores the words as vectors of size 1000. If you have  
close to 1000 words (or more), you'll see many hash  
collisions, which will decrease the accuracy of this  
encoding method.
```

```
dimensionality = 1000  
max_length = 10  
results = np.zeros((len(samples), max_length,  
dimensionality))  
  
for i, sample in enumerate(samples):  
    for j, word in list(enumerate(sample.split()))  
        [:max_length]:  
        # hashes the word into a random integer index between 0  
        and 1000  
        index = abs(hash(word)) % dimensionality  
        results[i, j, index] = 1
```

## Using Word Embeddings

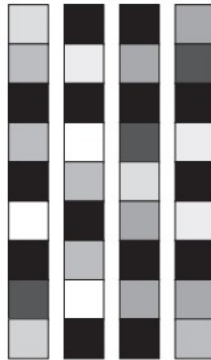
Another popular and powerful way to associate a vector with a word is the use of dense word vectors, also called word embeddings. Whereas the vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high-dimensional (same dimensionality as the number of words in the vocabulary), word embeddings are low-dimensional floating-point vectors (that is, dense vectors, as opposed to sparse vectors)





One-hot word vectors:

- Sparse
- High-dimensional
- Hardcoded



Word embeddings:

- Dense
- Lower-dimensional
- Learned from data

**Figure 6.2** Whereas word representations obtained from one-hot encoding or hashing are sparse, high-dimensional, and hardcoded, word embeddings are dense, relatively low-dimensional, and learned from data.

Unlike the word vectors obtained via one-hot encoding, word embeddings are learned from data. It's common to see word embeddings that are 256, 512 or 1024 dimensional when dealing with very large vocabularies. On the other hand, one-hot encoding leads to vectors that are 20,000 dimensional or greater (capturing a vocabulary of 20,000 tokens, in this case). So, word embeddings pack more information into far fewer dimensions.

There are two ways to obtain word embeddings:

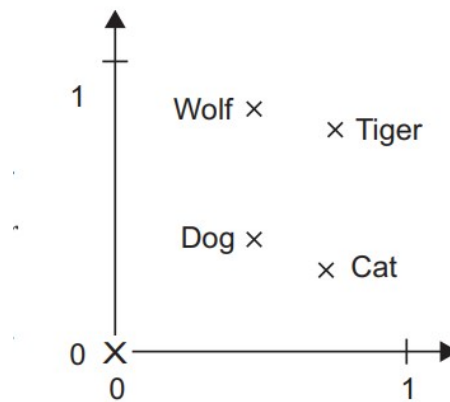
- Learn word embeddings jointly with the main task you care about (such as document classification or sentiment prediction). In this setup, you start with random word vectors and then learn word vectors in the same way you learn the weights of a neural network
- Load into your model word embeddings that were pre-computed using a different machine-learning task than the one you're trying to solve. These are called pre-trained word embeddings.

## Learning Word Embeddings with the Embedding Layer

The simplest way to associate a dense vector with a word is to choose the vector at random. The problem with this approach is that the resulting embedding space has no structure; for instance, the words accurate and exact may end up with completely different embeddings, even though they're interchangeable in most sentences. It's difficult for a deep neural network to make sense of such a noisy,

unstructured embedding space.

To get a bit more abstract, the geometric relationships between word vectors should reflect the semantic relationships between these words. Word embeddings are meant to map human language into a geometric space. For instance, in reasonable embedding space, you would expect synonyms to be embedded into similar word vectors; and in general, you would expect the geometric distance (such as L2 distance) between any two-word vectors to relate the semantic distance between the associated words (words meaning different things are embedded at points far away from each other, whereas related words are closer). In addition to distance, you may want specific directions in the embedding space to be meaningful.



Is there some ideal word-embedding space that would perfectly map human language and could be used for any natural-language-processing task? *Probably, but we have yet to compute anything of the sort.*

The word-embedding space for an English-language movie-review sentiment-analysis model may look different from the embedding space for an English-language legal-document-classification model because the importance of certain semantic relationships varies from task to task.

It is thus reasonable to learn a new embedding space with every new task.

```
torch.nn.Embedding(num_embeddings: int, embedding_dim:
int, padding_idx: Optional[int] = None, max_norm:
Optional[float] = None, norm_type: float = 2.0)
```

```

Optional[float] = None, norm_type: float = 2.0,
scale_grad_by_freq: bool = False, sparse: bool = False,
_weight: Optional[torch.Tensor] = None)

#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.

```

- **num\_embeddings** ([int](https://docs.python.org/3/library/functions.html#int)) – size of the dictionary of embeddings
- **embedding\_dim** ([int](https://docs.python.org/3/library/functions.html#int)) – the size of each embedding vector
- **padding\_idx** ([int](https://docs.python.org/3/library/functions.html#int), *optional*) – If given, pads the output with the embedding vector at `padding_idx` (initialized to zeros) whenever it encounters the index.
- **max\_norm** ([float](https://docs.python.org/3/library/functions.html#float), *optional*) – If given, each embedding vector with norm larger than `max_norm` is renormalized to have norm `max_norm`.
- **norm\_type** ([float](https://docs.python.org/3/library/functions.html#float), *optional*) – The p of the p-norm to compute for the `max_norm` option. Default `2`.
- **scale\_grad\_by\_freq** (*boolean, optional*) – If given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `False`.
- **sparse** ([bool](https://docs.python.org/3/library/functions.html#bool), *optional*) – If `True`, gradient w.r.t. `weight` matrix will be a sparse tensor. See Notes for more details regarding sparse gradients.

```

>>> # an Embedding module containing 10 tensors of size
3
>>> embedding = nn.Embedding(10, 3)

```

```
>>> embedding = nn.Embedding(10, 3)
>>> # a batch of 2 samples of 4 indices each
>>> input = torch.LongTensor([[1, 2, 4, 5], [4, 3, 2, 9]])
>>> embedding(input)
tensor([[[ -0.0251, -1.6902,  0.7172],
          [-0.6431,  0.0748,  0.6969],
          [ 1.4970,  1.3448, -0.9685],
          [-0.3677, -2.7265, -0.1685]],
        [[ 1.4970,  1.3448, -0.9685],
          [ 0.4362, -0.4004,  0.9400],
          [-0.6431,  0.0748,  0.6969],
          [ 0.9124, -2.3616,  1.1151]]]])

>>> # example with padding_idx
>>> embedding = nn.Embedding(10, 3, padding_idx=0)
>>> input = torch.LongTensor([[0, 2, 0, 5]])
>>> embedding(input)
tensor([[[ 0.0000,  0.0000,  0.0000],
          [ 0.1535, -2.0309,  0.9315],
          [ 0.0000,  0.0000,  0.0000],
          [-0.1655,  0.9897,  0.0635]]]])
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```

import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)

word_to_ix = {"hello": 0, "world": 1}
embeds = nn.Embedding(2, 5) # 2 words in vocab, 5 dimensional embeddings
lookup_tensor = torch.tensor([word_to_ix["hello"]], dtype=torch.long)
hello_embed = embeds(lookup_tensor)
print(hello_embed)

#tensor([[ 0.6614,  0.2669,  0.0617,  0.6213, -0.4519]],
        grad_fn=<EmbeddingBackward>)

```

The Embedding Layer is best understood as a dictionary that maps integer indices (which stand for specific words) to dense vectors. It takes integers as input, it looks up these integers in an internal dictionary, and it returns the associated vectors. It's effectively a dictionary lookup.

Word index → Embedding layer → Corresponding word vector

The embedding layer takes as input a 2D tensor of integers, of shape (samples, sequence\_length), where each entry is a sequence of integers. It can embed sequences of variable length: for instance, you could feed into the Embedding layer in the previous examples batch with shape (32, 10) (batch of 32 sequence of length 10) or (64, 15). All sequences in a batch must have the same length, so sequences that are shorter than others should be padded with zeros, and sequences that are longer should be truncated.

This layer returns a 3D floating-point tensor of shape (samples, sequence\_length, embedding\_dimensionality). Such a 3D tensor can then be processed by an RNN layer or a 1D convolution layer.

When you instantiate an Embedding layer, its weights (its internal dictionary of token vectors) are

initially random, just as with any other layer. During training, these word vectors are gradually adjusted.

initially random, just as with any other layer. During training, these word vectors are gradually adjusted via back-propagation, structuring the space into something the downstream model can exploit.

## Using Pre-trained word Embeddings

Instead of learning word embeddings jointly with the problem you want to solve, you can load embedding vectors from a pre-computed embedded space that you know is highly structured and exhibits useful properties - that capture generic aspects of language structure.

Such word embeddings are generally computed using word-occurrence statistics, using a variety of techniques, some involving neural networks, other not.

## Embedding Types

Word2Vec is a shallow, two-layer neural networks which is trained to reconstruct linguistic contexts of words.

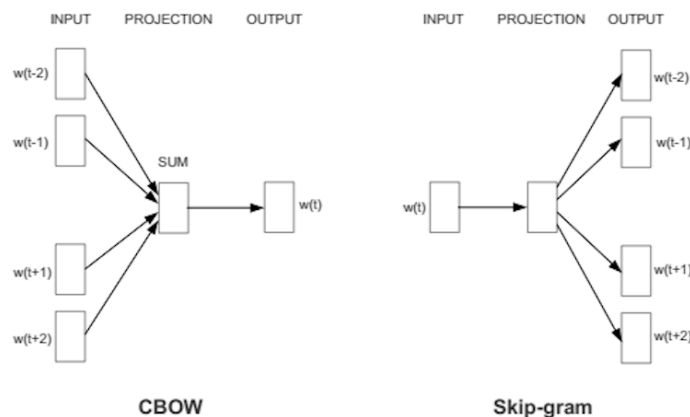
It takes as its input a large corpus of words and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space.

Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another in the space.

Word2Vec is a particularly computationally-efficient predictive model for learning word embeddings from raw text.

It comes in two flavors, the Continuous Bag-of-Words (CBOW) model and the Skip-Gram model.

Algorithmically, these models are similar.



## Continuous Bag-of-Words (CBOW)

CBOW predicts target words (e.g. 'mat') from the surrounding context words ('the cat sits on the').

Statistically, it has the effect that CBOW smoothes over a lot of the distributional information (by treating an entire context as one observation). For the most part, this turns out to be a useful thing for smaller datasets.

## Skip-Gram

Skip-gram predicts surrounding context words from the target words (inverse of CBOW).

Statistically, skip-gram treats each context-target pair as a new observation, and this tends to do better when we have larger datasets.

## How does Word2Vec produce word embeddings?

Word2Vec uses a trick you may have seen elsewhere in machine learning.

Word2Vec is a simple neural network with a single hidden layer, and like all neural networks, it has weights, and during training, its goal is to adjust those weights to reduce a loss function. However,



Word2Vec is not going to be used for the task it was trained on, instead, we will just take its hidden weights, use them as our word embeddings, and toss the rest of the model (*this came out a little bit evil*).

Another place you may have seen this trick is in unsupervised feature learning, where an auto-encoder is trained to compress an input vector in the hidden layer and decompress it back to the original in the output layer. After it's done training, the output layer (the decompression step) is stripped off and only the hidden layer is used since it has learned good features, it's a trick for learning good image features without having labeled training data.

## Architecture

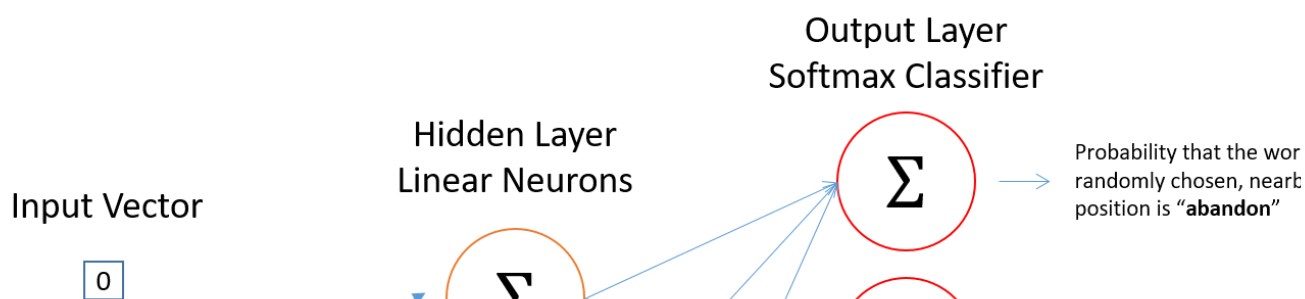
The architecture is similar to an autoencoder's one, you take a large input vector, compress it down to a smaller dense vector and then instead of decompressing it back to the original input vector as you do with autoencoders, you output probabilities of target words.

First of all, we cannot feed a word as string into a neural network.

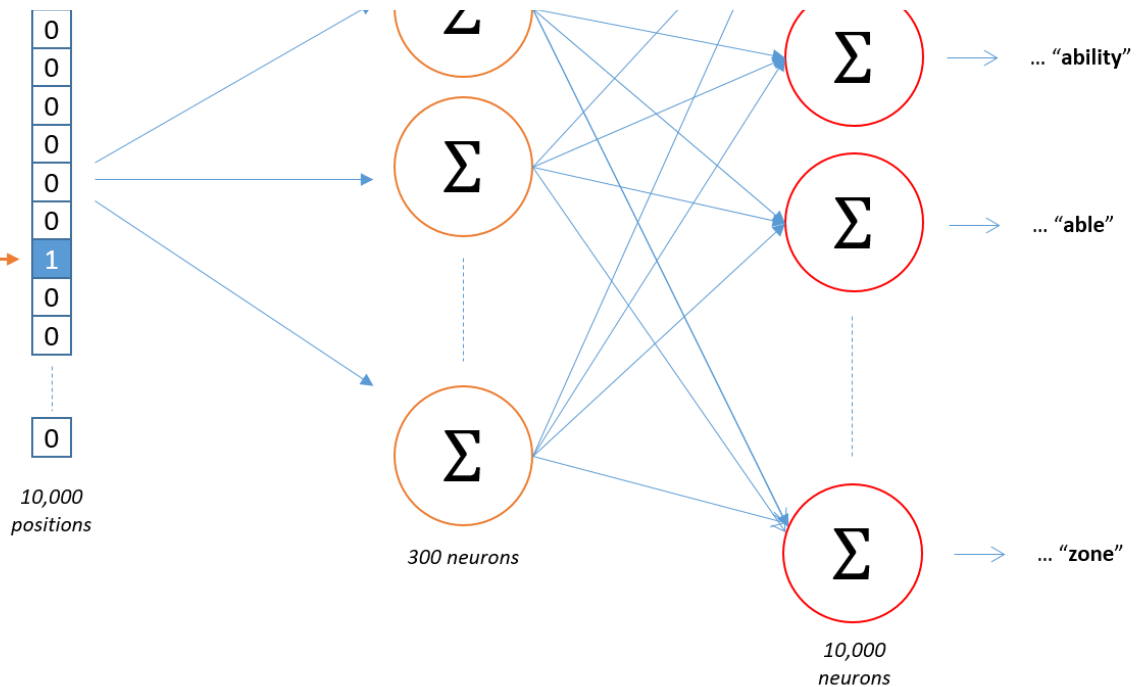
Instead, we feed words as one-hot vectors, which is basically a vector of the same length as the vocabulary, filled with zeros except at the index that represents the word we want to represent, which is assigned "1".

The hidden layer is a standard fully-connected (Dense) layer whose weights are the word embeddings.

The output layer outputs probabilities for the target words from the vocabulary.



A '1' in the position corresponding to the word "ants"



The input to this network is a one-hot vector representing the input word, and the label is also a one-hot vector representing the target word, however, the network's output is a probability distribution of target words, not *necessarily* a one-hot vector like the labels.

The rows of the hidden layer weight matrix, are actually the word vectors (word embeddings) we want!

Hidden Layer  
Weight Matrix



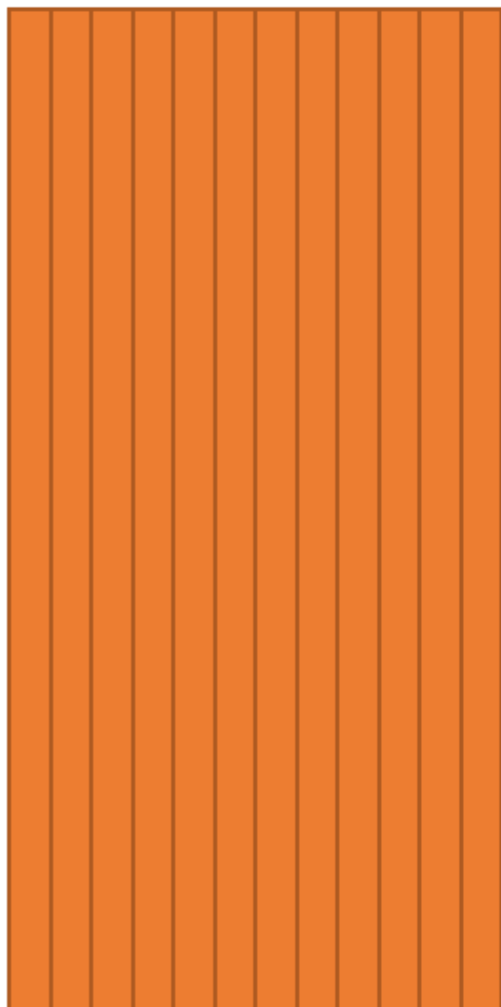
Word Vector  
Lookup Table!

300 neurons

300 features

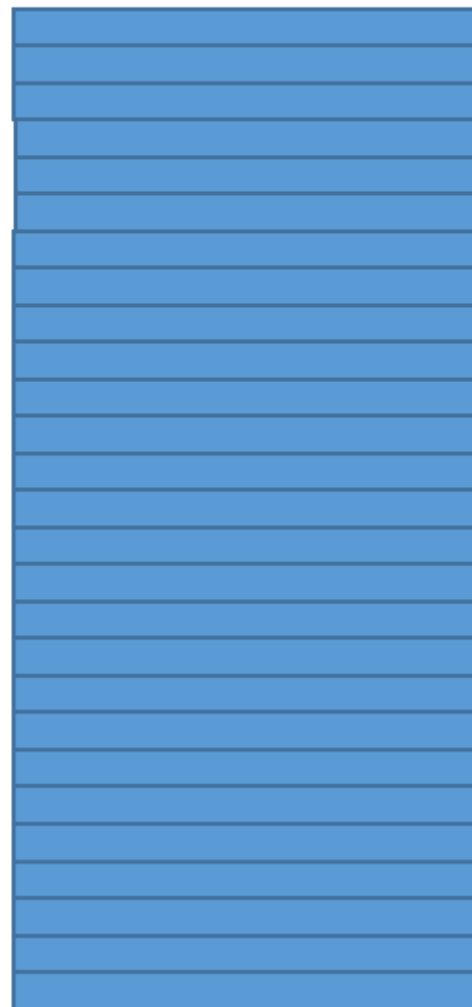
300 neurons

10,000 words



300 features

10,000 words

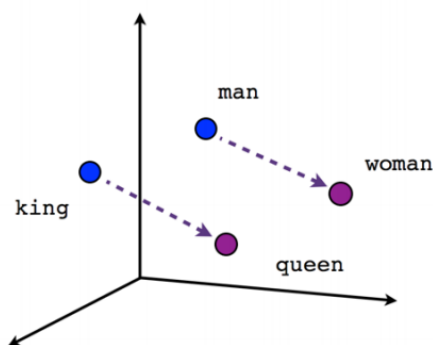


The hidden layer operates as a lookup table. The output of the hidden layer is just the “word vector” for the input word.

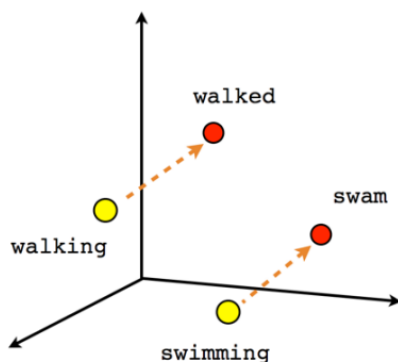
More concretely, if you multiply a  $1 \times 10,000$  one-hot vector by a  $10,000 \times 300$  matrix, it will effectively just select the matrix row corresponding to the ‘1’.

$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

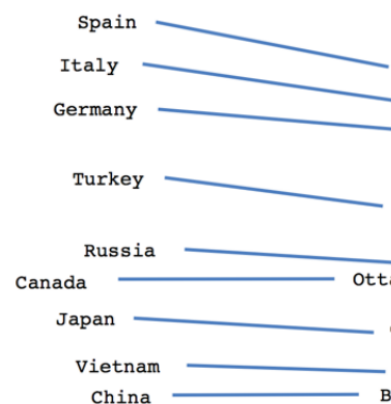
Word2Vec is able to capture multiple different degrees of similarity between words, such that semantic and syntactic patterns can be reproduced using vector arithmetic. Patterns such as “Man is to Woman as Brother is to Sister” can be generated through algebraic operations on the vector representations of these words such that the vector representation of “Brother” - “Man” + “Woman” produces a result which is closest to the vector representation of “Sister” in the model. Such relationships can be generated for a range of semantic relations (such as Country—Capital) as well as syntactic relations (e.g. present tense—past tense).



Male-Female



Verb tense

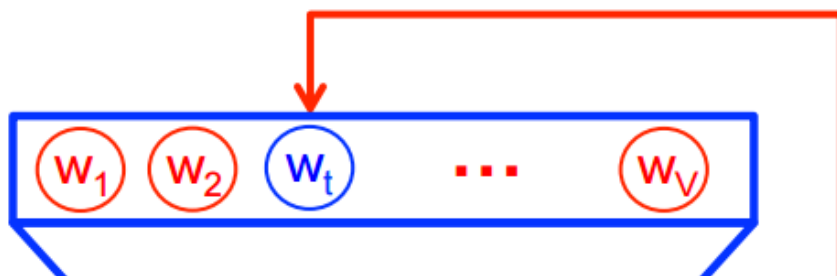


Country-Capital

## Training algorithm

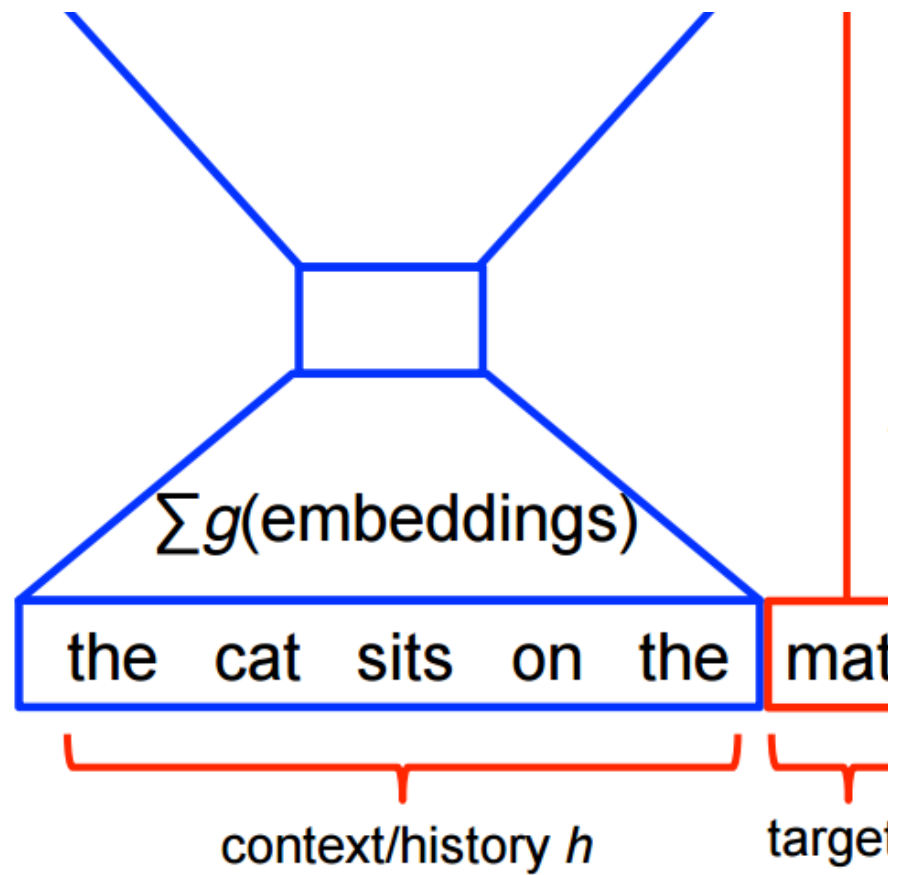
A Word2vec model can be trained with hierarchical softmax and/or negative sampling, usually, just negative sampling is used.

### Softmax classifier



Hidden layer

Projection layer



Source Text

The quick brown fox jumps over the lazy dog. ➡

Training Samples


(the, quick)  
(the, brown)

The	quick	brown	fox	jumps over the lazy dog.	→	(quick, the) (quick, brown) (quick, fox)
The	quick	brown	fox	jumps	→	(brown, the) (brown, quick) (brown, fox) (brown, jumps)
The	quick	brown	fox	jumps over	→	(fox, quick) (fox, brown) (fox, jumps) (fox, over)

## Coding Walk-through


**BASICS** [\\_\(https://pytorch.org/tutorials/beginner/nlp/word\\_embeddings\\_tutorial.html#sphx-glr-beginner-nlp-word-embeddings-tutorial-py\)](https://pytorch.org/tutorials/beginner/nlp/word_embeddings_tutorial.html#sphx-glr-beginner-nlp-word-embeddings-tutorial-py)

**SOURCE** [\\_\(https://github.com/bentrevett/pytorch-sentiment-analysis\)](https://github.com/bentrevett/pytorch-sentiment-analysis)


- 1 - **Simple Sentiment Analysis** [\\_\(https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/1%20-%20Simple%20Sentiment%20Analysis.ipynb\)](https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/1%20-%20Simple%20Sentiment%20Analysis.ipynb)  [\\_\(https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/1%20-%20Simple%20Sentiment%20Analysis.ipynb\)](https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/1%20-%20Simple%20Sentiment%20Analysis.ipynb)

This tutorial covers the workflow of a PyTorch with TorchText project. We'll learn how to: load data, create train/test/validation splits, build a vocabulary, create data iterators, define a model and

implement the train/evaluate/test loop. The model will be simple and achieve poor performance, but this will be improved in the subsequent tutorials.

- 2 - [Upgraded Sentiment Analysis](https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/2%20-%20Upgraded%20Sentiment%20Analysis.ipynb) [\\_ \(https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/2%20-%20Upgraded%20Sentiment%20Analysis.ipynb\)](https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/2%20-%20Upgraded%20Sentiment%20Analysis.ipynb)  [Open in Colab](#) [\\_ \(https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/2%20-%20Upgraded%20Sentiment%20Analysis.ipynb\)](https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/2%20-%20Upgraded%20Sentiment%20Analysis.ipynb)


Now we have the basic workflow covered, this tutorial will focus on improving our results. We'll cover: using packed padded sequences, loading and using pre-trained word embeddings, different optimizers, different RNN architectures, bi-directional RNNs, multi-layer (aka deep) RNNs and regularization.

- 3 - [Faster Sentiment Analysis](https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/3%20-%20Faster%20Sentiment%20Analysis.ipynb) [\\_ \(https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/3%20-%20Faster%20Sentiment%20Analysis.ipynb\)](https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/3%20-%20Faster%20Sentiment%20Analysis.ipynb)  [Open in Colab](#) [\\_ \(https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/3%20-%20Faster%20Sentiment%20Analysis.ipynb\)](https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/3%20-%20Faster%20Sentiment%20Analysis.ipynb)


After we've covered all the fancy upgrades to RNNs, we'll look at a different approach that does not use RNNs. More specifically, we'll implement the model from [Bag of Tricks for Efficient Text Classification](https://arxiv.org/abs/1607.01759) [\\_ \(https://arxiv.org/abs/1607.01759\)](https://arxiv.org/abs/1607.01759). This simple model achieves comparable performance as the *Upgraded Sentiment Analysis*, but trains much faster.

- 4 - [Convolutional Sentiment Analysis](https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/4%20-%20Convolutional%20Sentiment%20Analysis.ipynb) [\\_ \(https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/4%20-%20Convolutional%20Sentiment%20Analysis.ipynb\)](https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/4%20-%20Convolutional%20Sentiment%20Analysis.ipynb)  [Open in Colab](#) [\\_ \(https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/4%20-%20Convolutional%20Sentiment%20Analysis.ipynb\)](https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/4%20-%20Convolutional%20Sentiment%20Analysis.ipynb)

Next, we'll cover convolutional neural networks (CNNs) for sentiment analysis. This model will be an implementation of [Convolutional Neural Networks for Sentence Classification](https://arxiv.org/abs/1408.5882) [\\_ \(https://arxiv.org/abs/1408.5882\)](https://arxiv.org/abs/1408.5882).

- 5 - [Multi-class Sentiment Analysis](https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/5%20-%20Multi-class%20Sentiment%20Analysis.ipynb) [\\_ \(https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/5%20-%20Multi-class%20Sentiment%20Analysis.ipynb\)](https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/5%20-%20Multi-class%20Sentiment%20Analysis.ipynb)  [Open in Colab](#) [\\_ \(https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/5%20-%20Multi-class%20Sentiment%20Analysis.ipynb\)](https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/5%20-%20Multi-class%20Sentiment%20Analysis.ipynb)

Then we'll cover the case where we have more than 2 classes, as is common in NLP. We'll be using the CNN model from the previous notebook and a new dataset which has 6 classes.

- 6 - [Transformers for Sentiment Analysis](https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/6%20-%20Transformers%20for%20Sentiment%20Analysis.ipynb) [\\_ \(https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/6%20-%20Transformers%20for%20Sentiment%20Analysis.ipynb\)](https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/6%20-%20Transformers%20for%20Sentiment%20Analysis.ipynb)  [Open in Colab](#) [\\_ \(https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/6%20-%20Transformers%20for%20Sentiment%20Analysis.ipynb\)](https://colab.research.google.com/github/bentrevett/pytorch-sentiment-analysis/blob/master/6%20-%20Transformers%20for%20Sentiment%20Analysis.ipynb)

Finally, we'll show how to use the transformers library to load a pre-trained transformer model, specifically the BERT model from [this](https://arxiv.org/abs/1810.04805) [\\_ \(https://arxiv.org/abs/1810.04805\)](https://arxiv.org/abs/1810.04805) paper, and use it to provide the embeddings for text. These embeddings can be fed into any model to predict sentiment, however we use a gated recurrent unit (GRU).

## ASSIGNMENT

Practise the above 6 colab files.

Move anyone to Lambda.

EVA4P2S9

