Orbifold Consulting

Embedding and Tokenizer in Keras

in Artificial Intelligence /

Keras [https://keras.io] has some classes targetting NLP and preprocessing text but it's not directly clear from the documentation and samples what they do and how they work. So I looked a bit deeper at the source code and used simple examples to expose what is going on.

You can find the Jupyter notebook here. [https://gist.github.com/Orbifold/6b0db917340086d32bfcc1f291f6a2f0]

Tokenizer

The `Tokenizer` class in Keras has various methods which help to prepare text so it can be used in neural network models.

```
from keras.preprocessing.text import Tokenizer
```

The top-n words `nb_words` will not truncate the words found in the input but it will truncate the usage. Here we take only the top three words:

```
nb_words = 3
tokenizer = Tokenizer(nb_words=nb_words)
```

The training phase is by means of the `fit_on_texts` method and you can see the word index using the `word_index` property:

```
tokenizer.fit_on_texts(["The sun is shining in June!","Se
print(tokenizer.word_index)
```

```
{'sun': 3, 'september': 4, 'june': 5, 'other': 6, 'the':
```

Note that there is a basic filtering of text annotations (exclamation marks and such).

You can see that the value 3 is clearly not respected in the sense of limiting the dictionary. It is respected however in the `texts_to_sequences` method which turns input into numerical arrays:

```
tokenizer.texts_to_sequences(["June is beautiful and I li
```

```
[[1]]
```

You need to read this as: take only words with an index less or equal to 3 (the constructor parameter). A parameter-less constructor yields the full sequences:

```
tokenizer = Tokenizer()
texts = ["The sun is shining in June!","September is grey
tokenizer.fit_on_texts(texts)
print(tokenizer.word_index)
tokenizer.texts_to_sequences(["June is beautiful and I li
```

```
{'sun': 3, 'september': 4, 'june': 5, 'other': 6, 'the':




[[5, 1, 11, 8, 14, 9, 16]]
```

There are various properties of the tokenizer which can be helpful during development of a network. For example, the stats of the training:

```
print(tokenizer.word_counts)
```

```
{'sun': 1, 'september': 1, 'june': 1, 'other': 1, 'the':
```

or whether lower-casing was applied and how many sentences were used to train:

```
print("Was lower-case applied to %s sentences?: %s"%(toke
```

```
Was lower-case applied to 5 sentences?: True
```

If you want to feed sentences to a network you can't use arrays of variable lengths, corresponding to variable length sentences. So, the trick is to use the `texts_to_matrix` method to convert the sentences directly to equal size arrays:

```
tokenizer.texts_to_matrix(["June is beautiful and I like
```

```
array([[ 0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.,  1.,  1.,
         0.,  1.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,
         0.,  0.,  1.,  0.,  0.,  0.]])
```

This creates two rows for two sentences versus the amount of words in the vocabulary.

Now you can go ahead and use networks to do stuff.

Basic network with textual data

For example, let's say you want to detect the word 'shining' in the sequences above.
The most basic way would be to use a layer with some nodes like so:

```
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten
from keras.layers.wrappers import TimeDistributed
from keras.layers.embeddings import Embedding
from keras.layers.recurrent import LSTM

X = tokenizer.texts_to_matrix(texts)
y = [1,0,0,0,0]

vocab_size = len(tokenizer.word_index) + 1
```

```
model = Sequential()
model.add(Dense(2, input_dim=vocab_size))
model.add(Dense(1, activation='sigmoid'))


model.compile(loss='binary_crossentropy', optimizer='rmsp

model.fit(X, y=y, batch_size=200, nb_epoch=700, verbose=0
```

You can check that this indeed learned the word:

```
from keras.utils.np_utils import np as np
np.round(model.predict(X))
```

```
array([[ 1.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.]], dtype=float32)
```

You can also do more sophisticated things. If the vocabulary is very large the numerical sequences turn into sparse arrays and it's more efficient to cast everything to a lower dimension with the `Embedding` layer.

Embedding

How does embedding work? An example demonstrates best what is going on.

Assume you have a sparse vector `[0,1,0,1,1,0,0]` of dimension seven. You can turn it into a non-sparse 2d vector like so:

```
model = Sequential()
model.add(Embedding(2, 2, input_length=7))
model.compile('rmsprop', 'mse')
model.predict(np.array([[0,1,0,1,1,0,0]]))
```

```
array([[[ 0.03005414, -0.02224021],
        [ 0.03396987, -0.00576888],
        [ 0.03005414, -0.02224021],
        [ 0.03396987, -0.00576888],
        [ 0.03396987, -0.00576888],
        [ 0.03005414, -0.02224021],
        [ 0.03005414, -0.02224021]]], dtype=float32)
```

Where do these numbers come from? It's a simple map from the given range to a 2d space:

```
model.layers[0].W.get_value()
```

```
array([[ 0.03005414, -0.02224021],
       [ 0.03396987, -0.00576888]], dtype=float32)
```

The 0-value is mapped to the first index and the 1-value to the second as can be seen by comparing the two arrays. The first value of the `Embedding` constructor is the range of values in the input. In the example it's 2 because we give a binary vector as input. The second value is the target dimension. The third is the length of the vectors we give.
So, there is nothing magical in this, merely a mapping from integers to floats.

Now back to our 'shining' detection. The training data looks like a sequences of bits:

```
X
```

```
array([[ 0.,  1.,  1.,  1.,  0.,  1.,  0.,  1.,  0.,  0.,
         1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  1.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,
         0.,  1.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  1.,  0.,
         0.,  0.,  0.,  0.,  0.,  1.]])
```

If you want to use the embedding it means that the output of the embedding layer will have dimension (5, 19, 10). This works well with LSTM or GRU (see below) but if you want a binary classifier you need to flatten this to (5, 19*10):

```
model = Sequential()
model.add(Embedding(3, 10, input_length= X.shape[1] ))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='rmsp
model.fit(X, y=y, batch_size=200, nb_epoch=700, verbose=0
```

It detects 'shining' flawlessly:

```
model.predict(X)
```

```
array([[  1.00000000e+00],
       [  8.39483363e-08],
       [  9.71878720e-08],
       [  7.35597965e-08],
       [  9.91844118e-01]], dtype=float32)
```

An LSTM layer has historical memory and so the dimension outputted by the embedding works in this case, no need to flatten things:

```
model = Sequential()

model.add(Embedding(vocab_size, 10))
model.add(LSTM(5))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='rmsp
model.fit(X, y=y,  nb_epoch=500, verbose=0, validation_sp
```

Obviously, it predicts things as well:

```
model.predict(X)
```

```
array([[ 0.96855599],
       [ 0.01917232],
       [ 0.01917362],
       [ 0.01917258],
       [ 0.02341695]], dtype=float32)
```

Using word2vec

There is much confusion about whether the `Embedding` in Keras is like word2vec and how word2vec can be used together with Keras. I hope that the simple example above has made clear that the `Embedding` class does indeed map discrete labels (i.e. words) into a continuous vector space. It should be just as clear that this embedding **does not** in any way take the semantic similarity of the words into account. Check the source code [https://github.com/fchollet/keras/blob/master/keras/layers/embeddings.py] if want to see it even more clearly.

So if word2vec does bring along some extra info into the game how can you use it together with Keras?

The idea is that instead of mapping sequences of integer numbers to sequences of floats happens in a way which preserves the semantic affinity. There are various pretrained word2vec datasets on the net, we'll GloVe [http://nlp.stanford.edu/projects/glove/] since it's small and straightforward but check out the Google repo [https://code.google.com/archive/p/word2vec/] as well.

Loading the GloVe set is straightforward:

```
embeddings_index = {}
glove_data = '/Users/Swa/Desktop/AIML/Glove/glove.6B.50d.
f = open(glove_data)
for line in f:
    values = line.split()
    word = values[0]
    value = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = value
f.close()

print('Loaded %s word vectors.' % len(embeddings_index))
```

```
Loaded 400000 word vectors.
```

```
embedding_dimension = 10
word_index = tokenizer.word_index
```

The `embedding_matrix` matrix maps words to vectors in the specified embedding dimension (here 100):

```
embedding_matrix = np.zeros((len(word_index) + 1, embeddi
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-
        embedding_matrix[i] = embedding_vector[:embedding
```

Now you have an embedding matrix of 19 words into dimension 10:

```
embedding_matrix.shape
```

```
(19, 10)
```

```
embedding_layer = Embedding(embedding_matrix.shape[0],
                            embedding_matrix.shape[1],
```

```
                                weights=[embedding_matrix],
                                input_length=12)
```

In order to use this new embedding you need to reshape the training data `X` to the basic word-to-index sequences:

```
from keras.preprocessing.sequence import pad_sequences
X = tokenizer.texts_to_sequences(texts)
X = pad_sequences(X, maxlen=12)
```

We have used a fixed size of 12 here but anything works really. Now the sequences with integers representing word-index are mapped to a 10-dimensional vector space using the wrod2vec embedding and we're good to go:

```
model = Sequential()
model.add(embedding_layer)
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.layers[0].trainable=False
model.compile(loss='binary_crossentropy', optimizer='rmsp
model.fit(X, y=y, batch_size=20, nb_epoch=700, verbose=0,
```

You get the same razor sharp prediction. I know all of the above networks are overkill for the simple datasets but the intention was to show you the way to use the various NLP functionalities.

```
model.predict(X)
```

```
array([[  9.98708129e-01],
       [  3.05007357e-04],
       [  1.29467447e-03],
       [  1.42437394e-03],
       [  1.05901817e-02]], dtype=float32)
```

Tags: Artificial Intelligence, Python

You might also like

Unbalan data (SMOTE)	Probabil program estimati normal mean and deviatio	Languag understa using Keras, take one	Technica recipe to embrace AI	Linear regressi TensorF vs. TFLearn	Some feedforv neural network using Keras	IMDB review classifica using Keras	Microso Concept Graph in Neo4j