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 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.
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.
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.

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

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

Compounded data structures can represent more diverse data types.
PyTorch’s main features are:

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

```python
>>> 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
```
>>> 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.)
There is a confusion between “dimension” for a vector in linear algebra, which is its number of coefficients, and “dimension” for a tensor, which is the number of indices to specify one of its coefficients.

For instance an element of $\mathbb{R}^3$ is a three-dimension vector, but a one-dimension tensor.
PyTorch provides operators for component-wise and vector/matrix operations.

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

```python
>>> import torch
>>> x = torch.empty(2, 4).random_(10)
>>> x
tensor([[8., 1., 1., 3.],
        [7., 0., 7., 5.]])

>>> x[0]
tensor([8., 1., 1., 3.])

>>> x[0, :]
tensor([8., 1., 1., 3.])

>>> x[:, 0]
tensor([8., 7.])

>>> x[:, 1:3] = -1

>>> x
tensor([[ 8., -1., -1., 3.],
        [ 7., -1., -1., 5.]])
```
PyTorch provides interfacing to standard linear operations, such as linear system solving or eigen decomposition.

```python
>>> y = torch.empty(3).normal_()
>>> y
tensor([0.0477, 0.8834, -1.5996])
>>> m = torch.empty(3, 3).normal_()
>>> q, _ = torch.lstsq(y, m)
>>> torch.mm(m, q)
tensor([[0.0477],
        [0.8834],
        [-1.5996]])
```
Example: linear regression
Given a list of points

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

can we find the “best line”

\[f(x; a, b) = ax + b\]

going “through the points”
Given a list of points

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

can we find the “best line”

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

going “through the points”, e.g. minimizing the mean square error

$$\argmin_{a,b} \frac{1}{N} \sum_{n=1}^{N} (\underbrace{ax_n + b - y_n}_{f(x_n; a, b)})^2.$$
Given a list of points

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

can we find the “best line”

\[f(x; a, b) = ax + b\]

going “through the points”, e.g. minimizing the mean square error

\[
\arg\min_{a, b} \frac{1}{N} \sum_{n=1}^{N} \left( ax_n + b - y_n \right)^2.
\]

Such a model would allow to predict the \(y\) associated to a new \(x\), simply by calculating \(f(x; a, b)\).
<table>
<thead>
<tr>
<th>Age</th>
<th>Systolic Blood Pressure</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>144</td>
</tr>
<tr>
<td>47</td>
<td>220</td>
</tr>
<tr>
<td>45</td>
<td>138</td>
</tr>
<tr>
<td>47</td>
<td>145</td>
</tr>
<tr>
<td>65</td>
<td>162</td>
</tr>
<tr>
<td>46</td>
<td>142</td>
</tr>
<tr>
<td>67</td>
<td>170</td>
</tr>
<tr>
<td>42</td>
<td>124</td>
</tr>
<tr>
<td>67</td>
<td>158</td>
</tr>
<tr>
<td>56</td>
<td>154</td>
</tr>
<tr>
<td>64</td>
<td>162</td>
</tr>
<tr>
<td>56</td>
<td>150</td>
</tr>
<tr>
<td>59</td>
<td>140</td>
</tr>
<tr>
<td>34</td>
<td>110</td>
</tr>
<tr>
<td>42</td>
<td>128</td>
</tr>
</tbody>
</table>

/*...*/
\[
\begin{pmatrix}
x_1 & y_1 \\
x_2 & y_2 \\
\vdots & \vdots \\
x_N & y_N \\
\end{pmatrix}
\]

Data \( \in \mathbb{R}^{N \times 2} \)

\[
\begin{pmatrix}
x_1 & 1.0 \\
x_2 & 1.0 \\
\vdots & \vdots \\
x_N & 1.0 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
a \\
b \\
\end{pmatrix}
\]

\( \alpha \in \mathbb{R}^{2 \times 1} \)

\[
\begin{pmatrix}
y_1 \\
y_2 \\
\vdots \\
y_N \\
\end{pmatrix}
\]

\( y \in \mathbb{R}^{N \times 1} \)
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.lstsq(y, x)

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