Document design and purpose, not mechanics

Fluri Wieland, Pascal Neiger

Document interfaces and reasons, not implementations

Do's

- Document inputs and what the function does:

```
def zStandardization (input, m, std)
#Function standardizes input into z-
#distribution by means of m and #std.
#input can be a vector or matrix of
#doubles
#m is the mean of your distribution. If not
#set, m is calculated depending on your
#data.
#std is the standard deviation of your
#distribution. If not set, std is calculated
#depending on your data
```

Don't's

- Don't document single lines of code like:

```
meanInput = np.mean(input) #calculate
#mean of input
zDiff = input – m #calculates difference for
#each data point and m
zVal = zDiff / std #calculates z-Values
```

Refactor code in preference to explaining how it works

Do's

- If a code is very long, refactor it to multiple pieces and write the explanation at the beginning

0. Cell execution

Press `Ctrl+Enter` or `Shift+Enter` on the next cell to execute the content

In [ ]:
```
print('It works')
```

Navigate between cells with arrows. Press `Enter` to edit cell, `Esc` to exit. Press `a` or `b` too create a new cell above or below.

download libraries

In [ ]:
```
! wget http://scits-training.unibe.ch/data/tut_files/t1.tgz
! tar -xvzf t1.tgz
```

1. Load necessary libraries

In [ ]:
```
import sys

import numpy as np
import matplotlib.pyplot as plt
import IPython.display as ipyd
import tensorflow as tf

# We'll tell matplotlib to inline any drawn figures like so:
%matplotlib inline
plt.style.use('ggplot')
from utils import gr_disp

from IPython.core.display import HTML
HTML("""<style> .rendered_html code {
    padding: 2px 5px;
    color: #0000aa;
    background-color: #cccccc;
} </style>""")
```

Don't's

- Writing big paragraph to explain one big and difficult chunk of code

In [ ]:
```
## 4. Tensor operations
#Sample arrays of different size along first axis.
#They all can be fed into the input_arr placeholder since along first axis size is unconstrained
#For ML tasks we often need to perform operations on high-dimensional data. Theese are represented as tensors in TF. Fo
# sum only along 1st axis
#None stands for unknows length of the array
# squared = (1,4,9,16,25)
# out_sum = 55

tf.reset_default_graph()
input_arr = tf.placeholder(name='input_arr', dtype=tf.float32, shape=(5,))
squared = tf.multiply(input_arr, input_arr)
out_sum = tf.reduce_sum(squared)
np_arr = np.asarray((1,2,3,4,5), dtype=np.float32)
with tf.Session() as sess:
    print(sess.run(out_sum, feed_dict={input_arr: np_arr}))
tf.reset_default_graph()
input_arr = tf.placeholder(name='input_arr', dtype=tf.float32, shape=(None, 5))
squared = tf.multiply(input_arr, input_arr)
out_sum = tf.reduce_sum(squared, axis=1)
np_arr1 = np.asarray([[1,2,3,4,5]], dtype=np.float32)
np_arr2 = np.asarray([[1,2,3,4,5], [2,3,4,5,6]], dtype=np.float32)
np_arr3 = np.asarray([[1,2,3,4,5], [2,3,4,5,6], [25,65,12,12,11], [1,2,3,4,5], [2,3,4,5,6], [25,65,12,12,11]], dtype=np
with tf.Session() as sess:
    print(sess.run(out_sum, feed_dict={input_arr: np_arr1}))
    print(sess.run(out_sum, feed_dict={input_arr: np_arr2}))
    print(sess.run(out_sum, feed_dict={input_arr: np_arr3}))
```

Embed the documentation for a piece of software in that software

Do's

- Integrate documentation directly into code. To extract documentation, you can use knitr and Jupyter

Don't's

- Write a word-document and put it on roughly the same location as the program

Discussion points

- How detailed do you prefer a documentation to be?
- Do you think comments on the implementation are bad?
- Does it require different amounts of documentation depending on your expertise in programming?
- How much of commenting can be averted by proper variable naming / use of functions?