## **NEURAL WORD EMBEDDINGS**

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 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 text (words, characters, or n-grams) are called tokens, and breaking text into such tokens is called tokenization.

All text-vectorization process 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 ot tokens and token embedding (typically used exclusively for words and word embedding).

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

Word n-grams are 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 follow set of **2-grams**:

{

"The", "The cat", "cat", "cat sat", "sat on", "on", "on 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 sequence: 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 set, not a sequence, and the general structure of the sentences os 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 ith 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 could be an entire 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 and special 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 uniquq word. Note that you don't attribute index 0 to anything.

# 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 homework']

# All printable ASCII characters

characters = string.printable

token\_index = dict(zip(range(1, len(characters) + 1), character))

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

Note that Keras has built-in utilities 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).

from keras.preprocessing.text import Tokenizer

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

# creates a tokenizer, configured to only take into account the 1000 most common words.

tokenizer = Tokenizer(num\_words=1000)

# builds the word index

tokenizer.fit\_on\_texts(samples)

# turns strings into lists of integer indices

sequences = tokenizer.texts\_to\_sequences(samples)

#you could also directly get the one-hot binary representations. Vectorization modes other than one-hot encoding are supported by this tokenizer.

one\_hot\_results = tokenizer.texts\_to\_matrix(samples, mode='binary')

# how you can recover the word index that was computed

word\_index = tokenizer.word\_index

print('Found %s unique tokens.' % len(word\_index))

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]:

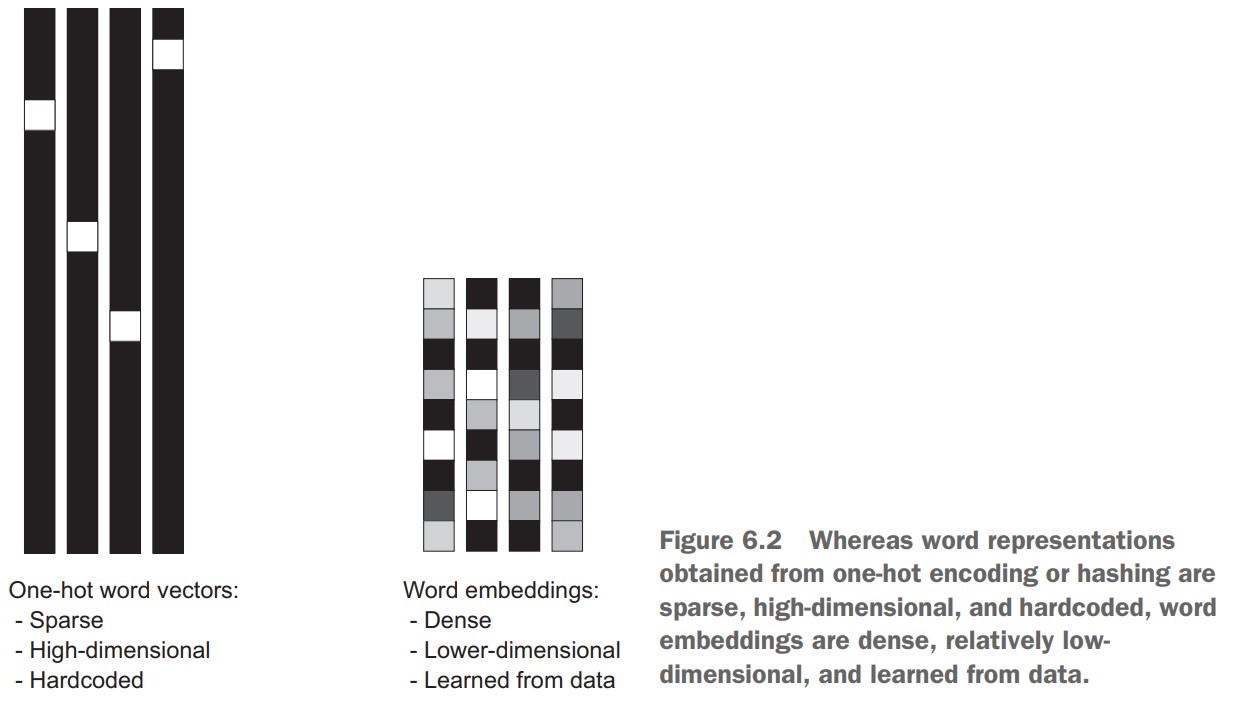
# hashes the word into a random integer index between 0 and 1000

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)



Unlike the word vectors obtained via one-hot encoding, word embeddings are learned from data. It's common to see word embedding 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 20000 tokens, in this case). So, word embeddings pack more information into far fewer dimensions.

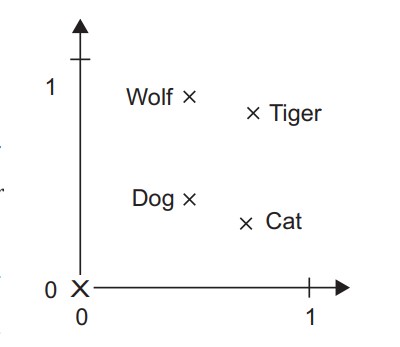
There are two ways to obtain word embeddings:

* 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 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 a 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 the 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. Fortunately, back-propagation makes that easy, and Keras makes it even easier. It is about learning the weights of a layer: the Embedding Layer.

from keras.layers import Embedding

#the embedding layer take at least two arguments: the number of possible tokens (here, 1000: 1 + max word index) and the dimensionality of the embeddings (here, 64).

embedding\_layer = Embedding(1000, 64)

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.



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) 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, it's weights (its internal dictionary of token vectors) are 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.

**Let's try it out on IMDB movie-review sentiment-prediction task.**

First, you'll quickly prepare the data. You'll restrict the movie reviews to the top 10,000 most common words and cut off the reviews after only 20 words. The network will learn 8-dimensional embeddings for each of the 10000 words, turn the input integer sequences (2D integer tensor) into embedding sequences (3D float tensor), flatten the tensor to 2D, and train a single Dense layer on top for classification.

**Loading the IMDB data for use with an Embedding layer**

from keras.datasets import imdb

from keras import preprocessing

# number of words to consider as features

max\_features = 10000

# cuts off the text after 20 number of words

maxlen = 20

# loads the data as lists of integer

(x\_train, y\_train), (x\_test, y\_test) = imdb.load\_data(num\_words=max\_features)

x\_train = preprocessing.sequence.pad\_sequences(x\_train, maxlen=maxlen)

x\_test = preprocessing.sequence.pad\_sequences(x\_text, maxlen=maxlen)

**Using an Embedding layer and classifier on the IMDB data**

from keras.models import Sequential

from keras.layers import Flatten, Dense

model = Sequential()

# specifies the max input length to the Embedding layer so you can later flatten the embedded inputs. After the Embedding layer, the activations have shape (samples, maxlen, 8)

model.add(Embedding(10000, 8, input\_length=maxlen))

# flattens the 3D tensor of embeddings into a 2D tensor of shape (Samples, maxlen\*8)

model.add(Flatten())

# adds the classifier on top

model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary\_crossentropy', metric=['acc'])

model.summary()

history = model.fit(x\_train, y\_train

epochs=10,

batch\_size=32,

validation\_split=0.2)

You get to a validation accuracy of ~76%, which is pretty good considering that you're only looking at the first 20 words in every review.

But note that merely flattening the embedded sequences and training a single Dense layer on top leads to a model that treats each word in the input sequence separately, without considering inter-word relationships and sentence structure (for example, this model would likely treat both "this movie is a bomb" and "this movie is the bomb" as being negative reviews). It's much better to add recurrent layers or 1D convolution layers on top the embedded sequences to learn features that take into account each sequence as a whole.

### **Using pretrained 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 captures 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.

**Downloading the IMDB data as raw text**

First, head to http://mng.nz/0tIo and download the raw IMDB dataset. Uncompress it. Now, let's collect the individual training reviews into a list of strings, one string per review. You'll also collect the review labels (positive/negative) into a labels list.

**Processing the labels of the raw IMDB data**

import os

imdb\_dir = '/aclImdb'

train\_dir = os.path.join(imdb\_dir, 'train')

labels = []

texts = []

for label\_type in ['neg', 'pos']:

dir\_name = os.path.join(train\_dir, label\_type)

for fname in os.listdir(dir\_name):

if fname[-4] == '.txt'

f = open(os.path.join(dir\_name, fname))

texts.append(f.read())

f.close()

if label\_type == 'neg':

labels.append(0)

else:

labels.append(1)

**Tokenizing the data**

from keras.preprocessing.text import Tokenizer

from keras.preprocessing.sequence import pad\_sequences

import numpy as np

maxlen = 100 # cuts off reviews after 100 words

training\_samples = 200 # train only on 200 samples

validation\_samples = 10000

max\_words = 10000

tokenizer = Tokenizer(num\_words=max\_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, maxlen=maxlen)

labels = np.asarray(labels)

print('Shape of data tensor:', data.shape)

print('Shape of label tensor:', labels.shape)

# Splits the data into a training set and a validation set, but first shuffles the data because you're starting with data in which samples are ordered, all negatives first, then all positives

indices = np.arange(data.shape[0])

np.random.shuffle(indices)

data = data[indices]

labels = labels[indices]

x\_train = data[:training\_samples]

y\_train = data[:training\_samples]

x\_val = data[training\_samples: training\_samples + valiation\_samples]

y\_val = labels[training\_samples: training\_samples + valiation\_samples]

**Download the Glove word embeddings**

glove\_dir = '/glove.6B'

embeddings\_index = {}

f = open(os.path.join(glove\_dir, 'glove.6B.100d.txt'))

for line in f:

values = line.split()

word = values[0]

coefs = np.assarray(values[1:], dtype='float32')

embeddings\_index[word] = coefs

f.close()

print('Found %s word vectors.' %len(embeddings\_index))

embedding\_dim = 100

embedding\_matrix = np.zeros((max\_words, embedding\_dim))

for word, i in word\_index.items():

if i < max\_words:

embedding\_vector = embedding\_index.get(word)

if embedding\_vector is not None:

embedding\_matrix[i] = embedding\_vector

# words not found in the embedding index will all be zeros

# model definition

from keras.models import Sequential

from keras.layers import Embedding, Flatten, Dense

model = Sequential()

model.add(Embedding(max\_words, embedding\_dim, input\_length=maxlen))

model.add(Flatten())

model.add(Dense(32, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

model.summary()

# load Glove model

model.layers[0].set\_weights([embedding\_matrix])

model.layers[0].trainable = False

model.compile(optimizer='rmsprop',

loss='binary\_crossentropy',

metrics=['acc'])

history = model.fit(x\_train, y\_train,

epochs = 10,

batch\_size=32,

validation\_data=(x\_val, y\_val))

model.save\_weights('pre\_trained\_glove\_model.h5')

Book:

http://faculty.neu.edu.cn/yury/AAI/Textbook/Deep%20Learning%20with%20Python.pdf