Deep learning

1.4. Tensor basics and linear regression

François Fleuret

https://fleuret.org/dlc/

A tensor is a generalized matrix, a finite table of numerical values indexed along several discrete dimensions.

- A 0d tensor is a scalar,
- A 1d tensor is a vector (e.g. a sound sample),
- A 2d tensor is a matrix (e.g. a grayscale image),
- A 3d tensor can be seen as a vector of identically sized matrix (e.g. a multi-channel image),
- A 4d tensor can be seen as a matrix of identically sized matrices, or a sequence of 3d tensors (e.g. a sequence of multi-channel images),
- etc.

Tensors are used to encode the signal to process, but also the internal states and parameters of models. Compounded data structures can represent more diverse data types.

Manipulating data through this constrained structure allows to use CPUs and GPUs at [near] peak performance.

Notes

An RGB image of H rows and W columns of pixels can be encoded as a tensor of size $3 \times H \times W$, or depending on the convention $H \times W \times 3$.
A series of N images can thus be encoded as a single tensor of size $N \times 3 \times H \times W$.
By using tensor operations, all the actors involved in deep learning from , the GPU maker to the driver, library, and application designers can make strong assumptions that allow to utilize the physical computation units optimally. As we will see in particular moving information in memory can be limited, resulting in faster computations.
Standard Python structures (dictionaries, etc.) should be avoided, and when one has many values to store, they should be in a tensor.

The "dimension" of a vector in linear algebra is its number of coefficients, while the "dimension" of a tensor is the number of indices to specify one of its coefficients.

E.g. an element of $\mathbb{R}^3$ is a three-dimension vector, but a one-dimension tensor.

PyTorch's main features are:

- Efficient tensor operations on CPU/GPU,
- automatic on-the-fly differentiation (autograd),
- optimizers,
- data I/O.

"Efficient tensor operations" encompass both standard linear algebra and, as we will see later, deep-learning specific operations (convolution, pooling, etc.)

A key specificity of PyTorch is the central role of autograd to compute derivatives of *anything* ! We will come back to this.

Notes

One of the key component of PyTorch is autograd, which allows to compute the gradient of any quantity with respect to any tensor involved in the computation. This will be presented in lecture 4.2. "Autograd".
Other deep-learning frameworks offer similar functionalities.

```
>>> x = torch.empty(2, 5)
>>> x.size()
torch.Size([2, 5])
>>> x.fill_(1.125)
tensor([[ 1.1250,  1.1250,  1.1250,  1.1250,  1.1250],
        [ 1.1250,  1.1250,  1.1250,  1.1250,  1.1250]])
>>> x.mean()
tensor(1.1250)
>>> x.std()
tensor(0.)
>>> x.sum()
tensor(11.2500)
>>> x.sum().item()
11.25
```

In-place operations are suffixed with an underscore, and a 0d tensor can be converted back to a Python scalar with `item()`.

Reading a coefficient returns a 0d tensor.

```
>>> x = torch.tensor([[11., 12., 13.], [21., 22., 23.]])
>>> x[1, 2]
tensor(23.)
```


Notes

`size()` returns the size / shape of the tensor and has as many components as the number of dimensions of the tensor. E.g. a tensor of size `torch.Size([2, 5])` is a matrix with two rows and five columns.

We should use `item()` when printing a single value (to a text file for instance), otherwise `tensor(...)` is printed.

The default tensor type `torch.Tensor` is an alias for `torch.FloatTensor`, but there are others with greater/lesser precision and on CPU/GPU. It can be set to a different type with `torch.set_default_tensor_type`. We will come back to this.

PyTorch provides operators for component-wise and vector/matrix operations.

```
>>> x = torch.tensor([ 10., 20., 30.])
>>> y = torch.tensor([ 11., 21., 31.])
>>> x + y
tensor([ 21.,  41.,  61.])
>>> x * y
tensor([ 110.,  420.,  930.])
>>> x**2
tensor([ 100.,  400.,  900.])
>>> m = torch.tensor([[ 0., 0., 3. ],
...                   [ 0., 2., 0. ],
...                   [ 1., 0., 0. ]])
>>> m.mv(x)
tensor([ 90.,  40.,  10.])
>>> m @ x
tensor([ 90.,  40.,  10.])
```

Notes

The `@` operator corresponds to matrix/vector or matrix/matrix multiplication, while `*` is component-wise product and can be applied to tensors of arbitrary size, in particular of dimension greater than 2.

And as in NumPy, the `:` symbol defines a range of values for an index and allows to slice tensors.

```
>>> import torch
>>> x = torch.randint(10, (2, 4))
>>> x
tensor([[8, 7, 6, 6],
        [5, 0, 4, 8]])
>>> x[0]
tensor([8, 7, 6, 6])
>>> x[0, :]
tensor([8, 7, 6, 6])
>>> x[:, 0]
tensor([8, 5])
>>> x[:, 1:3] = -1
>>> x
tensor([[ 8, -1, -1,  6],
        [ 5, -1, -1,  8]])
```

Notes

Pytorch tensors can be sliced in the same way as NumPy arrays can.

PyTorch provides interfacing to standard linear operations, such as linear system solving or eigen-decomposition.

```
>>> y = torch.randn(3)
>>> y
tensor([ 1.3663, -0.5444, -1.7488])
>>> m = torch.randn(3, 3)
>>> q = torch.linalg.lstsq(m, y).solution
>>> m@q
tensor([ 1.3663, -0.5444, -1.7488])
```


Notes

This is a simple example showing how to create a tensor, how to fill it with values following a Gaussian distribution, and how to call `lstsq` to solve a linear system, such that given $y \in \mathbb{R}^3$ and $m \in \mathscr{M}(3,3)$, we get $q \in \mathbb{R}^3$ such that $mq = y$.

Example: linear regression

Given a list of points

$$(x_n, y_n) \in \mathbb{R} \times \mathbb{R}, \ \ n = 1, \ldots, N,$$

can we find the affine function

$$f(x; a, b) = ax + b$$

that "goes best through the points", e.g. minimizes the mean square error

$$\operatorname*{argmin}_{a,b} \frac{1}{N} \sum_{n=1}^{N} \big(\underbrace{ax_n + b}_{f(x_n;a,b)} - y_n \big)^2.$$

Such a model would allow to predict the y associated to a new x, simply by calculating $f(x; a, b)$.

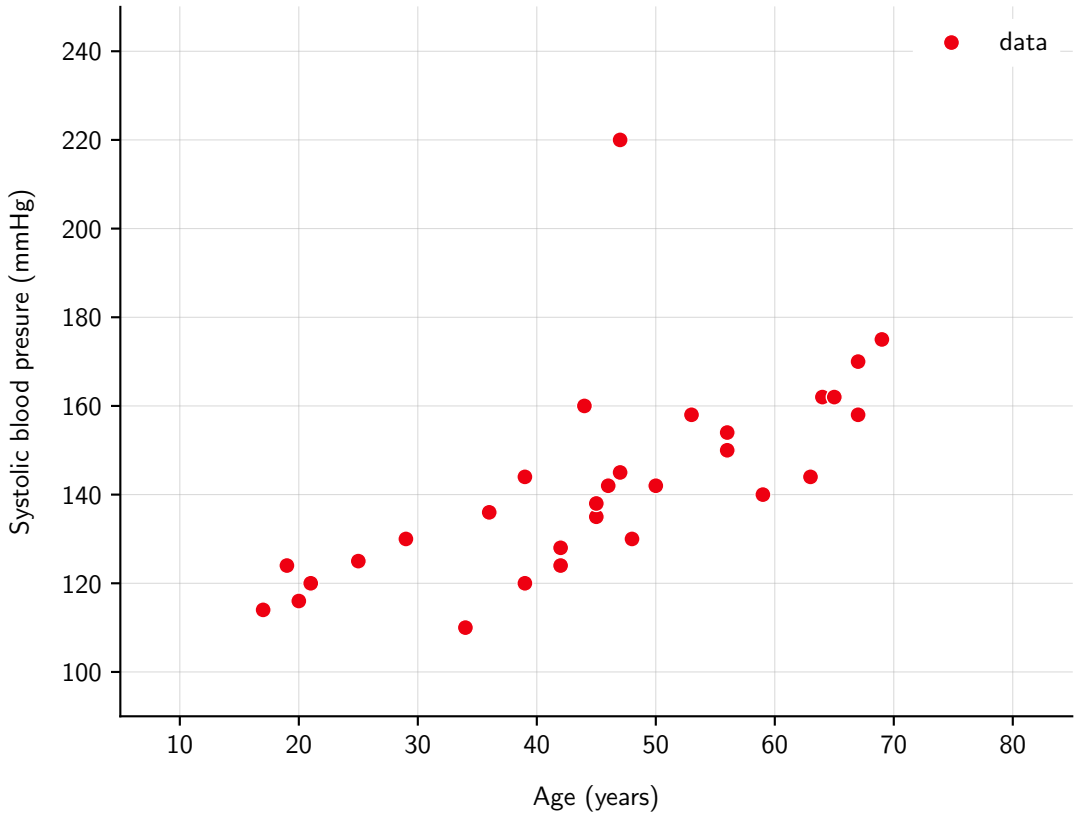
Notes

Linear regression consists in finding an affine function that best fits a list of pairs of points.
A standard way of formalizing the problem is to minimize the mean square error.
Under the affine model f, the output of each x_n is $f(x_n) = ax_n + b$, and we want this value to be as close as possible to y_n. This "closeness" is quantified with the quadratic error $(ax_n + b - y_n)^2$, for all n.
We want to find the best (a, b) that will minimize the mean square error.

```
bash> cat systolic-blood-pressure-vs-age.dat
39  144
47  220
45  138
47  145
65  162
46  142
67  170
42  124
67  158
56  154
64  162
56  150
59  140
34  110
42  128
/.../
```


Notes

The data we use to illustrate linear regression consists of pairs of age and systolic blood pressure. What we want to predict is the blood pressure y_n of a person given his/her age x_n.
Once the model is trained, we should be able to have an estimation of the systolic blood pressure of someone given his/her age.



Notes

This scatter plot depicts the data set. Each pair of the data set is represented by a point, the age being on the x-axis, and the corresponding blood pressure on the y-axis.
The plot shows that the systolic blood pressure increases with the age of a person, roughly in a linear (actually affine) way.

$$\underbrace{\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix}}_{\text{data} \in \mathbb{R}^{N \times 2}}$$

$$\underbrace{\begin{pmatrix} x_1 & 1.0 \\ x_2 & 1.0 \\ \vdots & \vdots \\ x_N & 1.0 \end{pmatrix}}_{x \in \mathbb{R}^{N \times 2}} \underbrace{\begin{pmatrix} a \\ b \end{pmatrix}}_{\alpha \in \mathbb{R}^{2 \times 1}} \simeq \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}}_{y \in \mathbb{R}^{N \times 1}}$$

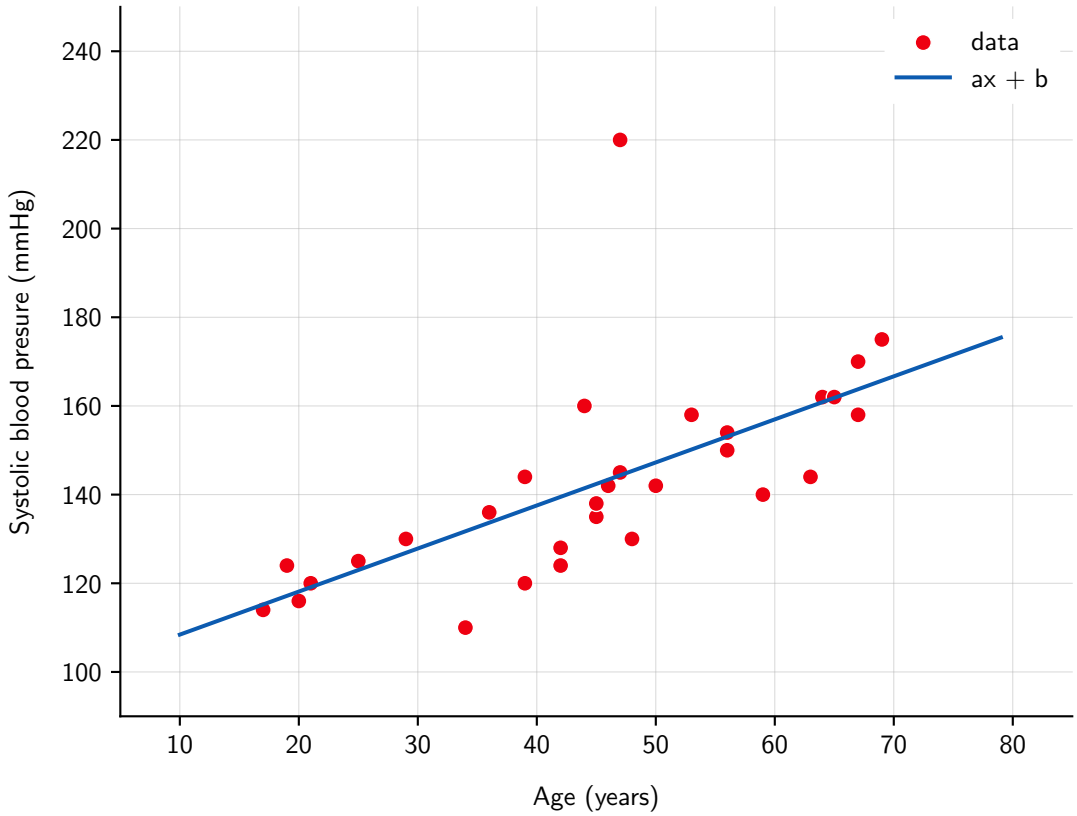
```
import torch, numpy

data = torch.tensor(numpy.loadtxt('systolic-blood-pressure-vs-age.dat'))
nb_samples = data.size(0)

x, y = torch.empty(nb_samples, 2), torch.empty(nb_samples, 1)

x[:, 0] = data[:, 0]
x[:, 1] = 1

y[:, 0] = data[:, 1]

alpha = torch.linalg.lstsq(x, y).solution

a, b = alpha[0, 0].item(), alpha[1, 0].item()
```


Notes

Given each pair (x_n, y_n), we want to compute $ax_n + b$, and make it as close as possible to y_n by finding the best (a, b).
The computation can be written in a matrix form:

$$\begin{pmatrix} x_1 & 1.0 \\ x_2 & 1.0 \\ \vdots & \vdots \\ x_N & 1.0 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} ax_1 + b \\ ax_2 + b \\ \vdots \\ ax_N + b \end{pmatrix}$$

`data.size(0)` returns the size of dimension 0 of `data`, which corresponds to the number of rows of `data`, the number of data points.
Slice `[:, 0]` means the first column of the tensor: dimension 0 on each row.



Notes

The plot shows the model obtained after fitting the data: this is the best linear approximation of the data that minimizes the mean square error. The model can now be used to predict the systolic blood pressure of someone given his/hew age. This example illustrates the limitations of machine learning, and even though the general structure is captured, the model may perform badly on some individuals. Still, this is better than not taking the age in to account and could be useful in practical situations.