Using pre-trained word embeddings in a Keras model

In this tutorial, we will walk you through the process of solving a text classification problem using pre-trained word embeddings and a convolutional neural network.

Sat 16 July 2016
By Francois Chollet
In Tutorials.

The full code for this tutorial is available on Github.

Note: all code examples have been updated to the Keras 2.0 API on March 14, 2017. You will need Keras version 2.0.0 or higher to run them.

What are word embeddings?

"Word embeddings" are a family of natural language processing techniques aiming at mapping semantic meaning into a geometric space. This is done by associating a numeric vector to every word in a dictionary, such that the distance (e.g. L2 distance or more commonly cosine distance) between any two vectors would capture part of the semantic relationship between the two associated words. The geometric space formed by these vectors is called an *embedding space*.

For instance, "coconut" and "polar bear" are words that are semantically quite different, so a reasonable embedding space would represent them as vectors that would be very far apart. But "kitchen" and "dinner" are related words, so they should be embedded close to each other.

Ideally, in a good embeddings space, the "path" (a vector) to go from "kitchen" and "dinner" would capture precisely the semantic relationship between these two concepts. In this case the relationship is "where x occurs", so you would expect the vector `kitchen - dinner` (difference of the two embedding vectors, i.e. path to go from dinner to kitchen) to capture this "where x occurs" relationship. Basically, we should have the vectorial identity: `dinner + (where x occurs) = kitchen` (at least approximately). If that's indeed the case, then we can use such a relationship vector to answer questions. For instance, starting from a new vector, e.g. "work", and applying this relationship vector, we should get sometime meaningful, e.g. `work + (where x occurs) = office`, answering "where does work occur?".

Word embeddings are computed by applying dimensionality reduction techniques to datasets of co-occurence statistics between words in a corpus of text. This can be done via neural networks (the "word2vec" technique), or via matrix factorization.

GloVe word embeddings

We will be using GloVe embeddings, which you can read about here. GloVe stands for "Global Vectors for Word Representation". It's a somewhat popular embedding technique based on factorizing a matrix of word co-occurence statistics.

Specifically, we will use the 100-dimensional GloVe embeddings of 400k words computed on a 2014 dump of English Wikipedia. You can download them here (warning: following this link will start a 822MB download).

20 Newsgroup dataset

The task we will try to solve will be to classify posts coming from 20 different newsgroup, into their original 20 categories --the infamous "20 Newsgroup dataset". You can read about the dataset and download the raw text data here.

Categories are fairly semantically distinct and thus will have quite different words associated with them. Here are a few sample categories:

```
comp.sys.ibm.pc.hardware
comp.graphics
comp.os.ms-windows.misc
comp.sys.mac.hardware
comp.windows.x

rec.autos
rec.motorcycles
rec.sport.baseball
rec.sport.hockey
```

Approach

Here's how we will solve the classification problem:

- convert all text samples in the dataset into sequences of word indices. A "word index" would simply be an integer ID for the word. We will only consider the top 20,000 most commonly occuring words in the dataset, and we will truncate the sequences to a maximum length of 1000 words.
- prepare an "embedding matrix" which will contain at index `i` the embedding vector for the word of index `i` in our word index.
- load this embedding matrix into a Keras `Embedding` layer, set to be frozen (its weights, the embedding vectors, will not be updated during training).
- build on top of it a 1D convolutional neural network, ending in a softmax output over our 20 categories.

Preparing the text data

First, we will simply iterate over the folders in which our text samples are stored, and format them into a list of samples. We will also prepare at the same time a list of class indices matching the samples:

```
texts = []  # list of text samples
labels_index = {}  # dictionary mapping label name to numeric id
labels = []  # list of label ids
for name in sorted(os.listdir(TEXT_DATA_DIR)):
    path = os.path.join(TEXT_DATA_DIR, name)
    if os.path.isdir(path):
        label_id = len(labels_index)
        labels_index[name] = label_id
        for fname in sorted(os.listdir(path)):
            if fname.isdigit():
                fpath = os.path.join(path, fname)
                if sys.version_info < (3,):
                    f = open(fpath)
                else:
                    f = open(fpath, encoding='latin-1')
                t = f.read()
                i = t.find('\n\n')  # skip header
                if 0 < i:
                    t = t[i:]
                texts.append(t)
                f.close()
                labels.append(label_id)

print('Found %s texts.' % len(texts))
```

Then we can format our text samples and labels into tensors that can be fed into a neural network. To do this, we will rely on Keras utilities `keras.preprocessing.text.Tokenizer` and `keras.preprocessing.sequence.pad_sequences`.

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

tokenizer = Tokenizer(nb_words=MAX_NB_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=MAX_SEQUENCE_LENGTH)

labels = to_categorical(np.asarray(labels))
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

# split the data into a training set and a validation set
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]
nb_validation_samples = int(VALIDATION_SPLIT * data.shape[0])

x_train = data[:-nb_validation_samples]
y_train = labels[:-nb_validation_samples]
```

```
x_val = data[-nb_validation_samples:]
y_val = labels[-nb_validation_samples:]
```

Preparing the Embedding layer

Next, we compute an index mapping words to known embeddings, by parsing the data dump of pre-trained embeddings:

```
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.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

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

At this point we can leverage our `embedding_index` dictionary and our `word_index` to compute our embedding matrix:

```
embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
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-zeros.
        embedding_matrix[i] = embedding_vector
```

We load this embedding matrix into an `Embedding` layer. Note that we set `trainable=False` to prevent the weights from being updated during training.

```
from keras.layers import Embedding

embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False)
```

An `Embedding` layer should be fed sequences of integers, i.e. a 2D input of shape `(samples, indices)`. These input sequences should be padded so that they all have the same length in a batch of input data (although an `Embedding` layer is capable of processing sequence of heterogenous length, if you don't pass an explicit `input_length` argument to the layer).

All that the `Embedding` layer does is to map the integer inputs to the vectors found at the corresponding index in the embedding matrix, i.e. the sequence `[1, 2]` would be converted to `[embeddings[1], embeddings[2]]`. This means that the output of the `Embedding` layer will be a 3D tensor of shape `(samples, sequence_length, embedding_dim)`.

Training a 1D convnet

Finally we can then build a small 1D convnet to solve our classification problem:

```
sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences = embedding_layer(sequence_input)
x = Conv1D(128, 5, activation='relu')(embedded_sequences)
x = MaxPooling1D(5)(x)
x = Conv1D(128, 5, activation='relu')(x)
x = MaxPooling1D(5)(x)
x = Conv1D(128, 5, activation='relu')(x)
x = MaxPooling1D(35)(x)  # global max pooling
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
preds = Dense(len(labels_index), activation='softmax')(x)

model = Model(sequence_input, preds)
model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['acc'])

# happy learning!
model.fit(x_train, y_train, validation_data=(x_val, y_val),
          epochs=2, batch_size=128)
```

This model reaches **95% classification accuracy** on the validation set after only 2 epochs. You could probably get to an even higher accuracy by training longer with some regularization mechanism (such as dropout) or by fine-tuning the `Embedding` layer.

We can also test how well we would have performed by not using pre-trained word embeddings, but instead initializing our `Embedding` layer from scratch and learning its weights during training. We just need to replace our `Embedding` layer with the following:

```
embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            input_length=MAX_SEQUENCE_LENGTH)
```

After 2 epochs, this approach only gets us to **90% validation accuracy**, less than what the previous model could reach in just one epoch. Our pre-trained embeddings were definitely buying us something. In general, using pre-trained embeddings is relevant for natural processing tasks were little training data is available (functionally the embeddings act as an injection of outside information which might prove useful for your model).

Powered by pelican, which takes great advantages of python.