Deep Learning for Natural Language Processing

Exercise 1 – Tool Kick-off

Niraj Dev Pandey

Natural Language Learning Group
Technische Universität Darmstadt

Docker

TECHNISCHE UNIVERSITÄT DARMSTADT

- „Containerization“ tool which can provide the same execution environment on different operating systems

- Idea:
  1. We provide you with a Docker container on Moodle.
  2. You will write code runnable in that container and submit your python scripts
  3. The tutor, can execute your code without worrying about virtualenvironments, conflicting Keras versions, etc.

- Careful warning: This is our first year of using Docker for DL4NLP.

Docker (cont.)

- Installation: see readme.md in the ZIP archive from moodle
- A hint from 2019 to Windows users:
  - Use Linux in a VM or use docker-toolbox

Python – Programming Language

TECHNISCHE UNIVERSITÄT DARMSTADT

- Home exercises have to be implemented in Python **3**

- Example: read numbers from a file and compute the median value

```python
def median(num):    # function definition
    num.sort()
    length = len(num)
    if length % 2 == 1:
        return num[(length) // 2]
    return 0.5 * (num[length // 2 - 1] + num[(length // 2)])

filename = "num.txt"
numbers = list()
with open(filename) as f:   # open file and close it later on (even after an exception)
    for line in f:
        numbers += [int(x) for x in line.split()]


med = median(numbers)    # call custom function
print(med)               # and print result
```

Python – Programming Language (cont.)

- Good quick reference for newcomers to Python: https://learnxinyminutes.com/docs/python3/

NumPy – Python Library

- Powerful library for scientific computing in Python
  - ndarray: special n-dimensional data type optimized for vector computations
    - **significant performance speedup** possible compared to native Pythonc
  - linear algebra methods
  - **broadcasting** functions

- Installation (if numpy isn't already packaged with your Python distribution)
  - Refer to the installation guide at https://scipy.org/install.html (SciPy contains NumPy)
  - Mind your PATH variables when using different Python installations!

NumPy by Example

- element-wise computations are the default case

```
input_list_a = [ [2,-4,2], [1, 5, 0], [-2,6,8] ]
a_m = np.matrix(input_list_a)                              # numpy matrix
a_arr = np.array([e for lst in input_list_a for e in lst])  # ndarray

input_list_b = [ [1,2,3], [6,2,1], [2,0,1] ]
b_m = np.matrix(input_list_b)                              # numpy matrix
b_arr = np.array([e for lst in input_list_b for e in lst])  # ndarray

c_m = a_m * b_m         # matrix multiplication
c_arr = a_arr * b_arr   # element-wise multiplication

print(c_m)
#[[-18  -4   4]
# [ 31  12   8]
# [ 50   8   8]]


print(c_arr)
# [ 2 -8  6  6 10  0 -4  0  8]  # 1D list
```

$$\begin{pmatrix} 2 & -4 & 2 \\ 1 & 5 & 0 \\ -2 & 6 & 8 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 6 & 2 & 1 \\ 2 & 0 & 1 \end{pmatrix} = ?$$

NumPy: Easy mathematical computations (1)

- Perform mathematical operations easily using NumPy
- *ndarray* != list

```
m1 = np.matrix([ [2,-4,2], [1, 5, 0], [-2,6,8] ])
m2 = np.matrix([ [2,2,3], [6,6,9], [1,4,8] ])

matrices = [m1, m2]
invertible = list()

# compute inverse matrix if possible
for m in matrices:
    print("Current matrix: ", m)
    print("Shape of current matrix: ", m.shape)

    det = np.linalg.det(m)
    print("Determinant of current matrix: ", det)

    if det != 0:
        inv = np.linalg.inv(m)
        invertible.append(inv)
        print("Inverse of current matrix: ", inv)
    else:
        print("Current matrix is not invertible.")
```

use Python lists to define matrices in NumPy

compute determinante to check if matrix is invertible

invert matrix

NumPy: Easy mathematical computations (2)

- Efficient implementations of reoccurring mathematical tasks
- Check documentation to understand behavior

```
import numpy as np

input_1 = [[1,2], [3,4]]
input_2 = [[2,3], [4,5]]

v1 = np.array(input_1).flat[:]
v2 = np.array(input_2).flat[:]
d = np.dot(v1, v2)                # dot product
print(d)
# 40

v1 = np.matrix(input_1)
v2 = np.matrix(input_2)
d = np.dot(v1, v2)                # dot product

print("dot product = ", d)
#  [[10, 13]
#   [22,29]]
```

NumPy – Broadcasting (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

- allows computations on arrays with different but compatible shapes
- can result in a significant performance speedup
- improves code readability and reduces code length

```
import numpy as np

input = [ [2,-4,2], [1, 5, 0], [-2,6,8] ]
a_m = np.matrix(input)

scalar = 3
res = scalar * a_m  # scalar is "broadcast" across the larger
                    # array so that they have compatible shapes

print(res)
# [[  6 -12   6]
#  [  3  15   0]
#  [ -6  18  24]]
```

Multiply a matrix by a scalar

NumPy – Broadcasting (2)

- Broadcasting is not always possible
- *numpy.reshape* allows to change explicitly the dimension

```
x = np.arange(4)
y = x.reshape(1,4)
# error = x.reshape(4,2)
yy = x.reshape(1,4)
yyy = x.reshape(2,2)
z = np.ones(5)
w = np.ones((3,4))
```

array containing the integer values 0,1,2,3

first argument: #rows second argument: #columns

Impossible to fill a 4x2 matrix with four numbers

2x2 matrix

get list with 5 ones

get 3x4 matrix initialized with ones

-
- ValueError: operands could not be broadcast together with shapes (1,4) (5,)

NumPy – Broadcasting (3)

- Possible use case:
  - 256x256x3 array of RGB values
  - Scale each color in the image by a different value

- Use broadcasting mechanism to solve trivial problems

„Find the nearest value to x in the list l“

```
x = 3.14159265359
l = np.random.uniform(3,4,1000000)
pi = l.flat[np.abs(l - x).argmin()]
print(pi)
```

returns a ndarray

Do you see the broadcasting operation?

NumPy – Performance Tips (1)

- It is possible to write highly optimized code with NumPy.
- It is possible to write non-optimized and even slow code with NumPy.

Is data copied or not?	Memory alignment	Broadcasting rules
	Efficient data access	Fancy indexing

- Copy: Original data is copied to a newly allocated memory
- View: Modifications affect the original data
  - Compare with pointers in C

NumPy – Performance Tips (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Copy vs. View:

```
x = np.arange(100)
y = x.view()            # equivalent to y = x[:]
z = x.copy()

print(x)
y[0] = 99
print(x)
print(z)
```

- It is not always obvious if you work on a copy or on a view

```
x = 5
x += 5        # inplace

# vs.

x = 5
x = x + 5    # data copy is made
```

Imagine you work on an array containing some billion values…

NumPy – Performance Tips (3)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Think about indexing!

```
x = np.array([[1,2,3,4],[5,6,7,8]])
y = x[[0], :]    # copy
print(y)
y = x[:1,:]
print(y)         # view
```

- Many NumPy functions have an additional 'out' parameter.
  - avoid allocating new memory data, but reuse memory

```
arr = np.arange(-5,5)
arr_copy = np.abs(arr)
print(arr)                    # [-5 -4 -3 -2 -1  0  1  2  3  4]

print(arr_copy)               # [5 4 3 2 1 0 1 2 3 4]

np.abs(arr, out=arr)
print(arr)                    # [5 4 3 2 1 0 1 2 3 4]
```

NumPy – Performance Tips (4)

Trigonometric functions

sin(x[, out])	Trigonometric sine, element-wise.
cos(x[, out])	Cosine element-wise.
tan(x[, out])	Compute tangent element-wise.
arcsin(x[, out])	Inverse sine, element-wise.
arccos(x[, out])	Trigonometric inverse cosine, element-wise.
arctan(x[, out])	Trigonometric inverse tangent, element-wise.
hypot(x1, x2[, out])	Given the "legs" of a right triangle, return its hypotenuse.
arctan2(x1, x2[, out])	Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.
degrees(x[, out])	Convert angles from radians to degrees.
radians(x[, out])	Convert angles from degrees to radians.
unwrap(p[, discont, axis])	Unwrap by changing deltas between values to 2*pi complement.
deg2rad(x[, out])	Convert angles from degrees to radians.
rad2deg(x[, out])	Convert angles from radians to degrees.

Hyperbolic functions

sinh(x[, out])	Hyperbolic sine, element-wise.
cosh(x[, out])	Hyperbolic cosine, element-wise.

…

http://docs.scipy.org/doc/numpy-1.10.0/reference/routines.math.html

NumPy – Performance Tips (5)

TECHNISCHE UNIVERSITÄT DARMSTADT

Data Alignment

- contiguous array: array is stored in an unbroken block of memory
- rows are stored next to each other → C contiguous (NumPy)
- colums are stored next to each other → Fortran contiguous

NumPy – Performance Tips (7)

Data Alignment

- Broadcasting not possible if data is not contiguously stored in memory

```
a = np.array([[1,2,4,4], [3,9,8,1], [1,1,7,5]])
b = a.copy()
print(a)
# [[1 2 4 4]
#  [3 9 8 1]
#  [1 1 7 5]]

a.shape = (12)
print(a)
# [1 2 4 4 3 9 8 1 1 1 7 5]

b = b.T               # transpose matrix
print(b)
# [[1 3 1]
#  [2 9 1]
#  [4 8 7]
#  [4 1 5]]

c = b.copy()
b.shape = (12)        # AttributeError: incompatible shape for a non-contiguous array
c = c.reshape(12)     # new memory allocated
print(c)
# [1 3 1 2 9 1 4 8 7 4 1 5]
```

Operation would require jumping forwards and backwards in memory ⇒ A transposed 2D matrix cannot be flattened without a copy.

NumPy – Compute the outer product (1)

- short reminder: special case of the *Kronecker product* of matrices
  - operation on two matrices of arbitrary size

- Example:

NumPy – Compute the outer product (2)

Traditional approach

```
x = np.array([1,3,2])
y = np.array([2,1,0,3])

x=x[:, np.newaxis]
x = np.tile(x, (1,4))
y=y[np.newaxis, :]
y=np.tile(y, (3,1))

print(x)
# [[1 1 1 1]
#  [3 3 3 3]
#  [2 2 2 2]]

print(y)
# [[2 1 0 3]
#  [2 1 0 3]
#  [2 1 0 3]]

print(x*y)
# [[2 1 0 3]
#  [6 3 0 9]
#  [4 2 0 6]]
```

Broadcasting approach

```
x = np.array([1,3,2])
x = x.reshape(3,1)
y = np.array([2,1,0,3])

print(x.shape)
# (3,1)

print(y.shape)
# (4,)

print(x*y)
# [[2 1 0 3]
#  [6 3 0 9]
#  [4 2 0 6]]
```

⇒ shorter and *faster*

NumPy – Implementation of evaluation system

- Evaluation of classification problems:
  - **Accuracy**: fraction of correct classifications
  - **Recall**: the fraction of relevant instances that are retrieve
  - **Precision**: fraction of retrieved instances that are relevant
  - **F1**: harmonic mean of precision and recall

- **Contingency matrix:**

gold

predicted

	Label 1	Label 2	Label 3	Label 4	Label 5
Label 1	3	6	1	3	7
Label 2	5	6	7	4	5
Label 3	5	3	4	2	9
Label 4	3	1	0	8	1
Label 5	13	2	1	4	7

Computation of Accuracy

- Accuracy: fraction of correct classifications
  - (sum of diagonal) / (sum of all cells)

gold

predicted

	Label 1	Label 2	Label 3	Label 4	Label 5
Label 1	3	6	1	3	7
Label 2	5	6	7	4	5
Label 3	5	3	4	2	9
Label 4	3	1	0	8	1
Label 5	13	2	1	4	7

- **cm:** contingency matrix
  - Quadratische Matrix

```
denominator = np.sum(cm)
if denominator != 0:
    accuracy = sum(np.diagonal(cm)) / denominator
```

Computation of Precision

- Precision: fraction of retrieved instances that are relevant
  - respectively a specific label x □ computation of a confusion matrix for label
  - Precision = 12/(12+14)

gold

predicted

	12	14
	14	16

fixed row,
all columns except current one

```
denominator = cm[label_id, label_id] + sum(cm[label_id,:][i] for i in range(len(cm[label_id,:])) if i != label_id)
if denominator != 0:
    precision = cm[label_id, label_id] / denominator
```

Final Result

```
def aprf(self, cm, label_id):

    accuracy = 0
    precision = 0
    recall = 0
    f1 = 0

    denominator = np.sum(cm)
    if denominator != 0:
        accuracy = sum(np.diagonal(cm)) / denominator

    denominator = cm[label_id, label_id] + sum(cm[label_id,:][i] for i in range(len(cm[label_id,:])) if i != label_id)
    if denominator != 0:
        precision = cm[label_id, label_id] / denominator

    denominator = cm[label_id, label_id] + sum(cm[:,label_id][i] for i in range(len(cm[:,label_id])) if i != label_id)
    if denominator != 0:
        recall = cm[label_id, label_id] / denominator

    denominator = precision + recall
    if denominator != 0:
        f1 = 2 * precision * recall / denominator

    return (accuracy, precision, recall, f1)
```