Deep learning for NLP

Multi-layer perceptron with Keras

Benoit Favre

20 Feb 2017

1 Python

The python language is a dynamically typed scripting language with a characteristic indentation style which mimics algorithms. It is widely used in the scientific community and most deep learning toolkits are written in that language. Python is concise and easy to read, however it is relatively slow in itself, so it is necessary to use native libraries for computation-intensive applications. It is advised to read the tutorials over the web[1] to make sure you are sufficiently fluent with the language. There are also tutorials to install it on Linux, Windows or OSX. In the following we will use python version 3.

1.1 Python extensions

There are many ways to install python extensions. The easiest one is `pip`. On Ubuntu it can be installed with:

```
sudo apt-get install python-pip python-dev build-essential
```

2 Numpy

Numpy is a python library with basic math routines, in particular matrix manipulation. It is advised to go through one of the tutorials[2] to refresh your memory. It is also advised to look at the reference documentation[3] for a specific function.

2.1 Numpy matrices

Here is a quick cheat sheet for matrix-specific constructs. Note that numpy arrays are tensors and can have more than two dimensions:

[1] `https://docs.python.org/3/tutorial/`
[2] `https://docs.scipy.org/doc/numpy-dev/user/quickstart.html`
[3] `https://docs.scipy.org/doc/numpy/reference`

```
import numpy as np
m = np.zeros((2, 3), dtype=np.float32) # create a 2x3 matrix
↪ filled with 0.0
m = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int32) # create
↪ an integer matrix from python lists
m = np.random.rand(2, 3) # create a matrix filled with random
↪ numbers in [0, 1]
print(m) # display a matrix (abbreviated for large matrices)
print(m.tolist()) # convert to list to print the whole thing
print(m.shape) # display the number rows and columns of the
↪ matrix
m[1,2] = 9 # access elements
m[0] = np.zeros((4,)) # set row at a time
print(m[:2, 1:3]) # use slices as indices to get a submatrix
m = a + b # matrix addition
m = a * b # elementwise multiplication
m = np.dot(a, b) # matrix multiplication
```

2.2 Acceleration

By default, numpy does not use an optimized matrix multiplication implementation. It can be compiled with BLAS acceleration which makes things much faster on CPU. First, check whether BLAS is compiled-in.

```
import numpy.distutils.system_info as sysinfo
sysinfo.get_info('openblas')
```

If not, make sure you installed numpy from your distribution, not from pip. On Ubuntu, the command is:

```
sudo apt-get install python-numpy
```

There are numerous tutorials on the web to install accelerated libraries for deep learning[4].

3 Keras

Keras is a deep learning framework designed on top of generalist frameworks which significantly reduce the code required to build a system. Theano and Tensorflow are available as backends.

3.1 Installing keras

You can install keras with pip. We will use the Theano backend, so install it as well.

[4] http://www.johnwittenauer.net/configuring-theano-for-high-performance-deep-learning/

```
pip install keras Theano matplotlib scipy pydot-ng --user
```

In the tutorials, we will manipulate small datasets which are processed in a reasonable time without a GPU. However, if you want to address real-life datasets, you will need a machine with a GPU, and you will have to install proprietary drivers, such as Nvidia CuDNN, so that keras backends can make use of GPU acceleration. Note that this part is not required for completing the tutorials.

The backend can be selected by setting the `KERAS_BACKEND` environment variable to either `theano` or `tensorflow`, or modifying the `~/.keras/keras.json` file[5].

To make sure that keras uses the GPU, you need to tell the backend (Theano or Tensorflow) to use it. Assuming cuda is installed in `$HOME/cuda` and cudnn is installed in `$HOME/cudnn-v5`, you can tell Theano to use the GPU by setting the following environment variables:

```
export PATH=$HOME/cuda/bin:$PATH
export CPATH=$HOME/cudnn-v5/include
export LIBRARY_PATH=$HOME/cudnn-v5/lib64
export LD_LIBRARY_PATH=$HOME/cudnn-v5/lib64:$HOME/cuda/lib64
export THEANO_FLAGS=device=gpu,floatX=float32
```

For TensorFlow, install `tensorflow-gpu` with pip, select the backend and set the paths and library paths[6].

3.2 Building a multilayer perceptron

Keras provides utilities and classes for building various kinds of neural networks. The basic workflow consists in loading training data in a numpy array with proper representation, building a model by stacking layers, compiling it, and finally fitting the model to the data. The following tutorial explains briefly those steps. For more information, go to the keras documentation at `http://keras.io`.

Creating data Let's start by building a predictor on a simple classification task: given a set of points in the 2d plane, predict those that belong to the unit circle. Reminding that a point of coordinates (x_0, x_1) belongs to the unit circles iif:

$$x_0^2 + x_1^2 \quad < \quad 1 \tag{1}$$

we can generate random points in $[-2, 2] \times [-2, 2]$ and label them with 1 if they are in the unit circle, 0 otherwise. In python, it would look like this:

[5] https://keras.io/backend/

[6] `https://www.tensorflow.org/get_started/os_setup`

```
import numpy as np
n = 10
X = np.random.rand(n, 2) * 4 - 2
labels = X[:,0] ** 2 + X[:,1] ** 2 < 1
print(X)
print(labels)
```

The X matrix contains on each row a pair of coordinates. The *labels* vector contains `True` or `False` depending on if the corresponding point belongs to the unit circle.

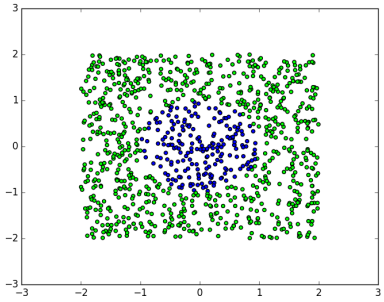
Now, keras expects labels to be specified in one-hot representation, that is for each point a two-dimension vector with 1 at the index of the label (assuming False has index 0 and True has index 1). So we have to convert the truth vector to that particular representation using indexing.

```
Y = np.zeros((n, 2))
Y[np.arange(n), labels.astype(int)] = 1
print(Y)
```

Earlier, we have installed matplotlib, a library which can display points graphically. This is a function which shows the generated data points and colors them according to their label.

```
import matplotlib.pyplot as plt
plt.scatter(X[:,0], X[:,1], c=Y[:,0], cmap='brg')
plt.show()
```

The `brg` cmap corresponds to coloring points in shades from blue to red to green between the lowest to the highest value. The label is specified by the first dimension of Y since both dimensions are complementary. Note that you have to close the window for your program to proceed. With $n = 10$, it's difficult to see the shape of the distribution, so try with a larger sample.



3.3 Building the model

Let's start with a very simple linear model of the form:

$$y = Wx + b \tag{2}$$

where x, y and b are a dimension-2 vectors, and W is a 2×2 matrix. In keras, such model has only one layer (a Dense layer which computes $Wx + b$). Our points have 2 dimensions, and the number of labels is 2.

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(output_dim=2, input_dim=2))
```

We will train this model with the mean squared error loss using the stochastic gradient descent optimizer with default learning rate, and display accuracy during training. The model is compiled this way:

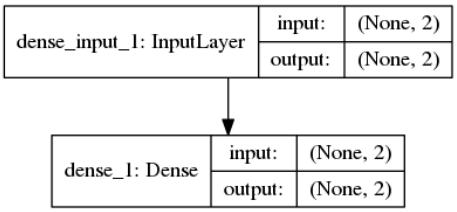
```
model.compile(loss='mse', optimizer='sgd', metrics=['accuracy'])
```

The compiled model is cached, so the next time you execute your code with the same model, it will be much faster.

For debugging purposes, it is often useful to see the content of your model as a picture. You can use the following to see the previously defined model.

```
from keras.utils.visualize_util import plot
plot(model, to_file='model.png', show_shapes=True)
```

The shape of the arrays expected by each layer is very useful to understand how keras wants the data and labels to look like. The first dimensions is None because the model can be fed with various batch sizes, and will adapt its shape depending on the batch size. Note that the first dimension is always the batch size. Keras also likes to separate the input layer to allow creating complex inputs, but here it essentially copies the content of X.



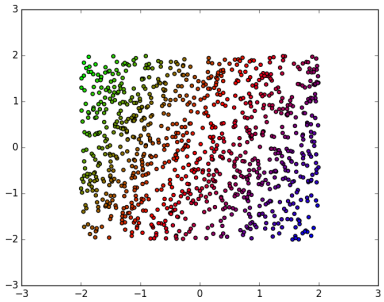
Fitting the model and generating predictions Next, let's fit the model with the training data. It is as simple as calling `model.fit` with the data points X, their labels Y and specifying that we want 50 epochs and a batch size of 5.

```
model.fit(X, Y, nb_epoch=50, batch_size=5)
```

The number of epochs and the batch size can have an impact on the accuracy of the final model. You can play a bit with them but first, let's create a validation set and generate predictions for that set:

```
X_valid = np.random.rand(n, 2) * 4 - 2
Y_pred = model.predict(X_valid)
print(Y_pred)
```

You can display the result on a scale from blue to green, which corresponds to the confidence of the model of a given point to have the 1 label. Can a linear model fit that data?



Now to the deep model So let's try to add more layers and build an actual multilayer perceptron. We can start with 2 hidden layers of size 3 with the *tanh* activation function and a $softmax$ on top of the last layer. To fit the softmax, we will use *binary_crossentropy* loss which is better suited.

The mathematical definition of the model is:

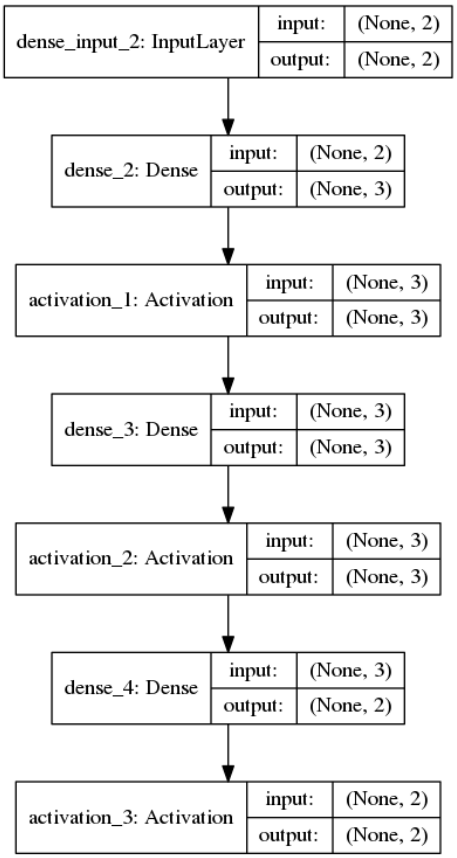
$$\begin{aligned} a_1 &= tanh(W_1 x + b_1) && (3)\\ a_2 &= tanh(W_2 a_1 + b_2) && (4)\\ y &= softmax(W_3 a_2 + b_3) && (5)\\ & && (6) \end{aligned}$$

$W_1, b_1, W_2, b_2, W_3, b_3$ are the parameters of our model. The corresponding keras code is as follows:

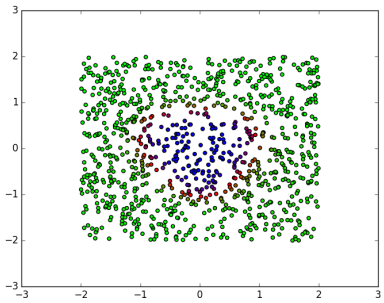
```
from keras.layers import Activation
model = Sequential()
model.add(Dense(output_dim=3, input_dim=2))
```

```
model.add(Activation('tanh'))
model.add(Dense(output_dim=3, input_dim=3))
model.add(Activation('tanh'))
model.add(Dense(output_dim=2, input_dim=3))
model.add(Activation('softmax'))
model.compile(loss='binary_crossentropy', optimizer='sgd',
↪  metrics=['accuracy'])
```

If we display the model, it looks like that:



After 50 epochs, it generally reaches an accuracy of 95% on the training data, which is much better than the linear model. On the validation data, it is much more fit than the linear model. Note that due to the reduced problem size, it is likely that the model does not converge on first run, so try again multiple times.



Extensions You can try to find the best activation function for that dataset, look into how training criteria and optimizers impact the results, etc.

3.4 Other aspects of Keras

Training regimes Training does not need to be performed in one session. You can call `model.fit()` multiple times with one epoch each time. You can also save the model to disk and reload it later and resume training. `model.fit()` also takes a `callback` argument which is a function executed after each epoch of training. It is useful to compute a custom loss on the validation set, save the model, change the learning rate, etc. A few callbacks are already implemented and it is useful to look at the documentation for full details.

Another approach is to directly implement the batch generation loop and call `model.train_on_batch(X_batch, Y_batch)` to train the model on a single batch. This way, you can spare loading the full dataset in memory and load one batch at a time. It is less efficient regarding transfers between CPU and GPU but more flexible. You have to implement some sort of shuffling of the data by yourself.

Saving and loading a model Models are stored in two files: a description of the model structure in json which is equivalent to rebuilding the model with calls to `model.add()`, and the weights of the model stored in HDF5 format.

```
model.save_weights('model.weights')
with open('model.structure', 'w') as fp:
    fp.write(model.to_json())
```

You can also save the model weights to text by getting them from each layers with `layer.get_weights()` as numpy matrices. See the documentation of the layers for the number and shape of these matrices.

To load a model, no need to instantiate it from `Sequential`. Just load it from the json structure, and the weights which have been saved to disk.

```
import json
from keras.models import model_from_json
```

```
with open('model.structure') as fp:
    model = model_from_json(json.loads(fp.read()))
model.load_weights('model.weights')
```

Going further with Keras You are now equipped with the basics of using the keras library for deep learning. To go further, you should read the documentation at `https://keras.io`, look at the examples provided with keras and the numerous implementations available on github and tutorials available on the web.