Deep Neural Network for Sentiment Classification

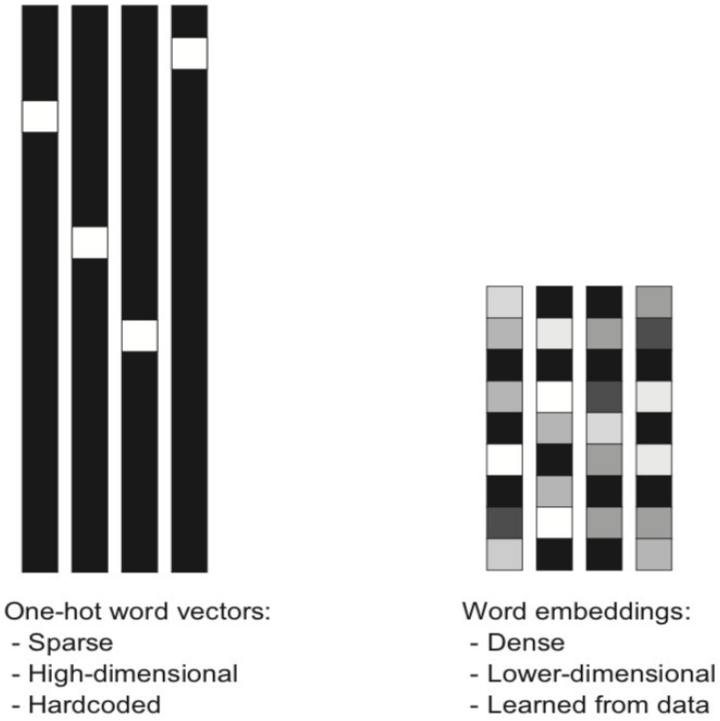
Deep learning text classification model architectures generally consist of the following components:

Input text (X) + Embeddings + Deep Neural Networks (RNN, LSTM, CNN+LSTM,..., followed by some FC layers) + Output (Y).

In this exercise, you will build different models in order to do the Sentiment Classification on the IMDB movie review dataset.

Word embeddings

Two possible ways to represent a word as a vector are one-hot encoding and word embeddings. While the vectors obtained through one-hot encoding are binary, sparse and very high-dimensional (same dimensionality as the number of words in the vocabulary), "word embeddings" are low-dimensional floating point vectors (i.e. "dense" vectors, as opposed to sparse vectors). Unlike word vectors obtained via one-hot encoding, word embeddings are learned from data. It is common to see word embeddings that are 256-dimensional, 512-dimensional, or 1024-dimensional when dealing with very large vocabularies.



Learning word embeddings with the Embedding layer

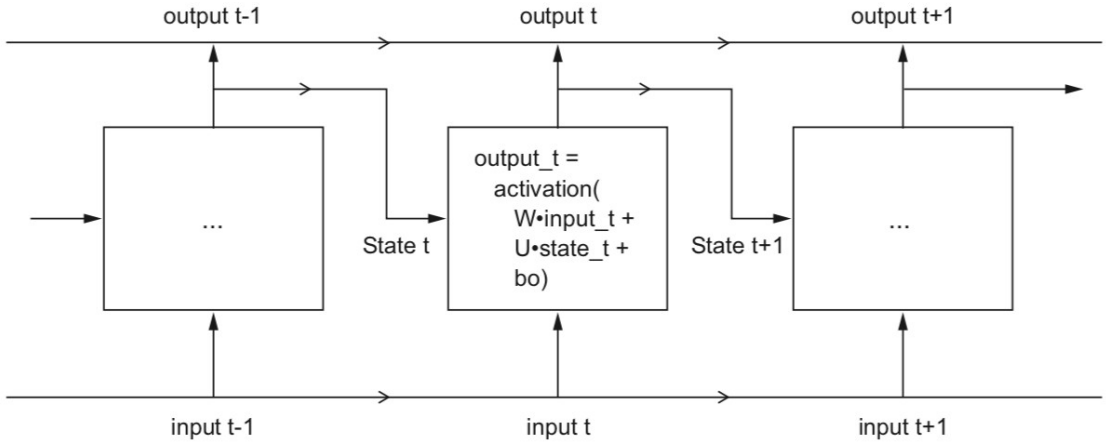
Word Embedding is a representation of text where words that have the same meaning have a similar representation. In other words it represents words in a coordinate system where related words, based on a corpus of relationships, are placed closer together. In the deep learning frameworks such as TensorFlow, Keras, this part is usually handled by an embedding layer which stores a lookup table to map the words represented by numeric indexes to their dense vector representations.

Understanding recurrent neural networks

A major characteristic of all neural networks you've seen so far, such as densely connected networks and convnets, is that they have no memory. Each input shown to them is processed independently, with no state

kept in between inputs. With such net- works, in order to process a sequence or a temporal series of data points, you have to show the entire sequence to the network at once: turn it into a single data point.

A recurrent neural network (RNN) processes sequences by iterating through the sequence elements and maintaining a state containing information relative to what it has seen so far. In effect, an RNN is a type of neural network that has an internal loop.



Each cell carries out the transformation of the input and state into an output which is parameterized by two matrices, W and U, and a bias vector.

The pseudocode for the RNN (there are two matrices and a bias vector as parameters for all cells).

```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

Let's look at a Numpy implementation of the forward pass of the simple RNN and make sure that you understand the parameter dimensions.

```
import numpy as np

timesteps = 100
input_features = 32
output_features = 64

inputs = np.random.random((timesteps, input_features))

state_t = np.zeros((output_features,))

W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))

successive_outputs = []
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)

    successive_outputs.append(output_t)

    state_t = output_t

final_output_sequence = np.concatenate(successive_outputs, axis=0)
```

Number of timesteps in the input sequence

Dimensionality of the input feature space

Dimensionality of the output feature space

Input data: random noise for the sake of the example

Initial state: an all-zero vector

Creates random weight matrices

input_t is a vector of shape (input_features,).

Stores this output in a list

Combines the input with the current state (the previous output) to obtain the current output

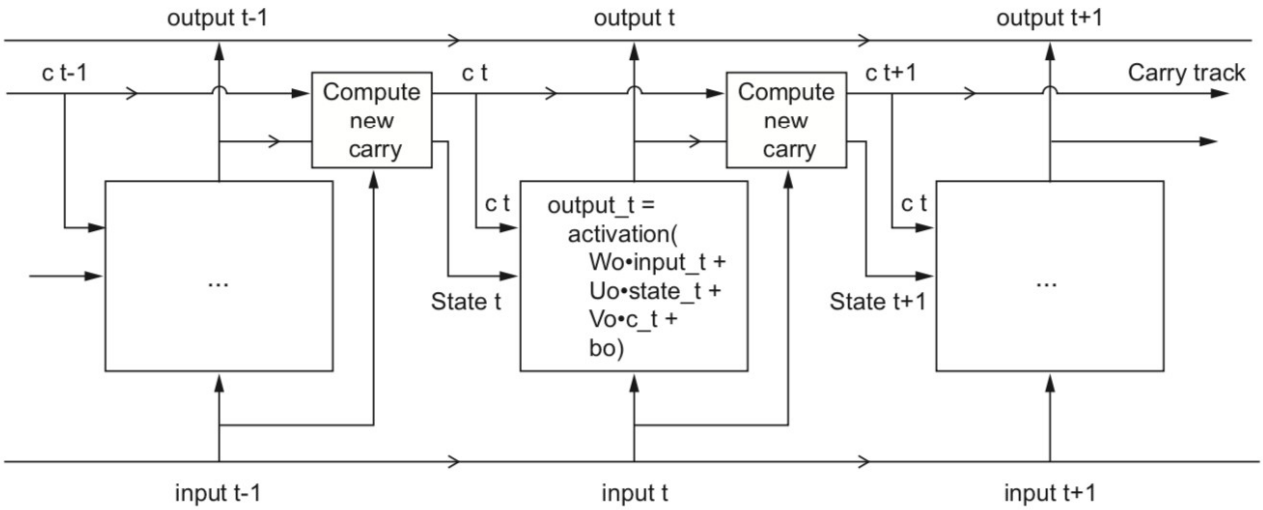
The final output is a 2D tensor of shape (timesteps, output_features).

Updates the state of the network for the next timestep

Understanding the LSTM (Long Short-Term Memory)

SimpleRNN has a major issue: although it should theoretically be able to retain at time t information about inputs seen many timesteps before, in practice, such long-term dependencies are impossible to learn. This is due to the *vanishing gradient problem.*

In comparison with simple RNN, let's add an additional data flow that carries information across time- steps. Call its values at different timesteps `Ct`, where *C* stands for *carry*. This information will have the following impact on the cell: it will be combined with the input connection and the recurrent connection (via a dense transformation: a dot product with a weight matrix followed by a bias add and the application of an activation function), and it will affect the state being sent to the next timestep (via an activation function an a multiplication operation)



In the carry dataflow, there are different “gates” to forget the irrelevant information as well as to provide the information about the present, updating the carry track with new information (with parameters to be defined).