ECS7001 - Neural Networks and NLP
Lab 1: Introduction to NLP with Keras

21/1/2020

The objective of this lab is to get you started with Python and with the Keras library, which will be our main programming platform in this module. The lab is broadly based on the first few chapters of Francois Chollet's introduction to Keras, *Deep Learning with Python*. If you're already familiar with Keras or similar libraries you may skip this. If not, we would encourage you to work your way through the lab making sure you understand the concepts introduced and

1. Starting up with Python and Keras

1.1 Using Python: colab

The easiest way to get started with Python if you've never done it before is via colab, a Jupyter Notebook environment running in the cloud. This requires no setup. If you've never done this, open a tab at

https://colab.research.google.com/notebooks/welcome.ipynb

in your browser. A Jupyter Notebook is an environment in which you can type code and add comments.

https://jupyter.org/

There are many introductions to setting up and using Jupyter Notebook, on jupyter's side and elsewhere. We expect most people to use colab for this lab (and for many of the following ones).

Alternatively, you could run Python locally on your machine, and then either use again a Jupyter Notebook to interact with Python, or using another interactive environment such as IDLE.

Python should already be installed on the lab machines. If you want to use your own laptop instead, and it doesn't have Python installed, look at one of the many explanations on how to do this on the Web - e.g., https://wiki.python.org/moin/BeginnersGuide/Download

1.2 Introduction to Keras

Keras:

https://keras.io/

Is a library for doing deep learning in Python which provides a straightforward environment for creating many types of deep learning models. Keras allows you to

1. Define a model mapping your input to your target, as in the following example code, which defines a two-layer model in which the first layer takes as input the overall input of the network and computes an output using the `relu` function, whereas the second layer takes as input the output of the first layer and produces the overall output of the network using the `softmax` function:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

2. Choose a loss function, an optimizer, and evaluation metrics:

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='mse',
              metrics=['accuracy'])
```

3. Train your model using the `fit()` method

   NB: you CAN'T run the following until you've defined the input and output tensors!! we'll do this in Section 3 after learning a bit more about tensors

```
model.fit(input_tensor, target_tensor, bach_size=128, epochs=10)
```

2. Tensors in Keras

All current machine learning systems use *tensors* as their basic data structure, so it is important you familiarize yourselves with tensors and their implementation in Numpy / Keras.

The concept of tensor is a generalization to an arbitrary number of dimensions of the notion of matrix, which should be familiar to those of you who have learned linear algebra (if you are not already familiar with linear algebra you may want to brush up on its basic concepts as it plays a key role in machine learning - see, e.g.,

https://www.deeplearningbook.org/contents/linear_algebra.html).

(This Section of the lab is based on Chollet's chapter 2.) Keras uses the implementation of tensors in the Numpy library.

2.1 Tensor dimensionality

In Numpy, a number is considered a scalar tensor, or tensor of 0 dimensions. Do the following to create a Numpy scalar:

```
import numpy as np

x = np.array(3)
x
```

The `ndim` attribute specifies the number of dimensions:

```
x.ndim
```

An array of numbers is called a *vector* or *1D tensor*.

```
x = np.array([1,2,3])
x
x.ndim
```

Confusingly, the term dimension is also used to refer to the number of elements in a vector, so to avoid such confusion the term *axis* is also used, and we say that vector x in the example above is a 1 axis tensor with 5 entries; i.e., we say that the number returned by x.ndim is the number of axis, or *rank*.

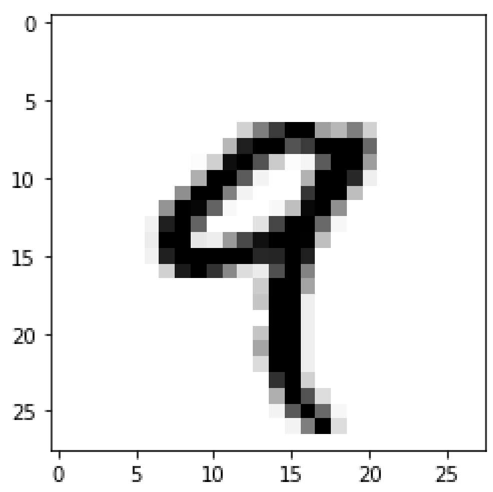
In the example of Keras use for NLP in the next Section we'll use 1D tensors as encodings of movie reviews.

An array of vectors is called a *matrix*, or *2D tensor*, or *tensor of rank 2*. The two axis of a matrix are generally called *rows* and *columns*. Matrices are commonly used to represent images. For example, one of the datasets most commonly used in deep learning, MNIST, is a set of grayscale images of handwritten digits, represented as matrices of 28x28 pixels labelled with the digits they are an image of (0 through 9). The dataset consists of 60,000 training images and 10,000 test images:

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()

digit = train_images[4]

digit.ndim
```

We can visualize the digit using the Matplotlib library:

```
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```



2.2 Tensor attributes

Tensors such as `digit` above have two other important attributes besides `ndim`:

- Shape: a tuple of integers describing how many entries a tensor has along each axis

```
digit.shape
```

- Data type: what type of data is contained in the cells. (In most of our code this will be either integer or real.). In the case of MNIST, the cells contain 8 bit integers, encoding grayscale values between 0 and 255.

```
digit.dtype
```

The variables `train_images` and `test_images` obtained from loading MNIST are examples of 3D tensor: arrays of matrices

```
print(train_images.ndim)
```

Specifically, `train_images` consists of 60,000 matrices of size 28x28:

```
print(train_images.shape)
```

Whereas train_labels (and test_labels) are 1d tensors:

```
print(train_labels.ndim)
print(train_labels.shape)
```

We should point out however that the more general format for image data is 4D, not 3D, as each sample consists of 3 matrices, one per channel.

2.3 Batches and slices

During training a model, the data are processed a *batch* at a time. This can be done by taking a *slice* of the training data. A batch of size 128 of the MNIST training images can be taken as follows:

```
batch = train_images[:128]
batch.shape
```

(Notice that the batch has shape 128x28x28). The next batch can be obtained by taking the next slice:

```
batch = train_images[128:256]
```

And in general, the n-th batch can be taken by

```
batch = train_images[128*n:128*(n+1)]
```

It's possible to take slices of a tensor along the second or third axis instead of the first:

```
batch = train_images[:, 14:, 14:]
```

2.4 Tensor operations

The computation carried out by a neural network is described by the operations carried out but its layers, and these are specified in terms of *tensor operations* over its input. A layer whose activation function is `relu` like the first layer in 1.2 computes its output as follows:

```
output = relu(dow(W,input) + b)
```

And in turn `relu` is defined as an operation over tensors:

```
 np.maximum(x,0)
```

It is important to understand that both `relu` and + are vectorized operations, i.e., operations that take two tensors in input and perform the same operation on all elements. What do you expect the value of z to be after the following operations?

```
x= np.array([1.,-1.,2.])
y = np.array([2.,4.,1.])
z = x+y
print(z)
```

What about the following one?

```
z = np.maximum(x, 0.)
print(z)
```

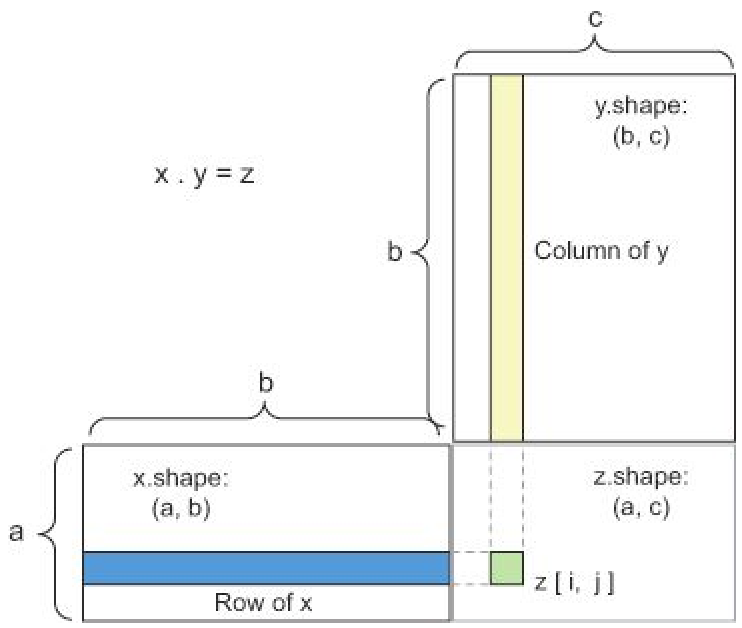
One of the most important tensor operations is tensor product - `dot` in Numpy and Keras - not to be confused with pointwise multiplication:

```
z = x*y
print(z)
```

For vectors, tensor product produces a scalar, the sum of the results of pointwise multiplication:

```
z = np.dot(x,y)
print(z)
```

More importantly, tensor product can be used to multiply two tensors, such as the tensor of inputs to a neural network and the tensor of weights. Crucially, computing the product of two matrices x and y is only possible if the size of the second axis of x (x.shape[1]) is the same as the size of the first axis of y (y.shape[0]).



More in general, the tensor product of two tensors is only possible if the size of the last axis of the first tensor is the same as the size of the first axis of the second.

(a, b, c, d) . (d,) -> (a, b, c)
(a, b, c, d) . (d, e) -> (a, b, c, e)

Finally, there are a number of operations to change the shape of a tensor. You may already be famliar with matrix transposition, which swaps rows with columns. If not, try the following:

```
w = np.array([[1,2,3,4],
              [5,6,7,8]])
print(w)

w.shape


w = np.transpose(w)
w.shape


print(w)
```

More in general, it is possible to reorganize the elements of a tensor using the reshape operation:

```
x = np.array([[0., 1.],
               [2., 3.],
               [4., 5.]])
print(x.shape)

x = x.reshape((6, 1))
x
```

3. An example of binary classification in Keras: classifying movie reviews

For our first example of using (deep) neural networks for NLP applications we will apply a standard Feed-Forward Network (FFN) to the task of classifying movie reviews into positive and negative. This Section of the lab is based on section 4 of Chapter 3 of Chollet's book, where you can find more details.

3.1 The IMDB dataset

The IMDB dataset is a corpus of 50,000 movie reviews from the Internet Movie Database. Half of the reviews are positive, half negative; and the dataset is divided in two parts, a training set of 25,000 reviews and a test set also of 25,000 reviews.

The dataset is included in the Keras library and has already been preprocessed, so it can be downloaded using `load_data()`, just like MNIST:

```
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels)
=imdb.load_data(num_words=10000)
```

(The argument to `load_data` specifies to load only the 10,000 most frequent words.)
`train_data` and `test_data` are lists of reviews, where each review in turn is a list of word indices:

```
train_data[0]
```

Whereas `train_labels` and `test_labels` are lists of 0s and 1s, where 1 means positive, 0 means negative:

```
train_labels

train_labels[0]
```

3.2 Vectorizing the IMDB data

Keras networks take tensors as input and output, not lists, so the first step you need to do in order to use these data to train a neural network is to *vectorize* these lists, i.e., turn them into tensors. There are two ways of doing this:

1. Pad the lists so that all documents are represented by lists of the same length, then turn this list of lists in a 2D tensor (a matrix) of shape (samples, word_indices) - i.e., one row per document, with column j of document i specifying (the index of) the word that occurs at position j in document i, and use as the first layer in the network a layer capable of handling such integer tensors;
2. Encode the lists as one-hot vectors - vectors of 0s and 1s such that a 1 in position j of a vector means that word (with index) j occurs in the document. This one-hot encoding can be fed to a `Dense` layer. This is the approach followed in this Section.

The function `vectorize_sequences` in the following code produces a one-hot encoding of the input data. Try to understand what the code does - in particular, how a sequence is used to set to 1 the cells in a review representation indexed by the word indices.

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  # set specific indices of results[i] to
1s
    return results

# Our vectorized training data
x_train = vectorize_sequences(train_data)
```

```
# Our vectorized test data
x_test = vectorize_sequences(test_data)
```

The labels should be vectorized as well. The following code turns `train_labels` and `test_labels` into vectors of 0s and 1s:

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

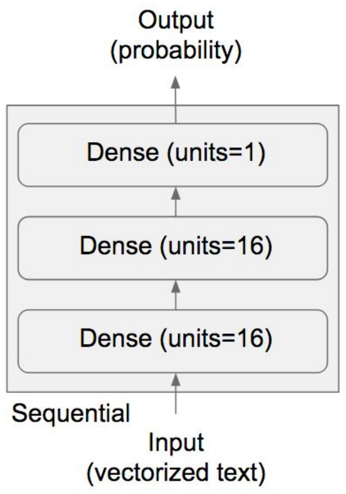
3.3 Defining the model

The neural network for this example takes a one-hot vector encoding of a document as input, and outputs scalar labels (0 or 1). A simple stack of fully connected layers with `relu` activations works well on this problems. The key decisions to make when building such an architecture are

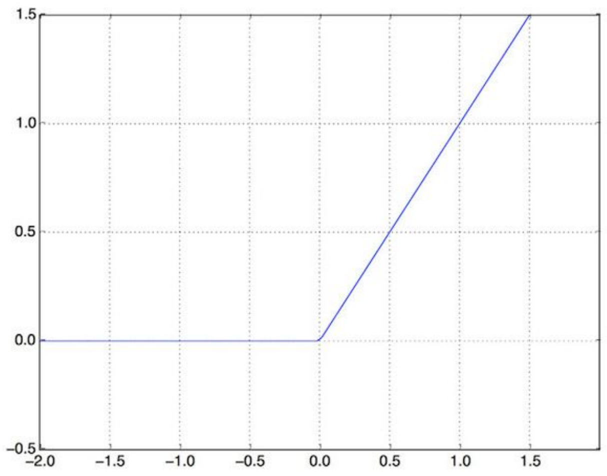
1. How many layers to use
2. How many hidden units to use for each layer.

The architecture used in this example has two intermediate layers with 16 hidden units in each, and a third layer that outputs the scalar encoding the sentiment of a document.

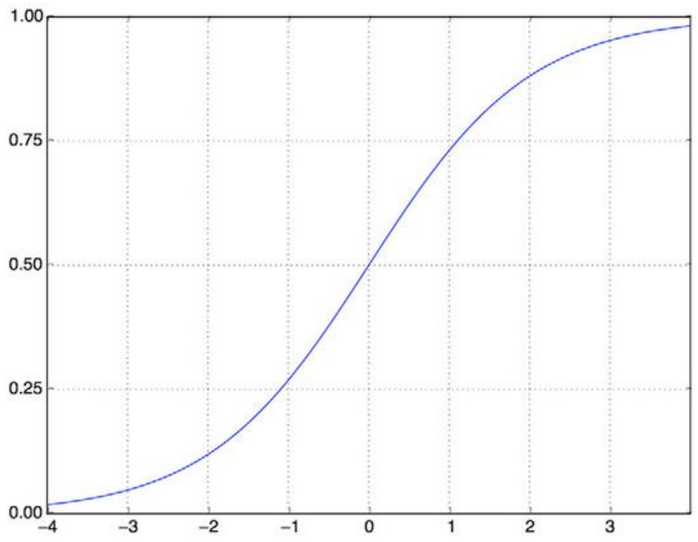
This architecture is illustrated by the following diagram.



Rectified Linear Unit, or `relu`, is used as activation function for the intermediate layers; this is the most commonly used activation function in neural networks. The shape of relu is shown in the following diagram:



A sigmoid activation function is used for the output. The sigmoid outputs a value between 0 and 1 that could be interpreted as a probability for the 1 value:



The following Keras code builds the network:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

The next decisions to make regard the loss function and the optimizer. Crossentropy is a standard choice for models that output probabilities, and rmsprop standard approach to gradient descent in such cases:

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

3.4 Training

The function `fit` is used to train a model. The arguments of this function include the training data (input and output), the number of epochs, and the batch size. The following code trains the model defined above by going for 4 epochs, and using a batch size of 512.

```
model.fit(x_train, y_train, epochs=4, batch_size=512)
```

3.5 Testing

Last, we use the function `evaluate` to run the trained model on test data and get an idea of how it may work on data it has never seen before. The following code evaluates the trained model on the test set:

```
results = model.evaluate(x_test, y_test)
results
```

Note that this model, while very simple, achieves an accuracy of 88%.

The function `predict` can be used to check the confidence of the model in its outputs

```
model.predict(x_test)
```

Note that the model is not equally confident with all reviews.

4. Preprocessing

The movie reviews dataset used in Section 3 had already been cleaned up and put in a format that made it easy for Python to process it, but in real applications, you will have to do this *preprocessing* yourselves. If you didn't take ECS763 NLP and/or have no previous experience with preprocessing you would be advised to get a bit of practice with that, ideally by going through the practice exercises in Bird et al's NLTK book, which is freely available online (in particular, chapters 3 and 5) or, if you have less time, through one of the many available text preprocessing tutorials online, such as this. Key notions you should familiarize yourselves with include:

1. Checking that a document uses a *character encoding* you can deal with (e.g., UTF-8)
2. How to *denoise* text , e.g., by removing non-text (HTML tags, etc), duplicate lines, etc.
3. How to *tokenize* text, i.e., identify tokens (words, punctuation, etc) - you will have noticed that the documents in the IMDB dataset were represented as lists of (encodings of) tokens.
4. How to compute the *frequency* of various *n-grams* (single tokens, sequences of 2 tokens, sequences of 3 tokens, etc) in a collection

Further steps that may be useful for some applications include lemmatization or stemming.