CHAPTER 6: DEEP LEARNING FOR TEXT AND SEQUENCES

Use **recurrent neural networks** and **1D convnet**

**Applications** for these algorithms:

 Document classification and timeseries classification, such as identifying the topic of an article or the author of a book

 Timeseries comparisons, such as estimating how closely related two docu-

ments or two stock tickers are

 Sequence-to-sequence learning, such as decoding an English sentence into

French`

 Sentiment analysis, such as classifying the sentiment of tweets or movie reviews as positive or negative

 Timeseries forecasting, such as predicting the future weather at a certain location, given recent weather data

1. **Working with text data:** NN just works with numeric tensors.

2. **Vectorizing** text is the process of transforming text into numeric tensors by:

 Segment text into words, and transform each word into a vector.

 Segment text into characters, and transform each character into a vector.

 Extract n-grams of words or characters, and transform each n-gram into a vector.

N -grams are overlapping groups of multiple consecutive words or characters.

3. **Tokens => tokenization:** units being break down from text (words, characters, or n-grams). Vectorize uses some tokenization scheme, then apply numeric vectors w/ the generated tokens.

4. **Ways to associate a vector w/ a token:** many ways but now using **one-hot encoding,** and **token embedding**

5. **n-grams and bags-of-words:** word n-grams are groups of N (or fewer) consecutive words that you can extract from a sentence (can also apply to character)

Ex: 2-grams of “The cat sat on the mat.”

=> {"The", "The cat", "cat", "cat sat", "sat",

"sat on", "on", "on the", "the", "the mat", "mat"}

It may also be decomposed into the following set of 3-grams:

{"The", "The cat", "cat", "cat sat", "The cat sat",

"sat", "sat on", "on", "cat sat on", "on the", "the",

"sat on the", "the mat", "mat", "on the mat"}

They are called **bag-of-2grams** or **bag-of-3grams,** bag means order not important.

=> Structure of sentence is lost, so used for shallow models rather than dl model.

It’s a powerful feature-engineering tool when using lightweight, shallow text-processing, models such as logistic regression and random forests.

6. **One-hot encoding of words and character:** associating a unique integer index with every word and then turning this integer index I into a binary vector of size N (size of vocabulary); vector is all zeros except *i*th entry, which is 1

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

…

results[i, j, index] = 1. #i is sentence, j is each word, index is position of that word in dict

Use keras for OHE:

from keras.preprocessing.text import Tokenizer

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# We create a tokenizer, configured to only take

# into account the top-1000 most common words

**tokenizer = Tokenizer(num\_words=1000)**

# This builds the word index

**tokenizer.fit\_on\_texts(samples)**

# This turns strings into lists of integer indices.

**sequences = tokenizer.texts\_to\_sequences(samples)**

# You could also directly get the one-hot binary representations.

# Note that other vectorization modes than one-hot encoding are supported!

**one\_hot\_results = tokenizer.texts\_to\_matrix(samples, mode='binary')**

# This is how you can recover the word index that was computed

word\_index = tokenizer.word\_index

print('Found %s unique tokens.' % len(word\_index))

7. **one-hot hashing trick:** when number of unique tokens in your vocabulary is too large to handle. We can use lightweight hashing function to hash words into vectors of fixed size.

=> save memory and can do online encoding of the data (generate token vectors right away, before see all the available data).

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# We will store our words as vectors of size 1000.

# Note that if you have close to 1000 words (or more)

# you will start seeing many hash collisions, which

# will decrease the accuracy of this encoding method.

dimensionality = 1000

max\_length = 10

results = np.zeros((len(samples), max\_length, dimensionality))

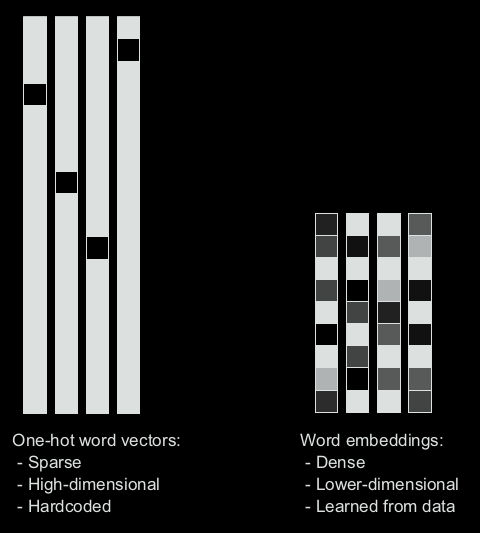
for i, sample in enumerate(samples):

for j, word in list(enumerate(sample.split()))[:max\_length]:

# Hash the word into a "random" integer index

# that is between 0 and 1000

index = abs(hash(word)) % dimensionality

 results[i, j, index] = 1.

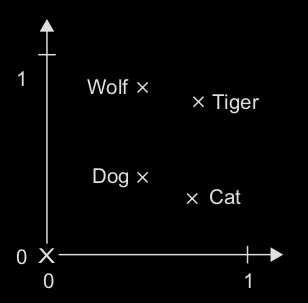
8. **Word embeddings: (or dense word vectors):** It’s common to see word embeddings that are 256-dimensional, 512-dimensional, or 1,024-dimensional when dealing with very large vocabularies. On the other hand, one-hot encoding words generally leads to vectors that are 20,000-dimensional or greater (capturing a vocabulary of 20,000 tokens, in this case). So, word embeddings pack more information into far fewer dimensions.

2 ways to do this

9. **LEARNING WORD EMBEDDINGS WITH THE EMBEDDING LAYER**

Simplest way to associate a dense vector w/ a word is to choose the vector at random. Problem w/ this approach is that the resulting embedding space has no structure. => hard for NN to make sense such a noisy, unstructured embedding space.

Geometric relationship between words (closer meaning relating to each other…)

Distance => semantic distance

In addition, want specific **direction** in the embedding space to be meaningful.

Can have vector representations (semantic relationships between words). Ex: “from pet to wild animal” vector

**In real-world** word-embedding spaces, common examples of meaningful geometric transformations are “gender” vectors and “plural” vectors. For instance, by adding a “female” vector to the vector “king,” we obtain the vector “queen.” By adding a “plural” vector, we obtain “kings.”

Word-embedding spaces typically feature thousands of such interpretable and poten-

tially useful vectors.

**It’s thus reasonable to learn a new embedding space with every new task. Fortu-**

**nately, backpropagation makes this easy, and Keras makes it even easier. It’s about**

**learning the weights of a layer: the Embedding layer.**

from keras.layers import Embedding

# The Embedding layer takes at least two arguments:

# the number of possible tokens, here 1000 (1 + maximum word index),

# and the dimensionality of the embeddings, here 64.

embedding\_layer = Embedding(1000, 64)

10. **Embedding layer is like a dictionary that map a word to its class (word vector)**

from keras.datasets import imdb

from keras import preprocessing

# Number of words to consider as features

max\_features = 10000

# Cut texts after this number of words

# (among top max\_features most common words)

maxlen = 20

# Load the data as lists of integers.

(x\_train, y\_train), (x\_test, y\_test) = imdb.load\_data(num\_words=max\_features)

# This turns our lists of integers

# into a 2D integer tensor of shape `(samples, maxlen)`

**x\_train = preprocessing.sequence.pad\_sequences(x\_train, maxlen=maxlen)**

x\_test = preprocessing.sequence.pad\_sequences(x\_test, maxlen=maxlen)

model = Sequential()

# We specify the maximum input length to our Embedding layer

# so we can later flatten the embedded inputs

**model.add(Embedding(10000, 8, input\_length=maxlen))**

# After the Embedding layer,

# our activations have shape `(samples, maxlen, 8)`.

# We flatten the 3D tensor of embeddings

# into a 2D tensor of shape `(samples, maxlen \* 8)`

model.add(Flatten())

# We add the classifier on top

model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary\_crossentropy', metrics=['acc'])

model.summary()

history = model.fit(x\_train, y\_train,

epochs=10,

batch\_size=32,

validation\_split=0.2)

Merely flattening the embedded sequences and training a single Dense layer on top leads to a model that treats each word in the input sequence separately, without considering inter-word relationships and structure sentence

11. **Using pre-trained word embeddings**

Such word embeddings are generally computed using word occurrence statistics

12. **Word2Vec algorithm** (by Mikolov at Google): It can capture specific semantic properties (ex: gender)

**GloVe** (by Stanford- Global vector for representation): based on factorizing a matrix of word co-occurence statistics. Have pre-computed embeddings for millions of English tokens.

13. **Training from scratch**

import os

imdb\_dir = '/home/khiem/Documents/smallData/khiem/aclImdb/aclImdb'

train\_dir = os.path.join(imdb\_dir, 'train')

labels = []

texts = []

for label\_type in ['neg', 'pos']:

dir\_name = os.path.join(train\_dir, label\_type)

**for fname in os.listdir(dir\_name):**

if fname[-4:] == '.txt':

f = open(os.path.join(dir\_name, fname))

texts.append(f.read())

f.close()

if label\_type == 'neg':

labels.append(0)

else:

labels.append(1)

from keras.preprocessing.text import Tokenizer

from keras.preprocessing.sequence import pad\_sequences

import numpy as np

maxlen = 100 # We will cut reviews after 100 words

training\_samples = 200 # We will be training on 200 samples

validation\_samples = 10000 # We will be validating on 10000 samples

max\_words = 10000 # We will only consider the top 10,000 words in the dataset

tokenizer = Tokenizer(num\_words=max\_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=maxlen)**

**labels = 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

# But first, shuffle the data, since we started from data

# where sample are ordered (all negative first, then all positive).

**indices = np.arange(data.shape[0])**

**np.random.shuffle(indices)**

**data = data[indices]**

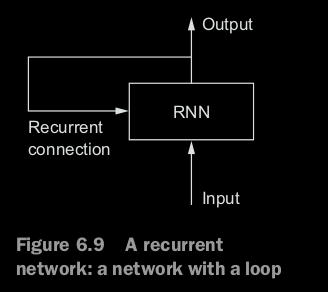
**labels = labels[indices]**

x\_train = data[:training\_samples]

y\_train = labels[:training\_samples]

x\_val = data[training\_samples: training\_samples + validation\_samples]

y\_val = labels[training\_samples: training\_samples + validation\_samples]

14. **Recurrent neural networks:** different from densely or convnet nw, RNN can have memory (remember what happen from previous layer). Avoid using feedforward nw.

Like how read text, keeping memories of what came before => learn fluid representation of meaning conveyed by this sentence.

Process sequences by iterating through the sequence elements and maintaining a **state** containing information relative to what it has seen sor far. => **Internal loop**

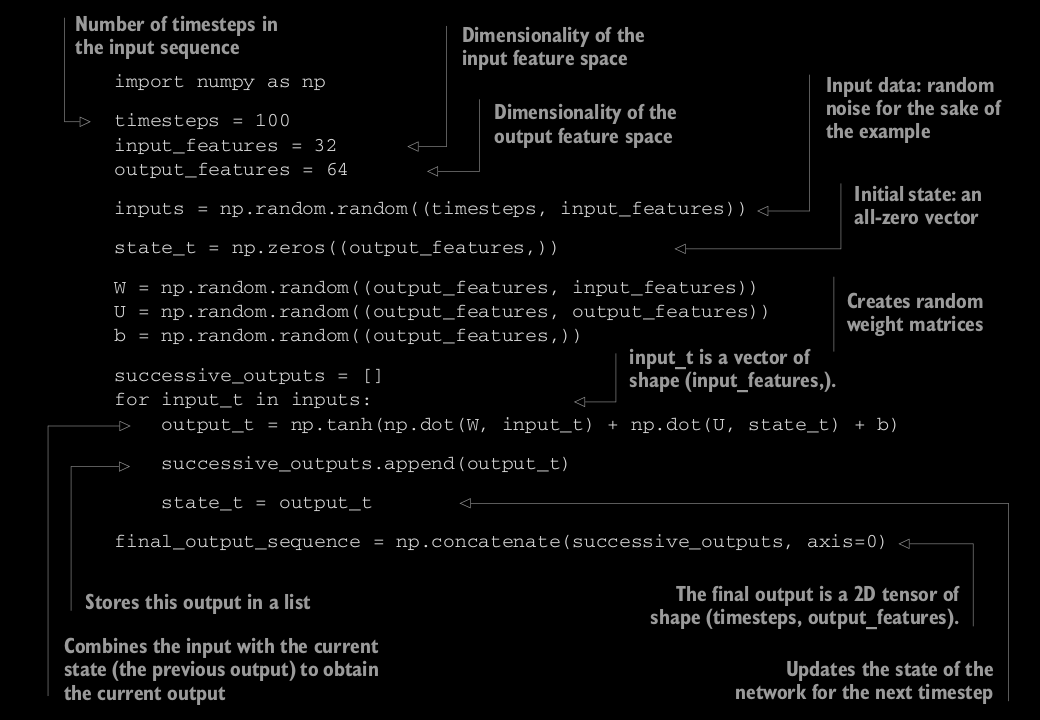
Ex: In a single review: the network internally loops over sequence elements

**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**

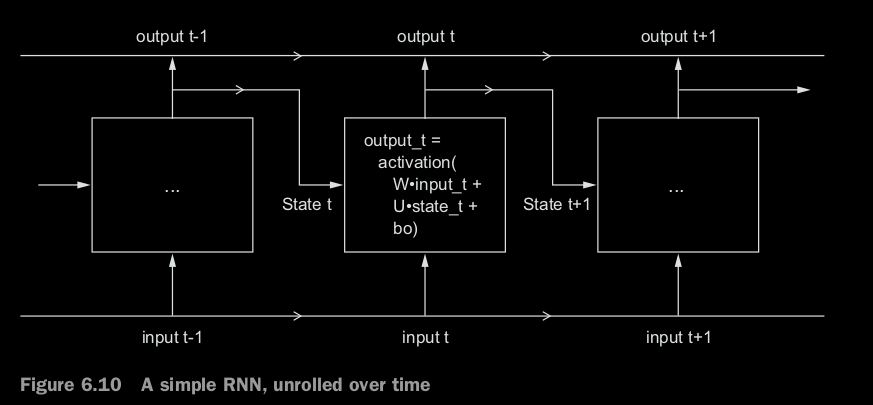
****

**np.random.random((x,y))**

**np.zeros((x,y))**

**np.concatenate(a1,a2…, axis = 0)**

an **RNN is a for loop that reuses quantities computed during the previous iteration of the loop**, nothing more. Of course, there are many different RNN s fitting this definition that you could build

=> **step function**

=> Last output contains information about the entire sequences

15. **Recurrent layer in Keras**

from keras.layers import SimpleRNN **#processes batches of sequence**

#**Take (batch\_size,timesteps,input\_features)**

2 modes: Return full sequences of output (batch, timesteps, output\_feature)

**model.add(SimpleRNN(32, return\_sequences=True))**

or just the last one (batch\_size, output\_feature)

**model.add(SimpleRNN(32))**

Can stack some full sequence layers together t

from keras.datasets import imdb

from keras.preprocessing import sequence

max\_features = 10000 # number of words to consider as features

maxlen = 500 # cut texts after this number of words (among top max\_features most common words)

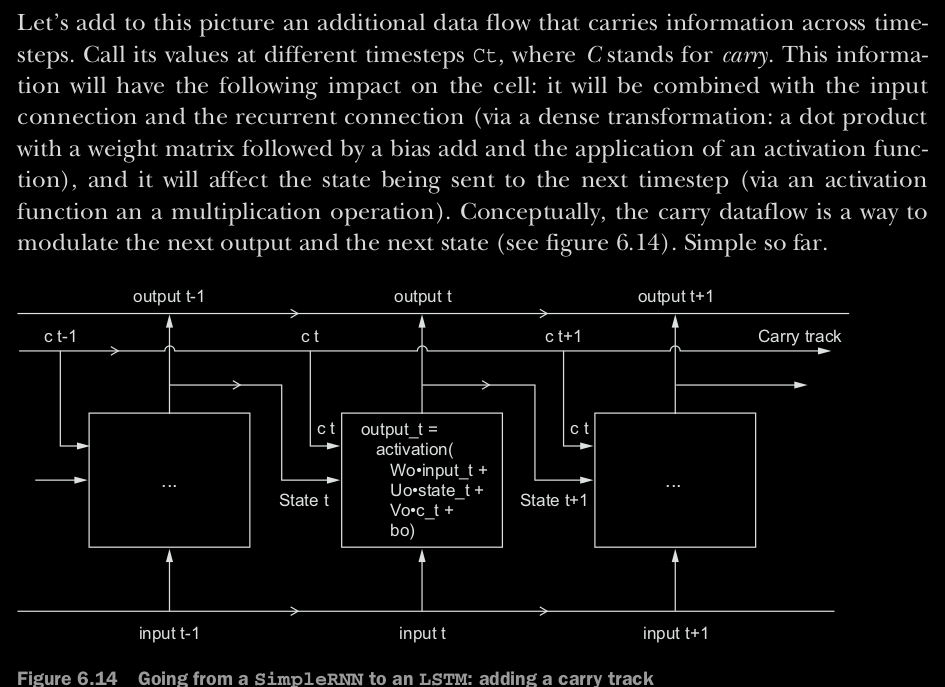
batch\_size = 32

(input\_train, y\_train), (input\_test, y\_test) = imdb.load\_data(num\_words=max\_features)

**input\_train = sequence.pad\_sequences(input\_train, maxlen=maxlen)**

**input\_test = sequence.pad\_sequences(input\_test, maxlen=maxlen)**

**16. LSTM and GRU layers**



**HOW TO COMPUTE CARRY C: by compute 3 separate SimpleRNN**

y = activation(dot(state\_t, U) + dot(input\_t, W) + b)

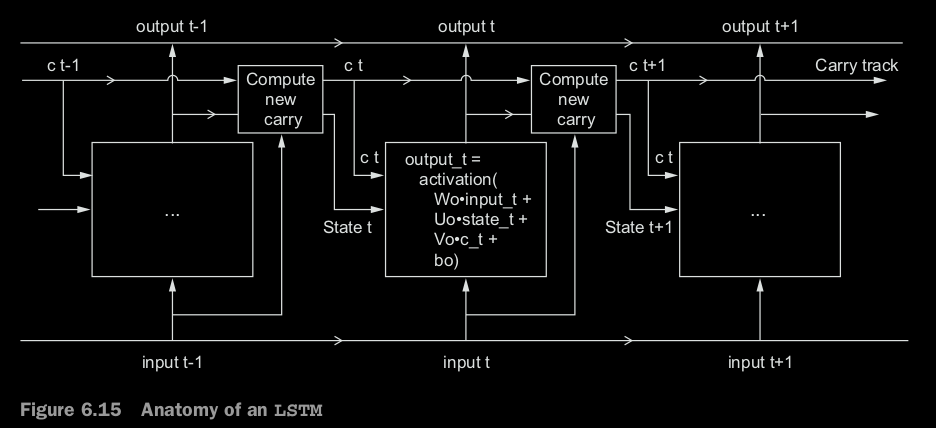
output\_t = activation(dot(state\_t, Uo) + dot(input\_t, Wo) + dot(C\_t, Vo) + bo)

i\_t = activation(dot(state\_t, Ui) + dot(input\_t, Wi) + bi)

f\_t = activation(dot(state\_t, Uf) + dot(input\_t, Wf) + bf)

k\_t = activation(dot(state\_t, Uk) + dot(input\_t, Wk) + bk)

c\_t+1 = i\_t \* k\_t + c\_t \* f\_t



The specification of an RNN cell (as just described) determines your hypoth-

esis space—the space in which you’ll search for a good model configuration during

training—but it doesn’t determine what the cell does; that is up to the cell weights. The same cell with different weights can be doing very different things. So the combination of operations making up an RNN cell is better interpreted as a set of constraints on your search, not as a design in an engineering sense.

**Practical example in Keras with LSTM**

from keras.layers import LSTM

model = Sequential()

model.add(Embedding(max\_features, 32))

model.add(LSTM(32))

model.add(Dense(1, activation='sigmoid'))

Avoid vanishing-gradient problem.

17. **Advanced use of RNN:** following techniques

- **Recurrent dropout**

**- Stacking recurrent layers**

**- Bidirectional recurrent layers**