Learning to Deep Learn using Python, Keras, TensorFlow and a GPU

Jonathon Hare, 21st Jan 2018 *(https://github.com/jonhare/DISCnetMachineLearningCourse)*

Change History

- 20180121: Initial version
- 20180416: Update for DISCnet

Introduction

In this final practical session we'll use Keras to model and analyse sequence data using recurrent neural networks made from computational blocks called a "Long Short Term Memory", or LSTM. In the first part of the tutorial we'll explore how we can predict language - given a starting character, can we predict what will come next? We'll start by implementing a simple "1st-order Markov Chain" to learn the transisition probabilities between characters, and we'll then compare this to a model that can learn longer-term dependencies using a recurrent neural network.

The second part will look at sequence classification. Sequence classification is a predictive modeling problem where you have some sequence of inputs over space or time and the task is to predict a category for the sequence. What makes this problem difficult is that the sequences can vary in length, be comprised of a very large vocabulary of input symbols and may require the model to learn the long-term context or dependencies between symbols in the input sequence. We've already explored how we can overcome this problem using Bag of Word approaches, but we've also seen that BoWs have limitations because they ignore word order. N-grams were suggested as an alternative, but they have their own problems with feature explosion. In this exercise, you will discover how you can overcome these problems by developing LSTM recurrent neural network models for sequence classification problems.

Through this part of the tutorial you'll learn how to:

- How to learn a language model using a recurrent network & to sample the model to generate new language.
- How to use callbacks during training to monitor progress.
- How to develop an LSTM model for a sequence classification problem.

Acknowledgements

The LSTM-based Nietzsche generator described in the first part of the tutorial comes from the Keras examples. The second part of this tutorial is largely based on the first section of Jason Brownlee's "Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras" (https://machinelearningmastery.com/sequence-classification-lstm-recurrent-neural-networks-python-keras/) tutorial.

Prerequisites

As with part 1 of the tutorial, you'll use Python 3 language the `keras`. We'll also again be using the `scikit-learn` and `numpy` packages.

You'll need access to a computer with the following installed:

- Python (> 3.6)
- keras (>= 2.0.0)
- tensorflow (>= 1.0.0)
- NumPy (>= 1.12.1)

- SciPy (>= 0.19.1)
- scikit-learn (>= 0.19.1)
- pillow (>=4.0.0)

Modelling sequences

Markov chains

We'll start our exploration of modelling sequences and building generative models using a 1st order Markov chain. The Markov chain is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. In our case we're going to learn a model over a set of characters from an English language text. The events, or states, in our model are the set of possible characters, and we'll learn the probability of moving from one character to the next.

Let's start by loading the data from the web:

```
from keras.utils.data_utils import get_file
import numpy as np
import random
import sys
import io

# Read the data
path = get_file('nietzsche.txt', origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
text = io.open(path, encoding='utf-8').read().lower()
print('corpus length:', len(text))
```

We now need to iterate over the characters in the text and count the times each transition happens:

```
transition_counts = dict()
for i in range(0,len(text)-1):
    currc = text[i]
    nextc = text[i+1]
    if currc not in transition_counts:
        transition_counts[currc] = dict()
    if nextc not in transition_counts[currc]:
        transition_counts[currc][nextc] = 0
    transition_counts[currc][nextc] += 1
```

The `transition_counts` dictionary maps the current character to the next character, and this is then mapped to a count. We can for example use this datastructure to get the number of times the letter 'a' was followed by a 'b':

```
print("Number of transitions from 'a' to 'b': " + str(transition_counts['a']['b']))
```

which, if we run the code at this point should print:

```
Number of transitions from 'a' to 'b': 813
```

Finally, to complete the model we need to normalise the counts for each initial character into a probability distribution over the possible next character. We'll slightly modify the form we're storing these and maintain a pair of array objects for each initial character: the first holding the set of possible characters, and the second holding the

corresponding probabilities:

```
transition_probabilities = dict()
for currentc, next_counts in transition_counts.items():
    values = []
    probabilities = []
    sumall = 0
    for nextc, count in next_counts.items():
        values.append(nextc)
        probabilities.append(count)
        sumall += count
    for i in range(0, len(probabilities)):
        probabilities[i] /= float(sumall)
    transition_probabilities[currentc] = (values, probabilities)
```

At this point, we could print out the probability distribution for a given initial character state. For example, to print the distribution for ‘a’:

```
for a,b in zip(transition_probabilities['a'][0], transition_probabilities['a'][1]):
    print(a,b)
```

The output should look like this:

```
c 0.03685183172083922
t 0.147217088814001530.05296771388194369
n 0.2322806826829003
l 0.11552886183280792
r 0.08794434177628004
s 0.0968583541689314
v 0.0192412218719426
i 0.03402543754755952
d 0.026986628981411024
g 0.017202956843135123
y 0.02505707142080661
k 0.012827481247961734
b 0.02209479291227307
p 0.020545711490379388
m 0.02030111968692249
u 0.011414284161321883
f 0.004429829329274921
w 0.004837482335036417
, 0.0010870746820306554

 0.005353842809000978
z 0.0006522448092183933
x 0.0007609522774214588
o 0.0005435373410153277
. 0.000489183606913795
- 0.0004348298728122622
' 5.4353734101532776e-05
j 0.0004348298728122622
h 0.00035329927165996303
e 0.0007337754103706925
: 5.4353734101532776e-05
a 5.4353734101532776e-05
) 0.00010870746820306555
! 2.7176867050766388e-05
; 2.7176867050766388e-05
" 8.153060115229916e-05
q 2.7176867050766388e-05
_ 8.153060115229916e-05
[ 2.7176867050766388e-05
```

It looks like the most probable letter to follow an ‘a’ is ‘n’.

We mentioned earlier that the Markov model is generative. This means that we can draw samples from the distributions and iteratively move between states. This can generate text (starting here from a ‘t’):

```
# sample
current = 't'
for i in range(0, 1000):
    print(current, end='')
    values, probabilities = transition_probabilities[current]
    current = np.random.choice(values, p=probabilities)
```

Running this should generate some text:

```
thy
chonevese juanongexpstiserorrean s antond f atomoulthed ng ives re whe ofig stouc thevey f esio daron ug nrulint se ag end tieve tepe ve u
gis wintas od crtire, in miowaver, ting trypal utid cut  the s t athet, ckecthictr rak orlvencowhe
s par.
mosomunurrtontimas ply by
apedl ws n onchanese intomes whution, mergerinse pr f-nces: irget ad
msieedis, wnts oflerer hevecy o ch
(ausa ilf psherit ther jend; nd f soto vesay heshtugis on gutf

io seted, t ise woly obase anconarored a akighemoind theentheh sthystimathevabysewin thesthe
t o owere
fofifithecofffe bof al

chouereste geapunin, thoneyeresthade iotsemassct, icthino tuphelite-and enoninse de
rerche mabo, t h id tsteavinablyphd ong.-wang cy on andior
of me haben).
anatos, thins
on pouce thimendind, pe sthe quche wo s lere tog akalin engs].
n sor cheokisunly: osse dla outhes, be t triksthithe
isttatonnnf s kinsule of e

mesif st nar to
erof himasthiouspud simofe, harnt dsous. ad ikimofrt lienderer, k oe toneser, dg n be tbe ced tot
pl
```

This is clearly not English, but it's obvious that some of the common structures in the English language have been captured.

> *Exercise: Rather than building a model based on individual characters, can you modify the model to work on words instead?*

RNN-based sequence modelling

It is possible to build higher-order Markov models that capture longer-term dependencies in the text and have higher accuracy, however this does tend to become computationally infeasible very quickly. Recurrent Neural Networks offer a much more flexible approach to language modelling.

To get started, as with out Markov model we need to start by loading the data:

```
from keras.callbacks import LambdaCallback
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.layers import LSTM
from keras.optimizers import RMSprop
from keras.utils.data_utils import get_file
import numpy as np
import random
import sys
import io

# Read the data
path = get_file('nietzsche.txt', origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
text = io.open(path, encoding='utf-8').read().lower()
print('corpus length:', len(text))
```

We'll need to create mappings of characters to numeric indices (and vice-versa) in order to perform the one-hot encoding of each character:

```
chars = sorted(list(set(text)))
print('total chars:', len(chars))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))
```

We now need to prepare the data into shorter subsequences that we can use to train the model. We'll make these redundant by overlapping them. Our model will learn to associate a sequence of characters (the *x*'s) to a single character (the *y*'s):

```
# cut the text in semi-redundant sequences of maxlen characters
maxlen = 40
step = 3
sentences = []
next_chars = []
for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i: i + maxlen])
    next_chars.append(text[i + maxlen])
print('nb sequences:', len(sentences))
```

The final step is to apply our character mapping to each subsequence and one-hot encode the data:

```
print('Vectorization...')
x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char_indices[char]] = 1
    y[i, char_indices[next_chars[i]]] = 1
```

We can now define the model. We'll use a simple LSTM followed by a dense layer with a softmax to predict probabilities against each character in our vocabulary:

```
# build the model: a single LSTM
print('Build model...')
model = Sequential()
model.add(LSTM(128, input_shape=(maxlen, len(chars))))
model.add(Dense(len(chars)))
model.add(Activation('softmax'))

optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

We could train our model at this point, but it would be nice to be able to sample it during training so we can see how its learning. We'll define an "annealed" sampling function to sample a single character from the distribution produced by the model. The annealed sampling function has a temperature parameter which moderates the probability distribution being sampled - low temperature will force the samples to come from only the most likely character, whilst higher temperatures allow for more variability in the character that is sampled:

```
def sample(preds, temperature=1.0):
    # helper function to sample an index from a probability array
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Keras lets us define 'hooks' or callbacks which can be triggered during training (for example at the end of each epoch). Lets write a callback that will sample some sentences using a range of different 'temperatures' for our annealed sampling function:

```
def on_epoch_end(epoch, logs):
    # Function invoked at end of each epoch. Prints generated text.
    print()
    print('----- Generating text after Epoch: %d' % epoch)

    start_index = random.randint(0, len(text) - maxlen - 1)
    for diversity in [0.2, 0.5, 1.0, 1.2]:
        print('----- diversity:', diversity)

        generated = ''
        sentence = text[start_index: start_index + maxlen]
        generated += sentence
        print('----- Generating with seed: "' + sentence + '"')
        sys.stdout.write(generated)

        for i in range(400):
            x_pred = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(sentence):
                x_pred[0, t, char_indices[char]] = 1.

            preds = model.predict(x_pred, verbose=0)[0]
            next_index = sample(preds, diversity)
            next_char = indices_char[next_index]

            generated += next_char
            sentence = sentence[1:] + next_char

            sys.stdout.write(next_char)
            sys.stdout.flush()
        print()

print_callback = LambdaCallback(on_epoch_end=on_epoch_end)
```

Finally we can train the model:

```
model.fit(x, y,
          batch_size=128,
          epochs=60,
          callbacks=[print_callback])
```

Running the code will produce output like the following:

```
Using TensorFlow backend.
corpus length: 600893
```

total chars: 57
nb sequences: 200285
Vectorization...
Build model...
Epoch 1/60
name: GeForce GTX TITAN X
major: 5 minor: 2 memoryClockRate (GHz) 1.076
pciBusID 0000:02:00.0
Total memory: 11.92GiB
Free memory: 385.50MiB
200192/200285 [==============================>.] - ETA: 0s - loss: 1.9911
----- Generating text after Epoch: 0
----- diversity: 0.2
----- Generating with seed: "olitude; in his strongest words, even in"
olitude; in his strongest words, even in the soched and the sount of the sore of the sight of the so the segratic upon the sounting and the self-even the sorting the so have sourther the something and the so the sounting the sortance and the so fact in the self-present and the entingly and the so findent that the
something the something of the something the self-precenting one the sore of the sould of the self-enoughther of the self-con
----- diversity: 0.5
----- Generating with seed: "olitude; in his strongest words, even in"
olitude; in his strongest words, even in the something in the endordly and stringled in the outher to has he becoungs and the sigin of the so re of a still the so without as the sich has he herther he however and the sensity and the weach of is the intinle of the prosention and oftance in their tay has atianly the fitht of sectly in any religion to hand can when when as to the outht with the ent and the sentume the strangatical to the so
----- diversity: 1.0
----- Generating with seed: "olitude; in his strongest words, even in"
olitude; in his strongest words, even in leattured to gable or elrogacal to-stranline,
by the prive by etrests
exiltivesan. everywent liken
heant weth the mouting
which sinch, conceptionory to to to , the
in aresus and beunimanly beturally suck
more
of imponsies freatack of which also
the wetu canies, the ,
when these noulble, they eastonce prictian
still, a gart been
our tombonge, cistentated in this sopess. the
then dircapts.

antace
----- diversity: 1.2
----- Generating with seed: "olitude; in his strongest words, even in"
olitude; in his strongest words, even in these vonatingles histyeuse--thas esters"-gyrty; he n reliend ischilal; "that is in a how patking it our doschys, in stren irmig-to in
espt?
obe"daday unevirti
al simallicarns only excuanity. the it is however it.

the creees of anti2liok,
hist those for the ienested of
the detiwly, a lingerse and womannie scsys;ite as reac, astroutuln?edyidity.
cacturelyce
fover myst.
this owinglusic in rin

...

Epoch 10/60
200192/200285 [==============================>.] - ETA: 0s - loss: 1.3943
----- Generating text after Epoch: 9
----- diversity: 0.2
----- Generating with seed: "music. but with regard to robert schuman"
music. but with regard to robert schuman to the seems to the sentiments of the same proper to the sentiments and among the same prospons of the same philosopher and the most say, which the first to the same freedom of the sentiments of the stands and a strength, and the man a

nd the sentiment of the soul of the same time of the same philosophers and delight of the sentiments of the sentiments of the sentiment of t
he same man and strength
----- diversity: 0.5
----- Generating with seed: "music. but **with** regard to robert schuman"
music. but with regard to robert schuman
be species and and invented and men as the
the prose of the early one should be a most has prouds to our coles and to the absolute it is all the most religion, of its experience, of the so
ul with the extended in the powerful, the order of a fruets of self-consequences of all the stain himself and preserved the extendance, and
interpered to be be a period of false and whole the stable and conseque
----- diversity: 1.0
----- Generating with seed: "music. but **with** regard to robert schuman"
music. but with regard to robert schumanly
to pleasard for-ledist is to
nobles therely
gradually as remath a conditions," whether
count is, as every pointed does not all as uniteing ent as there his empentaining from it, it was as heregation. leays the
religious old heartstop ailly-wensable morald
more pre"ful man to be the christeal consequence, no powers of sortable would really isw. cimilical cost account.

religions how hhad. t
----- diversity: 1.2
----- Generating with seed: "music. but **with** regard to robert schuman"
music. but with regard to robert schumanceophists," ones that **for** certaync-valk would, **for** eduming spirituals from loftiess, re-our higher
experiences, hissically sutklio**: if** you
jusk be a henceress givery, **else** is they serve apparered jane-man, just. man-freeds of **!**

yo wotl
plowe. every acted **with** the musimfured, must allopes,
one.

urorange sometempimal as treeess something and interf upon used **for this**
preseme
defouncious,
conhol

Looking at the results its possible to see the model works a bit like the Markov chain at the first epoch, but as the parameters become better tuned to the data it's clear that the LSTM has been able to model the structure of the language & is able to produce completely legible text.

> *Exercise: Try adding another LSTM layer or two to the network. How does this affect performance?*

Sequence Classification

The problem that we will use to demonstrate sequence classification in this tutorial is the IMDB movie review sentiment classification problem. Each movie review is a variable sequence of words and the sentiment of each movie review must be classified.

The Large Movie Review Dataset (often referred to as the IMDB dataset) contains 25,000 highly-polar movie reviews (good or bad) for training and the same amount again for testing. The problem is to determine whether a given movie review has a positive or negative sentiment. The data was collected by Stanford researchers and was used in a 2011 paper where a split of 50-50 of the data was used for training and test. An accuracy of 88.89% was achieved.

Keras provides access to the IMDB dataset built-in. The imdb.load_data() function allows you to load the dataset in a format that is ready for use in neural network and deep learning models. The words have been replaced by integers that indicate the ordered frequency of each word in the dataset. The sentences in each review are therefore comprised of a sequence of integers.

Word Embedding

We will map each movie review into a real vector domain using a popular technique when working with text called word embedding. Unlike one-hot encoding of words, a word embedding has a much lower dimensionality, and is designed to be able to capture synonomy. Word embedding is a technique where words are encoded as real-valued vectors in a high dimensional space, where the similarity between words in terms of meaning translates to closeness in the vector space.

Keras provides a convenient way to convert positive integer representations of words into a word embedding by an Embedding layer.

We will map each word onto a 32 length real valued vector. We will also limit the total number of words that we are interested in modeling to the 5000 most frequent words, and zero out the rest. Finally, the sequence length (number of words) in each review varies, so we will constrain each review to be 500 words, truncating long reviews and pad the shorter reviews with zero values.

Now that we have defined our problem and how the data will be prepared and modeled, we are ready to develop an LSTM model to classify the sentiment of movie reviews.

Simple LSTM for Sequence Classification

We can quickly develop a small LSTM for the IMDB problem and achieve good accuracy.

Let's start off by importing the classes and functions required for this model and initializing the random number generator to a constant value to ensure we can easily reproduce the results:

```
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
# fix random seed for reproducibility
numpy.random.seed(7)
```

We need to load the IMDB dataset. We are constraining the dataset to the top 5,000 words. We also split the dataset into train (50%) and test (50%) sets.

```
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
```

Next, we need to truncate and pad the input sequences so that they are all the same length for modeling. The model will learn the zero values carry no information so indeed the sequences are not the same length in terms of content, but same length vectors is required to perform the computation in Keras.

```
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
```

We can now define, compile and fit our LSTM model.

The first layer is the Embedded layer that uses 32 length vectors to represent each word. The next layer is the LSTM layer with 100 memory units (smart neurons). Finally, because this is a classification problem we use a Dense output layer with a single neuron and a sigmoid activation function to make 0 or 1 predictions for the two classes (good and bad) in the problem.

Because it is a binary classification problem, log loss is used as the loss function (binary_crossentropy in Keras). The efficient ADAM optimization algorithm is used. The model is fit for only 2 epochs because it quickly overfits the problem. A large batch size of 64 reviews is used to space out weight updates.

```
# create the model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vector_length, input_length=max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=3, batch_size=64)
```

Once fit, we estimate the performance of the model on unseen reviews.

```
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Running this example produces the following output:

```
Epoch 1/3
16750/16750 [==============================] - 107s - loss: 0.5570 - acc: 0.7149
Epoch 2/3
16750/16750 [==============================] - 107s - loss: 0.3530 - acc: 0.8577
Epoch 3/3
16750/16750 [==============================] - 107s - loss: 0.2559 - acc: 0.9019
Accuracy: 86.79%
```

You can see that this simple LSTM with little tuning achieves near state-of-the-art results on the IMDB problem. Importantly, this is a template that you can use to apply LSTM networks to your own sequence classification problems.

Exercise: What is the effect of changing the embedding length?