Deep Learning

A course about theory & practice

UNIVERSITÀ DI PAVIA

(Mini) Batches as Tensors

Marco Piastra

Say it with Tensors

Say it with matrices…

We may want to get rid of the summation when computing the loss function

$$L(D) = \frac{1}{N} \sum_{D} (\boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b}) + b - y^{(i)})^2$$

Let's focus first on $\boldsymbol{W}\boldsymbol{x}$

by defining

$$\boldsymbol{X} := \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \dots & x_d^{(N)} \end{bmatrix}$$

*input data in matrix form (**item index first**)*

Then we can write

$$\boldsymbol{W}\boldsymbol{X}^T = \begin{bmatrix} | & & | \\ \boldsymbol{W}\boldsymbol{x}^{(1)} & \dots & \boldsymbol{W}\boldsymbol{x}^{(N)} \\ | & & | \end{bmatrix}$$

Say it with Tensors

Say it with matrices…

We may want to get rid of the summation when computing the loss function

$$L(D) = \frac{1}{N} \sum_{D} (\boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b}) + b - y^{(i)})^2$$

Consider then $(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$

by defining

$$\hat{\boldsymbol{X}} := \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ x_1^{(N)} & \dots & x_d^{(N)} & 1 \end{bmatrix} \qquad \hat{\boldsymbol{W}} := \begin{bmatrix} & & | \\ \boldsymbol{W} & & \boldsymbol{b} \\ & & | \end{bmatrix}$$

Then we could write

$$\hat{\boldsymbol{W}}\hat{\boldsymbol{X}}^T = \begin{bmatrix} | & & | \\ \boldsymbol{W}\boldsymbol{x}^{(1)} + \boldsymbol{b} & \dots & \boldsymbol{W}\boldsymbol{x}^{(N)} + \boldsymbol{b} \\ | & & | \end{bmatrix}$$

Matrix $\hat{\boldsymbol{W}}$
includes <u>two</u> parameters: $\boldsymbol{W}$ *and* $\boldsymbol{b}$

this may be inconvenient for Autograd,
due to the lack of <u>modularity</u>
(more to follow)

Say it with Tensors

Say it with matrices…

We may want to get rid of the summation when computing the loss function

$$L(D) = \frac{1}{N} \sum_{D} (\boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b}) + b - y^{(i)})^2$$

Consider then $(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$

and let's keep the definition

$$\boldsymbol{X} := \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \dots & x_d^{(N)} \end{bmatrix}$$

It could be convenient to redefine the operator $+$ such that is interpreted as

$$\boldsymbol{W}\boldsymbol{X}^T + \boldsymbol{b} := \begin{bmatrix} | & & | \\ \boldsymbol{W}\boldsymbol{x}^{(1)} & \dots & \boldsymbol{W}\boldsymbol{x}^{(N)} \\ | & & | \end{bmatrix} + \underbrace{\begin{bmatrix} | & & | \\ \boldsymbol{b} & \dots & \boldsymbol{b} \\ | & & | \end{bmatrix}}_{N \text{ times}}$$

Yet, in standard linear algebra this expression is ***wrong***

Say it with Tensors

Say it with matrices…

We may want to get rid of the summation when computing the loss function

$$L(D) = \frac{1}{N} \sum_{D} (\boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b}) + b - y^{(i)})^2$$

Consider then $(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$

and let's keep the definition

$$\boldsymbol{X} := \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \dots & x_d^{(N)} \end{bmatrix}$$

It could be convenient to redefine the operator $+$ such that is interpreted as

$$\boldsymbol{W}\boldsymbol{X}^T + \boldsymbol{b} := \begin{bmatrix} | & & | \\ \boldsymbol{W}\boldsymbol{x}^{(1)} & \dots & \boldsymbol{W}\boldsymbol{x}^{(N)} \\ | & & | \end{bmatrix} + \underbrace{\begin{bmatrix} | & & | \\ \boldsymbol{b} & \dots & \boldsymbol{b} \\ | & & | \end{bmatrix}}_{N \text{ times}}$$

Nonetheless, in actual programming this expression works …

*The fix is called **broadcasting***

Say it with Tensors

Say it with matrices…

We may want to get rid of the summation when computing the loss function

$$L(D) = \frac{1}{N} \sum_{D} (\boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b}) + b - y^{(i)})^2$$

Using broadcasting, we would express the above as

$$L(D) = \frac{1}{N} (\boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{X}^T + \boldsymbol{b}) + b - \boldsymbol{y})^2$$

But it does NOT work

Matrix $\boldsymbol{W}\boldsymbol{X}^T \in \mathbb{R}^{h \times N}$ *and vector* $\boldsymbol{b} \in \mathbb{R}^h$ *are not aligned*
(for ***broadcasting****, the operands'* ***shapes*** *must be* ***right-aligned****)*

Say it with Tensors

Say it with matrices…

We may want to get rid of the summation when computing the loss function

$$L(D) = \frac{1}{N} \sum_{D} (\boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b}) + b - y^{(i)})^2$$

Using broadcasting, we would express the above as

$$L(D) = \frac{1}{N} (\boldsymbol{w} \cdot g(\boldsymbol{X}\boldsymbol{W}^T + \boldsymbol{b}) + b - \boldsymbol{y})^2$$

But it does NOT work yet

Now matrix $\boldsymbol{X}\boldsymbol{W}^T \in \mathbb{R}^{N \times h}$ and vector $\boldsymbol{b} \in \mathbb{R}^h$ are right-aligned in shape
Moreover, the resulting matrix is data item index first

Vector $\boldsymbol{w} \in \mathbb{R}^h$ cannot be left-multiplied with a matrix in $\mathbb{R}^{N \times h}$

Say it with Tensors

Say it with matrices…

We may want to get rid of the summation when computing the loss function

$$L(D) = \frac{1}{N} \sum_{D} (\boldsymbol{w} \cdot g(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b}) + b - y^{(i)})^2$$

Using broadcasting, we can express the above as

$$L(D) = \frac{1}{N}((g(\boldsymbol{X}\boldsymbol{W}^T + \boldsymbol{b})\boldsymbol{w} + b) - \boldsymbol{y})^2$$

The result is a vector in $\mathbb{R}^N$

Broadcasting applies

This is a vector in $\mathbb{R}^N$

A similar behavior of operators is standard in

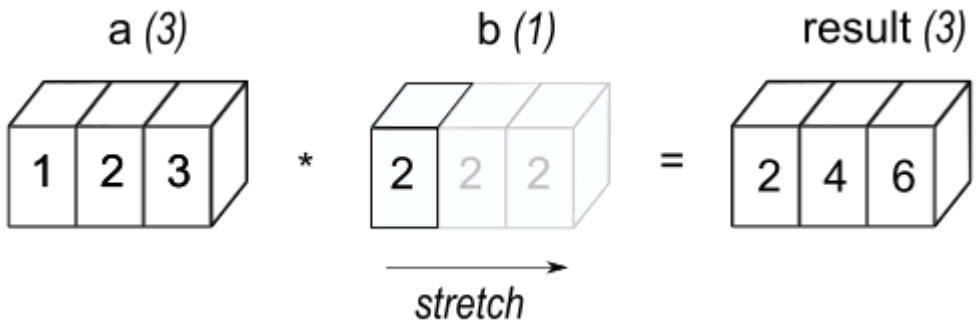
Tensor Broadcasting

Broadcasting in general

PyTorch, TensorFlow, Jax and all adopt the general broadcasting rules of NumPy

- When a tensor is *broadcast*, its entries are **copied only virtually**

  Broadcasting is a performance optimization; ***no actual copying occurs***



[image from https://numpy.org/doc/stable/user/basics.broadcasting.html]

The general broadcasting rule

- When operating on two tensors, NumPy compares their shapes element-wise
  It starts with the **trailing** dimensions, and works its way backward

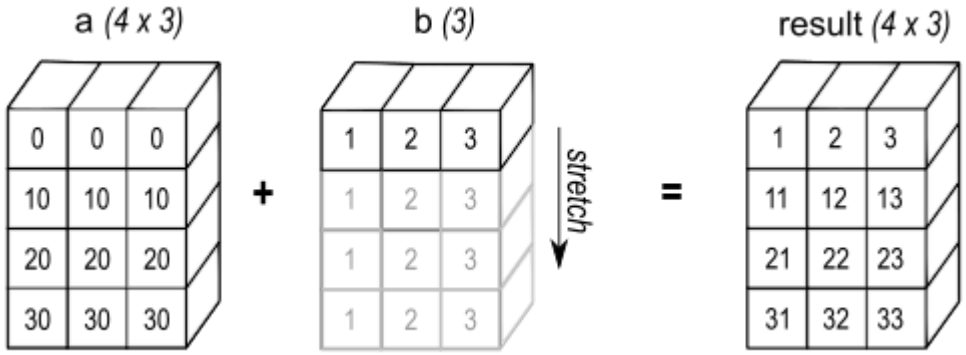
  Two dimensions are *compatible* when

  1. they are *equal*, or
  2. one of them is 1

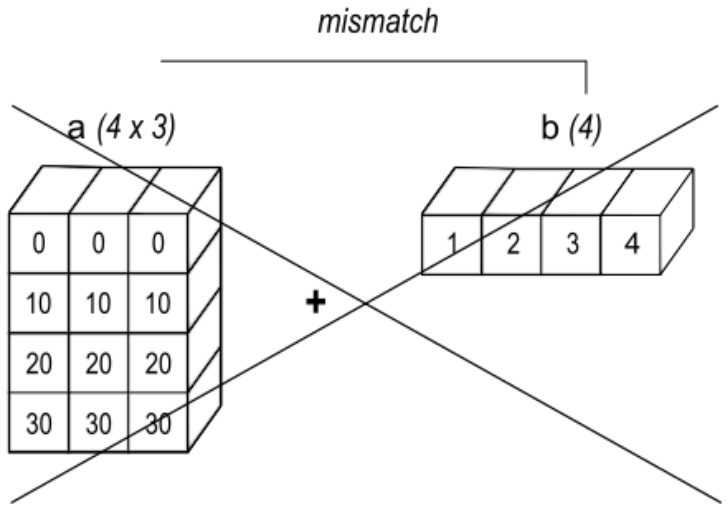
- The size of the resulting tensor is the **maximum size** along each dimension of the input tensors

```
a      (2d tensor):   5 x 4
b      (1d tensor):       1
result (2d tensor):   5 x 4

a      (3d tensor):  15 x 3 x 1
b      (2d tensor):      3 x 5
result (3d tensor):  15 x 3 x 5

a      (4d tensor):   8 x 1 x 6 x 5
b      (3d tensor):       7 x 1 x 5
result (4d tensor):   8 x 7 x 6 x 5
```

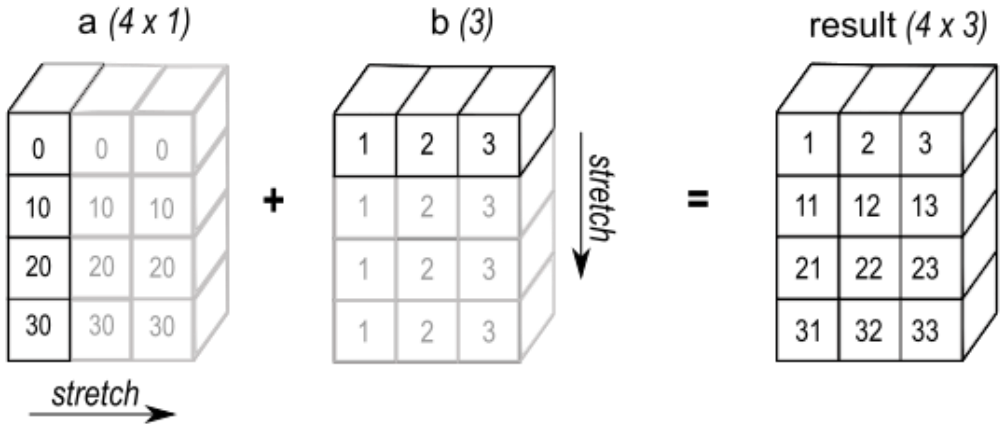
Broadcasting: a few examples



a (2d tensor): 4 x 3
b (1d tensor): 3
result (3d tensor): 4 x 3



a (2d tensor): 4 x 3
b (1d tensor): 4
result: error!



a (1d tensor): 4 x 1
b (1d tensor): 3
result (3d tensor): 4 x 3

[images from https://numpy.org/doc/stable/user/basics.broadcasting.html]