Tips and Tricks for Programming Machine Learning

Søren Hauberg
Section for Cognitive Systems
Technical University of Denmark
sohau@dtu.dk

February 5, 2019

1 Dimensions!

The — by far — most common failure case when implementing machine learning algorithms is to get the dimensions wrong. If you build a working habit of continuously checking dimensions, then you can easily avoid 80% of all bugs.

In the math: My personal approach is to determine all dimensions before I write a single line of code. For example, if I have been tasked with computing the covariance matrix of centered data,

$$\boldsymbol{\Sigma} = \frac{1}{N}\sum_{n=1}^{N} \mathbf{x}_n \mathbf{x}_n^\top, \tag{1.1}$$

then I annotate this equation with dimensions starting with the "inner most" expressions,

$$\boldsymbol{\Sigma} = \frac{1}{N}\sum_{n=1}^{N} \underbrace{\mathbf{x}_n}_{D\times 1}\underbrace{\mathbf{x}_n^\top}_{1\times D}. \tag{1.2}$$

Note that here I do not write $\mathbf{x}_n$ as a D-dimensional vector, but rather a $D \times 1$ dimensional matrix; it's the same thing, but the vector notation easily introduce bugs with missing a transpose.

Next step is to slowly expand the dimensions throughout the equation,

$$\boldsymbol{\Sigma} = \underbrace{\underbrace{\frac{1}{N}}_{1\times 1}\underbrace{\sum_{n=1}^{N}\underbrace{\mathbf{x}_n}_{D\times 1}\underbrace{\mathbf{x}_n^\top}_{1\times D}}_{D\times D}}_{D\times D}. \tag{1.3}$$

The key point is that you can check each of these dimensions as you write the code. If at some point, dimensions in your code do not match the annotation of the equation, then you know that you have either misunderstood the math, or gotten the code wrong. In either case, you have a bug.

In the code: We often end up writing code in fairly high-level programming language such as Python, Matlab, Julia, etc. Here we directly work with vectors, matrices, and general tensors. So, we also need to keep track of dimensions throughout the code. This can be surprisingly challenging.

My personal solution is to place a comment after all lines of code that produce a vector, matrix or tensor. In the beginning this may seem silly, but it pays of in the end. An example in Matlab:

```
X = randn(N, D); % NxD
S = X.' * X;     % DxD
```

or similarly in Python

```
import numpy as np

X = np.random.randn(N, D) # NxD
S = X.T.dot(X)            # DxD
```

In particular it can seem silly to write comments stating dimensions, when the code explicitly state the dimensions — e.g. `X = randn(N, D)` — but it is very useful to be consistent.

2 Implicit sums

Equation 1.1 above explicitly write a sum over all observations. This is readily implemented with a corresponding `for`-loop. However, in high-level programming languages such as Python and Matlab this can be painfully slow. To speed up such computations, it is useful to recall that matrix multiplication is just short-hand notation for many inner or outer products, e.g. if

$$\mathbf{S} = \mathbf{X}^\top \mathbf{Y} \tag{2.1}$$

where

$$\mathbf{X} = \begin{pmatrix} — & \mathbf{x}_1^\top & — \\ & \vdots & \\ — & \mathbf{x}_N^\top & — \end{pmatrix} \in \mathbb{R}^{N \times D} \qquad \text{and} \qquad \mathbf{Y} = \begin{pmatrix} — & \mathbf{y}_1^\top & — \\ & \vdots & \\ — & \mathbf{y}_N^\top & — \end{pmatrix} \in \mathbb{R}^{N \times D} \tag{2.2}$$

then

$$\mathbf{S} = \sum_{n=1}^{N} \mathbf{x}_n \mathbf{y}_n^\top. \tag{2.3}$$

In practice, Eq. 2.1 gives much faster code than Eq. 2.3 (at least in Python and Matlab), and once you get used to writing sums as matrix products, it also becomes easier to read.