How to Use Word Embedding Layers for Deep Learning with Keras

by **Jason Brownlee** on October 4, 2017 in **Natural Language Processing**

Word embeddings provide a dense representation of words and their relative meanings.

They are an improvement over sparse representations used in simpler bag of word model representations.

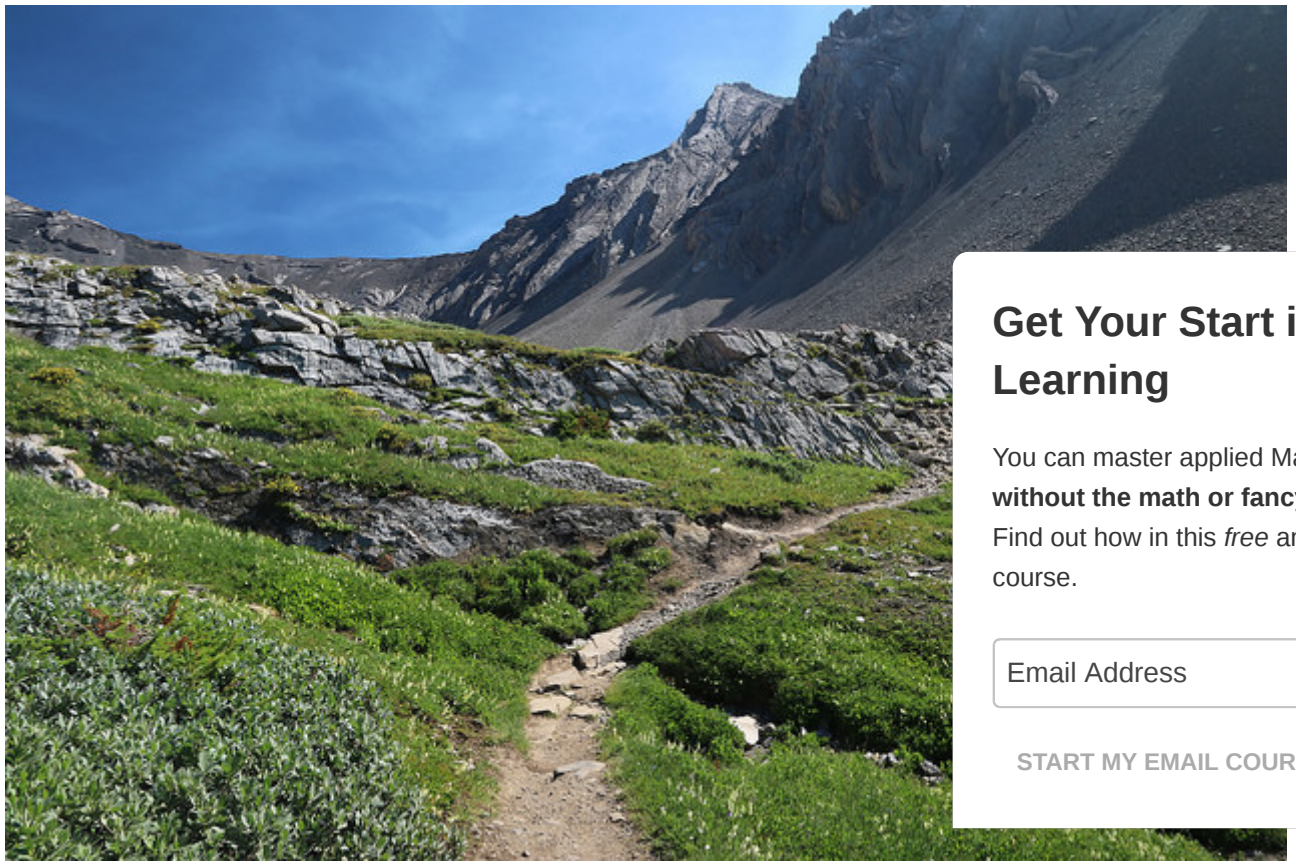
Word embeddings can be learned from text data and reused among projects. They can also be learned as part of fitting a neural network on text data.

In this tutorial, you will discover how to use word embeddings for deep learning in Python with Keras.

After completing this tutorial, you will know:

- About word embeddings and that Keras supports word embeddings via the Embedding layer.
- How to learn a word embedding while fitting a neural network.
- How to use a pre-trained word embedding in a neural network.

Let's get started.



How to Use Word Embedding Layers for Deep Learning with Keras
Photo by thisguy, some rights reserved.

Tutorial Overview

This tutorial is divided into 3 parts; they are:

Get Your Start in Machine Learning

1. Word Embedding
2. Keras Embedding Layer
3. Example of Learning an Embedding
4. Example of Using Pre-Trained GloVe Embedding

1. Word Embedding

A word embedding is a class of approaches for representing words and documents using a dense vector representation.

It is an improvement over more the traditional bag-of-word model encoding schemes where large sparse vectors were used to represent each word or to score each word within a vector to represent an entire vocabulary. These representations were sparse because the vocabularies were vast and a given word or document would be represented by a large vector comprised mostly of zero values.

Instead, in an embedding, words are represented by dense vectors where a vector represents the pr e.

The position of a word within the vector space is learned from text and is based on the words that su

The position of a word in the learned vector space is referred to as its embedding.

Two popular examples of methods of learning word embeddings from text include:

- Word2Vec.
- GloVe.

In addition to these carefully designed methods, a word embedding can be learned as part of a deep out tailors the model to a specific training dataset.

2. Keras Embedding Layer

Keras offers an Embedding layer that can be used for neural networks on text data.

It requires that the input data be integer encoded, so that each word is represented by a unique integer. This data preparation step can be performed using the Tokenizer API also provided with Keras.

Get Your Start in Machine Learning

The Embedding layer is initialized with random weights and will learn an embedding for all of the words in the training dataset.

It is a flexible layer that can be used in a variety of ways, such as:

- It can be used alone to learn a word embedding that can be saved and used in another model later.
- It can be used as part of a deep learning model where the embedding is learned along with the model itself.
- It can be used to load a pre-trained word embedding model, a type of transfer learning.

The Embedding layer is defined as the first hidden layer of a network. It must specify 3 arguments:

It must specify 3 arguments:

- **input_dim**: This is the size of the vocabulary in the text data. For example, if your data is integer encoded to values between 0-10, then the size of the vocabulary would be 11 words.
- **output_dim**: This is the size of the vector space in which words will be embedded. It defines the size of the output vectors from this layer for each word. For example, it could be 32 or 100 or even larger. Test different values for your problem.
- **input_length**: This is the length of input sequences, as you would define for any input layer of a Keras model. For example, if all of your input documents are comprised of 1000 words, this would be 1000.

For example, below we define an Embedding layer with a vocabulary of 200 (e.g. integer encoded words from 0 to 199, inclusive), a vector space of 32 dimensions in which words will be embedded, and input documents that have 50 words each.

```
1 e = Embedding(200, 32, input_length=50)
```

The Embedding layer has weights that are learned. If you save your model to file, this will include we

The output of the *Embedding* layer is a 2D vector with one embedding for each word in the input sec

If you wish to connect a *Dense* layer directly to an Embedding layer, you must first flatten the 2D output matrix to a 1D vector using the *Flatten* layer.

Now, let's see how we can use an Embedding layer in practice.

3. Example of Learning an Embedding

In this section, we will look at how we can learn a word embedding while fitting a neural network on a

We will define a small problem where we have 10 text documents, each with a comment about a piece of work a student submitted. Each text document is classified as positive “1” or negative “0”. This is a simple sentiment analysis problem.

First, we will define the documents and their class labels.

```
# define documents
docs = ['Well done!',
        'Good work',
        'Great effort',
        'nice work',
        'Excellent!',
        'Weak',
        'Poor effort!',
        'not good',
        'poor work',
        'Could have done better.']
# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
```

Next, we can integer encode each document. This means that as input the Embedding layer will hav th other more sophisticated bag of word model encoding like counts or TF-IDF.

Keras provides the one_hot() function that creates a hash of each word as an efficient integer encod hich is much larger than needed to reduce the probability of collisions from the hash function.

```
# integer encode the documents
vocab_size = 50
encoded_docs = [one_hot(d, vocab_size) for d in docs]
print(encoded_docs)
```

The sequences have different lengths and Keras prefers inputs to be vectorized and all inputs to hav s to have the length of 4. Again, we can do this with a built in Keras function, in this case the pad_sequences() function.

```
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
```

We are now ready to define our *Embedding* layer as part of our neural network model.

The *Embedding* has a vocabulary of 50 and an input length of 4. We will choose a small embedding

The model is a simple binary classification model. Importantly, the output from the *Embedding* layer will be 4 vectors of 8 dimensions each, one for each word. We flatten this to a one 32-element vector to pass on to the *Dense* output layer.

```
# define the model
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# summarize the model
print(model.summary())
```

Finally, we can fit and evaluate the classification model.

```
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

The complete code listing is provided below.

```
from keras.preprocessing.text import one_hot
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
# define documents
docs = ['Well done!',
        'Good work',
        'Great effort',
        'nice work',
        'Excellent!',
        'Weak',
        'Poor effort!',
        'not good',
        'poor work',
        'Could have done better.']
# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
# integer encode the documents
vocab_size = 50
encoded_docs = [one_hot(d, vocab_size) for d in docs]
```

```
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
# define the model
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# summarize the model
print(model.summary())
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

Running the example first prints the integer encoded documents.

```
[[6, 16], [42, 24], [2, 17], [42, 24], [18], [17], [22, 17], [27, 42], [22, 24], [49, 46,
```

Then the padded versions of each document are printed, making them all uniform length.

```
[[ 6 16  0  0]
 [42 24  0  0]
 [ 2 17  0  0]
 [42 24  0  0]
 [18  0  0  0]
 [17  0  0  0]
 [22 17  0  0]
 [27 42  0  0]
 [22 24  0  0]
 [49 46 16 34]]
```

After the network is defined, a summary of the layers is printed. We can see that as expected, the output of the Embedding layer is a 4×8 matrix and this is squashed to a 32-element vector by the Flatten layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 4, 8)              400
_________________________________________________________________
```

```
flatten_1 (Flatten)          (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 433
Trainable params: 433
Non-trainable params: 0
_________________________________________________________________
```

Finally, the accuracy of the trained model is printed, showing that it learned the training dataset perfectly (which is not surprising).

```
Accuracy: 100.000000
```

You could save the learned weights from the Embedding layer to file for later use in other models.

You could also use this model generally to classify other documents that have the same kind vocabu

Next, let's look at loading a pre-trained word embedding in Keras.

4. Example of Using Pre-Trained GloVe Embedding

The Keras Embedding layer can also use a word embedding learned elsewhere.

It is common in the field of Natural Language Processing to learn, save, and make freely available w

For example, the researchers behind GloVe method provide a suite of pre-trained word embeddings license. See:

- GloVe: Global Vectors for Word Representation

The smallest package of embeddings is 822Mb, called "*glove.6B.zip*". It was trained on a dataset of one billion tokens (words) with a vocabulary of 400 thousand words. There are a few different embedding vector sizes, including 50, 100, 200 and 300 dimensions.

You can download this collection of embeddings and we can seed the Keras *Embedding* layer with weights from the pre-trained embedding for the words in your training dataset.

This example is inspired by an example in the Keras project: pretrained_word_embeddings.py.

After downloading and unzipping, you will see a few files, one of which is "*glove.6B.100d.txt*", which contains a 100-dimensional version of the embedding.

If you peek inside the file, you will see a token (word) followed by the weights (100 numbers) on each line. For example, below are the first line of the embedding ASCII text file showing the embedding for "*the*".

```
the -0.038194 -0.24487 0.72812 -0.39961 0.083172 0.043953 -0.39141 0.3344 -0.57545 0.087459 0.28787 -0.06731 0.30906 -0.26384 -0.13231 -0.
```

As in the previous section, the first step is to define the examples, encode them as integers, then pad the sequences to be the same length.

In this case, we need to be able to map words to integers as well as integers to words.

Keras provides a Tokenizer class that can be fit on the training data, can convert text to sequences consistently by calling the *texts_to_sequences()* method on the *Tokenizer* class, and provides access to the dictionary mapping of words to integers i

```
# define documents
docs = ['Well done!',
        'Good work',
        'Great effort',
        'nice work',
        'Excellent!',
        'Weak',
        'Poor effort!',
        'not good',
        'poor work',
        'Could have done better.']
# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
# prepare tokenizer
t = Tokenizer()
t.fit_on_texts(docs)
vocab_size = len(t.word_index) + 1
# integer encode the documents
encoded_docs = t.texts_to_sequences(docs)
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
```

Next, we need to load the entire GloVe word embedding file into memory as a dictionary of word to e

```
# load the whole embedding into memory
embeddings_index = dict()
f = open('glove.6B.100d.txt')
for line in f:
    values = line.split()
    word = values[0]
    coefs = asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('Loaded %s word vectors.' % len(embeddings_index))
```

This is pretty slow. It might be better to filter the embedding for the unique words in your training data.

Next, we need to create a matrix of one embedding for each word in the training dataset. We can do that by enumerating all unique words in the *Tokenizer.word_index* and locating the embedding weight vector from the loaded GloVe embedding.

The result is a matrix of weights only for words we will see during training.

```
# create a weight matrix for words in training docs
embedding_matrix = zeros((vocab_size, 100))
for word, i in t.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

Now we can define our model, fit, and evaluate it as before.

The key difference is that the embedding layer can be seeded with the GloVe word embedding weig fore the Embedding layer must be defined with *output_dim* set to 100. Finally, we do not want to update t ve will set the *trainable* attribute for the model to be *False*.

```
e = Embedding(vocab_size, 100, weights=[embedding_matrix], input_length=4, trainable=False)
```

The complete worked example is listed below.

```
from numpy import asarray
from numpy import zeros
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
```

```
from keras.layers import Embedding
# define documents
docs = ['Well done!',
        'Good work',
        'Great effort',
        'nice work',
        'Excellent!',
        'Weak',
        'Poor effort!',
        'not good',
        'poor work',
        'Could have done better.']
# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
# prepare tokenizer
t = Tokenizer()
t.fit_on_texts(docs)
vocab_size = len(t.word_index) + 1
# integer encode the documents
encoded_docs = t.texts_to_sequences(docs)
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
# load the whole embedding into memory
embeddings_index = dict()
f = open('../glove_data/glove.6B/glove.6B.100d.txt')
for line in f:
    values = line.split()
    word = values[0]
    coefs = asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('Loaded %s word vectors.' % len(embeddings_index))
# create a weight matrix for words in training docs
embedding_matrix = zeros((vocab_size, 100))
for word, i in t.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
# define model
model = Sequential()
e = Embedding(vocab_size, 100, weights=[embedding_matrix], input_length=4, trainable=False)
model.add(e)
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
```

```
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# summarize the model
print(model.summary())
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

Running the example may take a bit longer, but then demonstrates that it is just as capable of fitting this simple problem.

```
[[6, 2], [3, 1], [7, 4], [8, 1], [9], [10], [5, 4], [11, 3], [5, 1], [12, 13, 2, 14]]

[[ 6  2  0  0]
 [ 3  1  0  0]
 [ 7  4  0  0]
 [ 8  1  0  0]
 [ 9  0  0  0]
 [10  0  0  0]
 [ 5  4  0  0]
 [11  3  0  0]
 [ 5  1  0  0]
 [12 13  2 14]]

Loaded 400000 word vectors.

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 4, 100)            1500
_________________________________________________________________
flatten_1 (Flatten)          (None, 400)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 401
=================================================================
Total params: 1,901
Trainable params: 401
Non-trainable params: 1,500
_________________________________________________________________


Accuracy: 100.000000
```


In practice, I would encourage you to experiment with learning a word embedding using a pre-trained embedding that is fixed and to learn to perform learning on top of a pre-trained embedding.

Get Your Start in Machine Learning

See what works best for your specific problem.

Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Word Embedding on Wikipedia
- Keras Embedding Layer API
- Using pre-trained word embeddings in a Keras model, 2016
- Example of using a pre-trained GloVe Embedding in Keras
- GloVe Embedding
- An overview of word embeddings and their connection to distributional semantic models, 2016
- Deep Learning, NLP, and Representations, 2014

Summary

In this tutorial, you discovered how to use word embeddings for deep learning in Python with Keras.

Specifically, you learned:

- About word embeddings and that Keras supports word embeddings via the Embedding layer.
- How to learn a word embedding while fitting a neural network.
- How to use a pre-trained word embedding in a neural network.

Do you have any questions?
Ask your questions in the comments below and I will do my best to answer.

About Jason Brownlee

Get Your Start in Machine Learning

Dr. Jason Brownlee is a husband, proud father, academic researcher, author, professional developer and a machine learning practitioner. He is dedicated to helping developers get started and get good at applied machine learning. Learn more.

View all posts by Jason Brownlee →

‹ How to Prepare Text Data for Deep Learning with Keras

How to Develop Word Embeddings in Python with Gensim ›

24 Responses to *How to Use Word Embedding Layers for Deep Learning w*

Mohammad October 4, 2017 at 7:58 am #

Thank you Jason,
I am excited to read more NLP posts.

Jason Brownlee October 4, 2017 at 8:03 am #

Thanks.

shiv October 5, 2017 at 10:07 am #

REPLY

I split my data into 80-20 test-train and I'm still getting 100% accuracy. Any idea why? It is ~99% on epoch 1 and the rest its 100%.

Jason Brownlee October 5, 2017 at 5:22 pm #

Get Your Start in Machine Learning

Consider using the procedure in this post to evaluate your model:
https://machinelearningmastery.com/evaluate-skill-deep-learning-models/

trulia October 6, 2017 at 12:47 pm #

REPLY

Use drop-out 20%, your model is overfit!!

Sandy October 6, 2017 at 2:44 pm #

REPLY

Thank you Jason. I always find things easier when reading your post.
I have a question about the vector of each word after training. For example, the word “done” in sentence
from that word in sentence “Could have done better!”. Is that right? I mean the presentation of each wor

Jason Brownlee October 7, 2017 at 5:48 am #

No, each word in the dictionary is represented differently, but the same word in different co

It is the word in its different contexts that is used to define the representation of the word.

Does that help?

Sandy October 7, 2017 at 5:37 pm #

REPLY

Yes, thank you. But I still have a question. We will train each context separately, then after training the first context, in this case is “Well done!”, we will have a vector representation of the word “done”. After training the second context, “Could have done better”, we have another vector representation of the word “done”. So, which vector will we choose to be the representation of the word “done”?
I might misunderstand the procedure of training. Thank you for clarifying it for me.

Jason Brownlee October 8, 2017 at 8:32 am #

REPLY

No. All examples where a word is used are used as part of the training of the representation of the word. There is only one representation for each word during and after training.

Sandy October 8, 2017 at 2:46 pm #

I got it. Thank you, Jason.

Chiedu October 7, 2017 at 5:36 pm #

Hi Jason,
any ideas on how to “filter the embedding for the unique words in your training data” as mentioned in the

Jason Brownlee October 8, 2017 at 8:32 am #

The mapping of word to vector dictionary is built into Gensim, you can access it directly to r
model.wv.vocab

Abbey October 8, 2017 at 2:19 am #

REPLY

Hi, Jason

Good day.

Get Your Start in Machine Learning

I just need your suggestion and example. I have two different dataset, where one is structured and the other is unstructured. The goal is to use the structured to construct a representation for the unstructured, so apply use word embedding on the two input data but how can I find the average of the two embedding and flatten it to one before feeding the layer into CNN and LSTM.

Looking forward to your response.
Regards
Abbey

Jason Brownlee October 8, 2017 at 8:40 am #

REPLY

Sorry, what was your question?

If your question was if this is a good approach, my advice is to try it and see.

Abiodun Modupe October 9, 2017 at 7:46 pm #

Hi, Jason
How can I find the average of the word embedding from the two input?
Regards
Abbey

Jason Brownlee October 10, 2017 at 7:43 am #

REPLY

Perhaps you could retrieve the vectors for each word and take their average?

Perhaps you can use the Gensim API to achieve this result?

Vinu October 9, 2017 at 5:54 pm #

Hi Jason…Could you also help us with R codes for using Pre-Trained GloVe Embedding

REPLY

Jason Brownlee October 10, 2017 at 7:43 am #

Sorry, I don't have R code for word embeddings.

REPLY

Hao October 12, 2017 at 5:49 pm #

Hi Jason, really appreciate that you answered all the replies! I am planning to try both CNN and RNN (maybe LSTM & GRU) on text classification. Most of my documents are less than 100 words long, but about 5 % are longer than 500 words. How do I set it to be 1000, will it degrade the learning result? Should I just use 100? Will it be different in the case of I set it to be 1000, will it degrade the learning result? Should I just use 100? Will it be different in the cas Thank you!

Jason Brownlee October 13, 2017 at 5:45 am #

I would recommend experimenting with different configurations and see how the impact mo

Michael October 13, 2017 at 10:22 am #

I'd like to thank you for this post. I've been struggling to understand this precise way of using keras for a week now and this is the only post I've found that actually explains what each step in the process is doing – and provides code that self-documents what the data looks like as the model is constructed and trained. This makes it so much easier to adapt to my particular requirements.

Jason Brownlee October 13, 2017 at 2:55 pm #

REPLY

Thanks, I'm glad it helped.

Azim October 17, 2017 at 5:47 pm #

REPLY

In the above Keras example, how can we predict a list of context words given a word? Lets say i have a word named 'sudoku' and want to predict the sourrounding words. how can we use word2vec from keras to do that?

Jason Brownlee October 18, 2017 at 5:32 am #

REPLY

It sounds like you are describing a language model. We can use LSTMs to learn these relat

Here is an example:
https://machinelearningmastery.com/text-generation-lstm-recurrent-neural-networks-python-keras/

Leave a Reply

Name (required)

Get Your Start in Machine Learning

Email (will not be published) (required)

Website

SUBMIT COMMENT

Welcome to Machine Learning Mastery

Hi, I'm Dr. Jason Brownlee.
My goal is to make practitioners like YOU awesome at applied machine learning.

Read More

You're a Professional (and you need results)!

The field moves quickly…
How Long Can You Afford To Wait?

Take Action Now!

GET THE TRAINING YOU NEED

Get Your Start in Machine Learning

You can master applied Machine Learning **without the math or fancy degree**.
Find out how in this *free* and *practical* email course.

Email Address

START MY EMAIL COURSE

POPULAR

Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras
JULY 21, 2016

Get Your Start in Machine Learning

Your First Machine Learning Project in Python Step-By-Step
JUNE 10, 2016

Develop Your First Neural Network in Python With Keras Step-By-Step
MAY 24, 2016

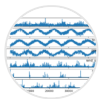
Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras
JULY 26, 2016

How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda
MARCH 13, 2017

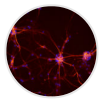
Time Series Forecasting with the Long Short-Term Memory Network in Python
APRIL 7, 2017

Multi-Class Classification Tutorial with the Keras Deep Learning Library
JUNE 2, 2016

Regression Tutorial with the Keras Deep Learning Library in Python
JUNE 9, 2016



Multivariate Time Series Forecasting with LSTMs in Keras
AUGUST 14, 2017



How to Implement the Backpropagation Algorithm From Scratch In Python
NOVEMBER 7, 2016

Get Your Start in Machine Learning

Get Your Start in Machine Learning