Text preprocessing

As with all machine learning algorithms, neural networks only deal with numbers. Therefore, samples containing text need to be preprocessed into tensors of numbers. This procedure involves the following steps:

1. Break the text in the samples into smaller units (tokens) such as words, or groups of words (*n-grams*). This *tokenization* stage often involves dealing with uppercase and special characters, as well as removing stopwords.

2. Assemble the different tokens found in all the samples to a *vocabulary*. The size of the vocabulary is often limited to a certain maximum number of most frequently encountered words.

3. Associate a unique vector to each of the tokens found in the text samples. This can be done in several different ways, *e.g.* by one-hot encoding or word embedding.

Tokenization with Keras

Keras has a class `Tokenizer` for turning text samples (strings) into lists of integer indices for words. It can be given `num_words` as an argument, restricting the maximum number of words in the index (only the `num_words` most frequently occurring words in the database are then kept).

By default, the tokenization is on word level separated by " ", converts to lowercase, and involves filtering out special characters: !"#$%&()*+,./:;<=>?@[\\]^_`{|}~\t\n

Example use of Tokenizer: `samples` is a list of strings containing the original text documents, and `sequences` has the corresponding integer lists after tokenization. Finally, padding ensures that all the lists are of the same length.

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

tokenizer = Tokenizer(num_words = 10000)
tokenizer.fit_on_texts(samples)

sequences = tokenizer.texts_to_sequences(samples)

data = pad_sequences(sequences, maxlen = 200)
```

One-hot token encoding

Assume that the vocabulary of the problem has size *N*, with tokens represented by integer indices. Each token in the vocabulary is then associated with a vector of length *N* so that all its elements are zero except for the one having the corresponding index, which has value 1.

One-hot encoding is a very simple way to associate a unique vector to all the tokens. However, it also has obvious drawbacks:

- Since the vocabulary can easily be very large (even millions of tokens), the one-hot representation is sparse, high-dimensional and wasteful.

- The associated vectors carry no information concerning relationships between words. A more meaningful representation could e.g. state that the word ”accurate” is in some sense closer to the word ”precise” than the word ”banana”. There is no notion of similarity in one-hot encodings.

Both of these shortcomings can be addressed by turning to *word embeddings*.

Word embeddings

In word embedding, each word is associated with a relatively low-dimensional (much less than the dimension of the vocabulary) vector of real numbers. The obtained word vectors are situated near each other in the embedding space, if the corresponding words share some semantic meaning.

The word vectors are acquired from data as an output of a learning algorithm – either as a separate problem, or jointly with the task under consideration. The components of the vectors in embedding space are treated as trainable parameters, and are gradually changed after random initialization.

It is possible to use pretrained word embeddings. In this case the word vector representation has been computed beforehand, and is loaded to the model (e.g. Word2Vec, GloVe). Whether this approach is useful or not depends obviously on the relevance of the texts with which the embeddings have been obtained to the problem in hand.

Word embeddings with Keras

Keras has a built-in way of obtaining distributed word representations together with solving the main task; this is accomplished with an `Embedding` layer. When placed as a first layer in a network model, it takes as input the 2D integer word-index lists of shape `(samples, sequence_length)`, and outputs the corresponding 3D floating-point word vector tensors of shape `(samples, sequence_length, embedding_dimension)`for the rest of the model to process. As the training proceeds, the form of the word vectors is shaped together with the other trainable parameters in the model, in a task-specific manner.

This example model takes as an input the integer lists of samples (each 200 words long, with 10000 different words in the index), and converts the words into dense 8-dimensional floating-point vectors. The rest of the classifier (after the flattening layer) is not shown.

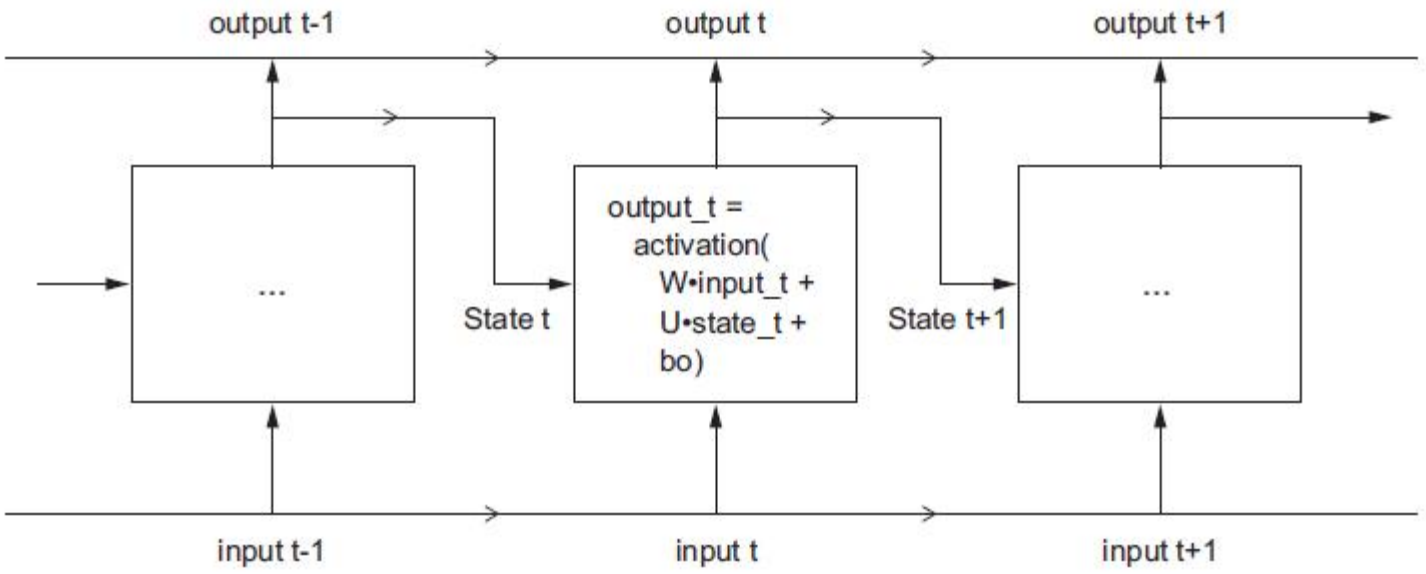
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten

model = Sequential()
model.add(Embedding(10000, 8, input_length = 200))
model.add(Flatten())
…
```

Recurrent neural networks

In Dense or ConvNet neural networks, the entire sample data is processed simultaneously (*feedforward* networks), with no intermediate states or concept of memory. However, text samples are an example of sequential data – they are an ordered sequence of words. The same applies to e.g. sensor data consisting of a time series of individual observations. *Recurrent neural networks* (RNNs) are designed for processing data in such a form.

RNN maintains a state which is updated as the input sequence is processed. The network model has a loop structure.



Simple RNN in Keras

Keras has a built-in `SimpleRNN` layer that can be added straight after an `Embedding` layer. It takes as input a batch of sequences in a form of tensor having shape `(batch_size, timesteps, input_features)`. By default, it gives an output only after the entire sequence is processed, of shape `(batch_size, input_features)`; if intermediate outputs are needed, they can be requested with argument `return_sequences = True`.

`SimpleRNN` layer is given the output feature dimension (`output_features`) as an argument.

Example of creating a simple RNN model with 32-dimensional word vectors created by the Embedding layer.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN

model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32))
…
```

From SimpleRNN to LSTM

Actually, simple RNNs are seldom useful, because they have a severe practical deficiency. As the number of timesteps grows large, the partial derivatives that must be computed by the optimizer consist of a long product of individual terms, each for a particular timestep. Such products can turn out to produce very small final values, preventing the optimization process and rendering the network untrainable. This *vanishing gradient problem* can be attacked with a more complex recurrent network structure, such as LSTM or GRU.

LSTM (Long Short Term Memory) network with a separate *carry* track introduced for temporary information storage.