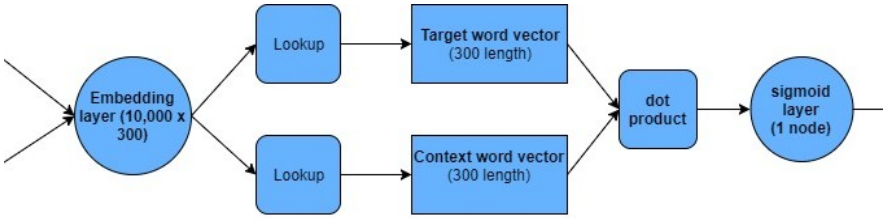
A Word2Vec Keras tutorial

August 30, 2017 Andy Deep learning, Keras, NLP, Word2Vec 1



Word2Vec Keras - negative sampling architecture

Understanding Word2Vec word embedding is a critical component in your machine learning journey. Word embedding is a necessary step in performing efficient natural language processing in your machine learning models. This tutorial will show you how to perform Word2Vec word embeddings in the Keras deep learning framework – to get an introduction to Keras, check out my tutorial (or the recommended course below). In a previous post, I introduced Word2Vec implementations in TensorFlow. In that tutorial, I showed how using a naive, softmax-based word embedding training regime results in an extremely slow training of our embedding layer when we have large word vocabularies. To get around this problem, a technique called "negative sampling" has been proposed, and a custom loss function has been created in TensorFlow to allow this (*nce_loss*).

Unfortunately, this loss function doesn't exist in Keras, so in this tutorial, we are going to implement it ourselves. This is a fortunate omission, as implementing it ourselves will help us to understand how negative sampling works and therefore better understand the Word2Vec Keras process.

POPULAR TUTORIALS

Neural Networks Tutorial – A Pathway to Deep Learning
Python TensorFlow Tutorial – Build a Neural Network
Convolutional Neural Networks Tutorial in TensorFlow
Keras tutorial – build a convolutional neural network in 11 lines
Word2Vec word embedding tutorial in Python and TensorFlow

CATEGORIES

Amazon AWS
CNTK
Convolutional Neural Networks
Deep learning
gensim
GPUs
Keras
LSTMs
Neural networks
NLP
Optimisation
PyTorch

Recommended online courses: If you'd like to dig deeper into Keras via a video course, check out this inexpensive online Udemy course: Zero to Deep Learning with Python and Keras. Also, if you'd like to do a video course in natural language processing concepts, check out this Udemy course: Natural Language Processing with Deep Learning in Python

Word embedding

If we have a document or documents that we are using to try to train some sort of natural language machine learning system (i.e. a chatbot), we need to create a vocabulary of the most common words in that document. This vocabulary can be greater than 10,000 words in length in some instances. To represent a word to our machine learning model, a naive way would be to use a one-hot vector representation i.e. a 10,000-word vector full of zeros except for one element, representing our word, which is set to 1. However, this is an inefficient way of doing things – a 10,000-word vector is an unwieldy object to train with. Another issue is that these one-hot vectors hold no information about the meaning of the word, how it is used in language and what is its usual context (i.e. what other words it generally appears close to).

Enter word embeddings – word embeddings try to "compress" large one-hot word vectors into much smaller vectors (a few hundred elements) which preserve some of the meaning and context of the word. Word2Vec is the most common process of word embedding and will be explained below.

Context, Word2Vec and the skip-gram model

The context of the word is the key measure of meaning that is utilized in Word2Vec. The context of the word "sat" in the sentence "the cat sat on the mat" is ("the", "cat", "on", "the", "mat"). In other words, it is the words which commonly occur around the *target* word "sat". Words which have similar contexts share meaning under Word2Vec, and their reduced vector representations will be similar. In the skip-gram model version of Word2Vec (more on this later), the goal is to take a *target* word i.e. "sat" and predict the surrounding context words. This involves an iterative learning process.

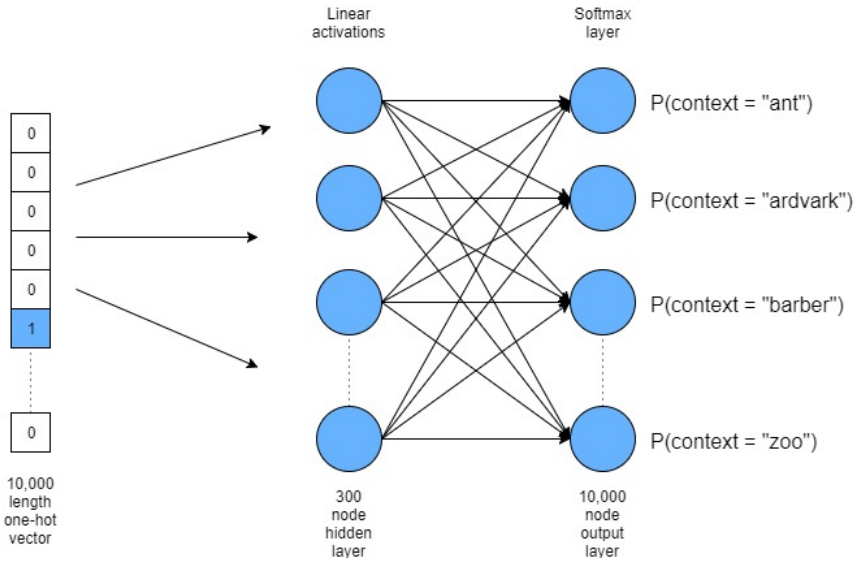
The end product of this learning will be an embedding layer in a network – this embedding layer is a kind of lookup table – the rows are vector representations of each word in our vocabulary. Here's

Recurrent neural networks
Reinforcement learning
TensorFlow
Word2Vec

a simplified example (using dummy values) of what this looks like, where *vocabulary_size=7* and *embedding_size=3*:

$anarchism$	0.5	0.1	-0.1
$originated$	-0.5	0.3	0.9
as	0.3	-0.5	-0.3
a	0.7	0.2	-0.3
$term$	0.8	0.1	-0.1
of	0.4	-0.6	-0.1
$abuse$	0.7	0.1	-0.4

As you can see, each word (row) is represented by a vector of size 3. Learning this embedding layer/lookup table can be performed using a simple neural network and an output softmax layer – see the diagram below:



A softmax trainer for word embedding

The idea of the neural network above is to supply our input *target* words as one-hot vectors. Then, via a hidden layer, we want to train the neural network to increase the probability of valid context words, while decreasing the probability of invalid context words (i.e. words that never show up in the surrounding context of the target words). This involves using a softmax function on the output

layer. Once training is complete, the output layer is discarded, and our embedding vectors are the weights of the hidden layer.

There are two variants of the Word2Vec paradigm – skip-gram and CBOW. The skip-gram variant takes a target word and tries to predict the surrounding context words, while the CBOW (continuous bag of words) variant takes a set of context words and tries to predict a target word. In this case, we will be considering the skip-gram variant (for more details – see this tutorial).

The softmax issue and negative sampling

The problem with using a full softmax output layer is that it is very computationally expensive. Consider the definition of the softmax function:

$$P(y = j \mid x) = \frac{e^{x^T w_j}}{\sum_{k=1}^{K} e^{x^T w_k}}$$

Here the probability of the output being class *j* is calculated by multiplying the output of the hidden layer and the weights connecting to the class *j* output on the numerator and dividing it by the same product but over *all the remaining weights*. When the output is a 10,000-word one-hot vector, we are talking millions of weights that need to be updated in any gradient based training of the output layer. This gets seriously time-consuming and inefficient, as demonstrated in my TensorFlow Word2Vec tutorial.

There's another solution called negative sampling. It is described in the original Word2Vec paper by Mikolov et al. It works by reinforcing the strength of weights which link a target word to its context words, but rather than reducing the value of *all* those weights which aren't in the context, it simply samples a small number of them – these are called the "negative samples".

To train the embedding layer using negative samples in Keras, we can re-imagine the way we train our network. Instead of

constructing our network so that the output layer is a multi-class softmax layer, we can change it into a simple binary classifier. For words that are in the context of the target word, we want our network to output a 1, and for our negative samples, we want our network to output a 0. Therefore, the output layer of our Word2Vec Keras network is simply a single node with a sigmoid activation function.

We also need a way of ensuring that, as the network trains, words which are similar end up having similar embedding vectors. Therefore, we want to ensure that the trained network will always output a 1 when it is supplied words which are in the same context, but 0 when it is supplied words which are never in the same context. Therefore, we need a vector similarity score supplied to the output sigmoid layer – with similar vectors outputting a high score and un-similar vectors outputting a low score. The most typical similarity measure used between two vectors is the cosine similarity score:

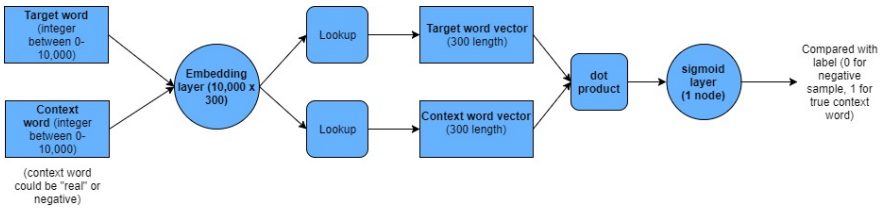
$$similarity = cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|_2 \|\mathbf{B}\|_2}$$

The denominator of this measure acts to normalize the result – the real similarity operation is on the numerator: the dot product between vectors ***A*** and ***B***. In other words, to get a simple, non-normalized measure of similarity between two vectors, you simply apply a dot product operation between them.

So with all that in mind, our new negative sampling network for the planned Word2Vec Keras implementation features:

- An (integer) input of a target word and a real or negative context word
- An embedding layer lookup (i.e. looking up the integer index of the word in the embedding matrix to get the word vector)
- The application of a dot product operation
- The output sigmoid layer

This architecture of this implementation looks like:



Word2Vec Keras – negative sampling architecture

Let's go through this architecture more carefully. First, each of the words in our vocabulary is assigned an integer index between 0 and the size of our vocabulary (in this case, 10,000). We pass two words into the network, one the target word and the other either a word from the surrounding context or a negative sample. We "look up" these indexes as the rows of our embedding layer (10,000 x 300 weight tensor) to retrieve our 300 length word vectors. We then perform a dot product operation between these vectors to get the similarity. Finally, we output the similarity to a sigmoid layer to give us a 1 or 0 indicator which we can match with the label given to the Context word (1 for a true context word, 0 for a negative sample).

The back-propagation of our errors will work to update the embedding layer to ensure that words which are truly similar to each other (i.e. share contexts) have vectors such that they return high similarity scores. Let's now implement this architecture in Keras and we can test whether this turns out to be the case.

A Word2Vec Keras implementation

This section will show you how to create your own Word2Vec Keras implementation – the code is hosted on this site's Github repository.

Data extraction

To develop our Word2Vec Keras implementation, we first need some data. As in my Word2Vec TensorFlow tutorial, we'll be using a document data set from here. To extract the information, I'll be using some of the same text extraction functions from the aforementioned Word2Vec tutorial, in particular, the *collect_data* function – check out that tutorial for further details. Basically, the function calls other functions which download the data, then a function that converts the text data into a string of integers – with each word in the vocabulary represented by a unique integer. To call this function, we run:

```
vocab_size = 10000
data, count, dictionary, reverse_dictionary = coll
```

The first 7 words in the dataset are:

['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse']
After running collect_data, the new representation of these words (*data*) is:

[5239, 3082, 12, 6, 195, 2, 3134]
There are also two dictionaries returned from *collect_data* – the first where you can look up a word and get its integer representation, and the second the reverse i.e. you look up a word's integer and you get its actual English representation.

Next, we need to define some constants for the training and also create a validation set of words so we can check the learning progress of our word vectors.

Constants and the validation set

```
window_size = 3
vector_dim = 300
epochs = 1000000

valid_size = 16     # Random set of words to evalu
valid_window = 100  # Only pick dev samples in the
valid_examples = np.random.choice(valid_window, va
```

The first constant, *window_size*, is the window of words around the target word that will be used to draw the context words from. The second constant, *vector_dim*, is the size of each of our word embedding vectors – in this case, our embedding layer will be of size 10,000 x 300. Finally, we have a large *epochs* variable – this designates the number of training iterations we are going to run. Word embedding, even with negative sampling, can be a time-consuming process.

The next set of commands relate to the words we are going to check to see what other words grow in similarity to this validation set. During training, we will check which words begin to be deemed similar by the word embedding vectors and make sure these line up with our understanding of the meaning of these words. In this case, we will select 16 words to check, and pick these words randomly from the top 100 most common words in the data-set (*collect_data* has assigned the most common words in the data set integers in ascending order i.e. the most common word is assigned 1, the next most common 2, etc.).

Next, we are going to look at a handy function in Keras which does all the skip-gram / context processing for us.

The skip-gram function in Keras

To train our data set using negative sampling and the skip-gram method, we need to create data samples for both valid context words and for negative samples. This involves scanning through the data set and picking target words, then randomly selecting context words from within the window of words around the target word (i.e. if the target word is “on” from “the cat sat on the mat”, with a window size of 2 the words “cat”, “sat”, “the”, “mat” could all be randomly selected as valid context words). It also involves randomly selecting negative samples outside of the selected target word context. Finally, we also need to set a label of 1 or 0, depending on whether the supplied context word is a true context word or a negative sample. Thankfully, Keras has a function (*skipgrams*) which does all that for us – consider the following code:

```
sampling_table = sequence.make_sampling_table(voca
couples, labels = skipgrams(data, vocab_size, wind
word_target, word_context = zip(*couples)
word_target = np.array(word_target, dtype="int32")
word_context = np.array(word_context, dtype="int32
print(couples[:10], labels[:10])
```

Ignoring the first line for the moment (*make_sampling_table)*, the Keras *skipgrams* function does exactly what we want of it – it returns the word couples in the form of (*target, context*) and also gives a matching label of 1 or 0 depending on whether *context* is a true context word or a negative sample. By default, it returns randomly shuffled *couples* and *labels*. In the code above, we then split the *couples* tuple into separate *word_target* and *word_context* variables and make sure they are the right type. The print function produces the following instructive output:

couples:

[[6503, 5], [187, 6754], [1154, 3870], [670, 1450], [4554, 1], [1037, 250], [734, 4521], [1398, 7], [4495, 3374], [2881, 8637]]
labels:

[1, 0, 1, 0, 1, 1, 0, 1, 0, 0]
The *make_sampling_table()* operation creates a table that *skipgrams* uses to ensure it produces negative samples in a balanced manner and not just the most common words.
The *skipgrams* operation by default selects the same amount of negative samples as it does true context words.

We'll feed the produced arrays (*word_target, word_context*) into our Keras model later – now onto the Word2Vec Keras model itself.

The Keras functional API and the embedding layers

In this Word2Vec Keras implementation, we'll be using the Keras functional API. In my previous Keras tutorial, I used the Keras sequential layer framework. This sequential layer framework allows the developer to easily bolt together layers, with the tensor outputs from each layer flowing easily and implicitly into the next layer. In this case, we are going to do some things which are a little tricky – the sharing of a single embedding layer between two

tensors, and an auxiliary output to measure similarity – and therefore we can't use a straightforward sequential implementation.

Thankfully, the functional API is also pretty easy to use. I'll introduce it as we move through the code. The first thing we need to do is specify the structure of our model, as per the architecture diagram which I have shown above. As an initial step, we'll create our input variables and embedding layer:

```
# create some input variables
input_target = Input((1,))
input_context = Input((1,))

embedding = Embedding(vocab_size, vector_dim, inpu
```

First off, we need to specify what tensors are going to be input to our model, along with their size. In this case, we are just going to supply individual target and context words, so the input size for each input variable is simply (1,). Next, we create an embedding layer, which Keras already has specified as a layer for us – Embedding(). The first argument to this layer definition is the number of rows of our embedding layer – which is the size of our vocabulary (10,000). The second is the size of each word's embedding vector (the columns) – in this case, 300. We also specify the input length to the layer – in this case, it matches our input variables i.e. 1. Finally, we give it a name, as we will want to access the weights of this layer after we've trained it, and we can easily access the layer weights using the name.

The weights for this layer are initialized automatically, but you can also specify an optional *embeddings_initializer* argument whereby you supply a Keras initializer object. Next, as per our architecture, we need to look up an embedding vector (length = 300) for our target and context words, by supplying the embedding layer with the word's unique integer value:

```
target = embedding(input_target)
target = Reshape((vector_dim, 1))(target)
context = embedding(input_context)
context = Reshape((vector_dim, 1))(context)
```

As can be observed in the code above, the embedding vector is easily retrieved by supplying the word integer (i.e. *input_target* and *input_context*) in brackets to the previously created *embedding* operation/layer. For each word vector, we then use a Keras *Reshape* layer to reshape it ready for our upcoming dot product and similarity operation, as per our architecture.

The next layer involves calculating our cosine similarity between the supplied word vectors:

```
# setup a cosine similarity operation which will b
similarity = merge([target, context], mode='cos',
```

As can be observed, Keras supplies a *merge* operation with a *mode* argument which we can set to 'cos' – this is the cosine similarity between the two word vectors, *target*, and *context.* This *similarity* operation will be returned via the output of a secondary model – but more on how this is performed later.

The next step is to continue on with our primary model architecture, and the dot product as our measure of similarity which we are going to use in the primary flow of the negative sampling architecture:

```
# now perform the dot product operation to get a s
dot_product = merge([target, context], mode='dot',
dot_product = Reshape((1,))(dot_product)
# add the sigmoid output layer
output = Dense(1, activation='sigmoid')(dot_produc
```

Again, we use the Keras *merge* operation and apply it to our *target* and *context* word vectors, with the *mode* argument set to 'dot' to get the simple dot product. We then do another Reshape layer, and take the reshaped dot product value (a single data point/scalar) and apply it to a Keras *Dense* layer, with the activation function of the layer set to 'sigmoid'. This is the output of our Word2Vec Keras architecture.

Next, we need to gather everything into a Keras model and compile it, ready for training:

```
# create the primary training model
model = Model(input=[input_target, input_context],
model.compile(loss='binary_crossentropy', optimize
```

Here, we create the functional API based model for our Word2Vec Keras architecture. What the model definition requires is a specification of the input arrays to the model (these need to be numpy arrays) and an output tensor – these are supplied as per the previously explained architecture. We then compile the model, by supplying a loss function that we are going to use (in this case, binary cross entropy i.e. cross entropy when the labels are either 0 or 1) and an optimizer (in this case, rmsprop). The loss function is applied to the *output* variable.

The question now is, if we want to use the *similarity* operation which we defined in the architecture to allow us to check on how things are progressing during training, how do we access it? We could output it via the *model* definition (i.e. *output*=[*similarity*, *output*]) but then Keras would be trying to apply the loss function and the optimizer to this value during training and this isn't what we created the operation for.

There is another way, which is quite handy – we create another model:

```
# create a secondary validation model to run our s
validation_model = Model(input=[input_target, inpu
```

We can now use this *validation_model* to access the *similarity* operation, and this model will actually *share* the embedding layer with the primary model. Note, because this model won't be involved in training, we don't have to run a Keras *compile* operation on it.

Now we are ready to train the model – but first, let's setup a function to print out the words with the closest similarity to our validation examples (*valid_examples*).

The similarity callback

We want to create a "callback" which we can use to figure out which words are closest in similarity to our validation examples, so we can monitor the training progress of our embedding layer.

```
class SimilarityCallback:
    def run_sim(self):
        for i in range(valid_size):
            valid_word = reverse_dictionary[valid_
            top_k = 8  # number of nearest neighbo
            sim = self._get_sim(valid_examples[i])
            nearest = (-sim).argsort()[1:top_k + 1
            log_str = 'Nearest to %s:' % valid_wor
            for k in range(top_k):
                close_word = reverse_dictionary[ne
                log_str = '%s %s,' % (log_str, clo
            print(log_str)

    @staticmethod
    def _get_sim(valid_word_idx):
        sim = np.zeros((vocab_size,))
        in_arr1 = np.zeros((1,))
        in_arr2 = np.zeros((1,))
        for i in range(vocab_size):
            in_arr1[0,] = valid_word_idx
            in_arr2[0,] = i
            out = validation_model.predict_on_batc
            sim[i] = out
        return sim
sim_cb = SimilarityCallback()
```

This class runs through all the *valid_examples* and gets the similarity score between the given validation word and all the other words in the vocabulary. It gets the similarity score by running *_get_sim*(), which features a loop which runs through each word in the vocabulary, and runs a *predict_on_batch*() operation on the validation model – this basically looks up the embedding vectors for the two supplied words (the *valid_example* and the looped vocabulary example) and returns the *similarity* operation result. The main loop then sorts the similarity in descending order and

creates a string to print out the top 8 words with the closest similarity to the validation example.

The output of this callback will be seen during our training loop, which is presented below.

The training loop

The main training loop of the model is:

```
arr_1 = np.zeros((1,))
arr_2 = np.zeros((1,))
arr_3 = np.zeros((1,))
for cnt in range(epochs):
    idx = np.random.randint(0, len(labels)-1)
    arr_1[0,] = word_target[idx]
    arr_2[0,] = word_context[idx]
    arr_3[0,] = labels[idx]
    loss = model.train_on_batch([arr_1, arr_2], ar
    if i % 100 == 0:
        print("Iteration {}, loss={}".format(cnt,
    if cnt % 10000 == 0:
        sim_cb.run_sim()
```

In this loop, we run through the total number of epochs. First, we select a random index from our *word_target, word_context* and *labels* arrays and place the values in dummy numpy arrays. Then we supply the input ([*word_target, word_context*]) and outputs (*labels*) to the primary model and run a *train_on_batch*() operation. This returns the current loss evaluation, *loss,* of the model and prints it. Every 10,000 iterations we also run functions in the SimilarityCallback.

Here are some of the word similarity outputs for the validation example word “eight” as we progress through the training iterations:

Iterations = 0:

Nearest to eight: much, holocaust, representations, density, fire, senators, dirty, fc

Iterations = 50,000:

Nearest to eight: six, finest, championships, mathematical, floor, pg, smoke, recurring

Iterations = 200,000:

Nearest to eight: six, five, two, one, nine, seven, three, four

As can be observed, at the start of the training, all sorts of random words are associated with “six”. However, as the training iterations increase, slowly other word numbers are associated with “six” until finally all of the closest 8 words are number words.

There you have it – in this Word2Vec Keras tutorial, I’ve shown you how the Word2Vec methodology works with negative sampling, and how to implement it in Keras using its functional API. In the next tutorial, I will show you how to reload trained embedding weights into both Keras and TensorFlow. You can also checkout how embedding layers work in LSTM networks in this tutorial.

« **PREVIOUS**
An introduction to TensorFlow queuing and threading

NEXT »
Python gensim Word2Vec tutorial with TensorFlow and Keras

1 COMMENT

Cognac

SEPTEMBER 20, 2017 AT 10:45 PM

at tutorial, thanks! However, since Keras has deprecated
rge" method (replaced by "Merge" layer), maybe it's
ropriate to use merge.Dot(normalize=True)?

EPLY

ave a Reply

r email address will not be published.

nment

Name*

Email*

Website

POST COMMENT

Note: some posts
contain Udemy affiliate
links