# TUM - I2DL - Matrix derivatives

Dan Halperin - Tutor

January 3, 2023

## 1 Affine layer

$$y = XW + b \tag{1}$$

Where $X_{NxD} \quad W_{DxM} \quad b_{1xM}$.

A known use case is the 1-dim case of a line equation:

$$y = ax + b$$

### 1.1 What is X?

- In the affine layer context, the matrix $X$ is considered to be the input.
- In neural networks, we almost always refer to it as a **batch** of input elements (e.g. images).
- In some deep learning applications (e.g. "style-transfer"), it is also trained by backpropogation.
- Besides being the input of each layer of the network, $X$ is also the **output** of the previous layer.

Let us take for example this one input instance (image) from the **MNIST** handwritten digits' dataset. Each **grayscale** image in this dataset is a $1 \times 8 \times 8$ tensor: 1 for the channels, 8 for the height and 8 for the width.

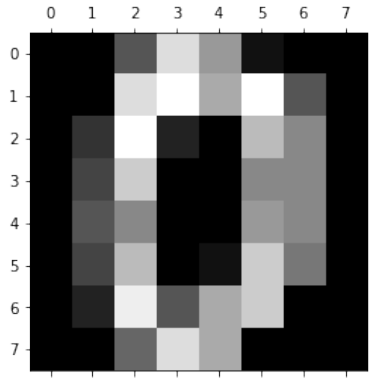


Figure 1: Mnist handwritten $8 \times 8$ image of the digit 0

For the affine layer, as phrased in (1), each input instance is **flattened** to be a row vector inside $X$. Let us take a batch of 2 images from the MNIST dataset.

$$X = \begin{bmatrix} \begin{bmatrix} x_{111} & \dots & x_{118} \\ \vdots & \ddots & \vdots \\ x_{181} & \dots & x_{188} \end{bmatrix} \begin{bmatrix} x_{211} & \dots & x_{218} \\ \vdots & \ddots & \vdots \\ x_{281} & \dots & x_{288} \end{bmatrix} \end{bmatrix} \rightarrow \begin{bmatrix} \begin{bmatrix} x_{111}, & \dots & , x_{118}, & , x_{121}, & \dots & , x_{181}, & \dots & , x_{188} \\ x_{211}, & \dots & , x_{218}, & , x_{221}, & \dots & , x_{281}, & \dots & , x_{288} \end{bmatrix} \end{bmatrix} \tag{2}$$

Here, the batch shape is $2 \times 1 \times 8 \times 8$

**Question**: What if we had a 3-channels RGB images?

**Answer**: The images are flattened the same, row by row and channel by channel. The actual order doesn't matter, but it is important that it will remain consistent among all input instances, so the weights will correspond to the correct entries.

## 1.2 What is W?

- The coefficient matrix.
- In a learning model, they represent the **learnable weights**, and modified during the backpropogation step.
- If in $X$ each row represents one input inside the batch, in $W$ each column represents the weights that are attached from all input neurons (cells in the input vector) to one neuron in the next layer, which is the input to the next layer, as could be seen in Figure 2.

## 1.3 Notes

- **Note**: It is not a linear function, but we treat it as an approximation. Why not? It doesn't follow the rules of linearity, where
$$f(x+y) = f(x) + f(y)$$
or
$$f(ax) = af(x)$$
- Another common notation of an affine layer is
$$y = Wx + b = \left((Wx+b)^T\right)^T = (X^T W^T + b^T)^T \tag{3}$$
Which calculates the exact same thing, but results in a column vector and not a row. $W$ weight vectors are now row vectors and $X$ inputs are now column vectors. It is just a matter of how we construct our inputs and weights.

# 2 Derivatives

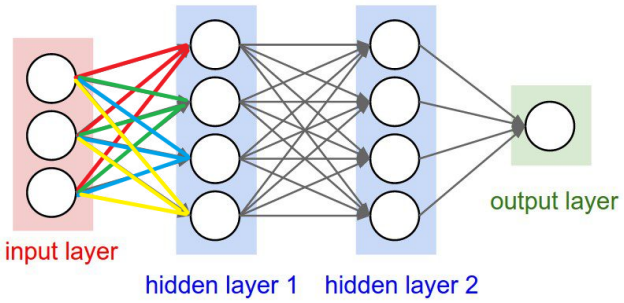


Figure 2: A neural network computational graph. Note: Although we always deal with batches of inputs, in the sketch, the input layer represents only one input instance (e.g one flattened image). Each colour represnts a different weights column vector in $W$. Also, each neuron in the input layer (true to any neuron in the network) will collect the gradients from the flow on the colourful edges that are attached to it.

## 2.1 What is a gradient

It all depends on the function!

- $$f : \mathbb{R} \to \mathbb{R}, \quad x \in \mathbb{R}, \quad \frac{\partial f}{\partial x} = a \in \mathbb{R} \tag{4}$$

- Gradient:
$$f : \mathbb{R}^n \to \mathbb{R}, \qquad x \in \mathbb{R}^n, \quad \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} \in \mathbb{R}^n \tag{5}$$

- Gradient:
$$f : \mathbb{R}^{n \times m} \to \mathbb{R}, \qquad x \in \mathbb{R}^{n \times m}, \quad \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_{1,1}} & \cdots & \frac{\partial f}{\partial x_{1,m}} \\ \vdots & \ddots & \cdots \\ \frac{\partial f}{\partial x_{n,1}} & \cdots & \frac{\partial f}{\partial x_{n,m}} \end{bmatrix} \in \mathbb{R}^{n \times m} \tag{6}$$

- Jacobian:
$$f : \mathbb{R}^n \to \mathbb{R}^m, \quad f(\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}) = \begin{bmatrix} f_1 \\ \vdots \\ f_m \end{bmatrix}, \quad \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{7}$$

- Note that if $x$ was a row vector and so was the function 'image' (result), then this Jacobian matrix would have been transposed.

- An ugly Jacobian (Tensor - A multidimensional matrix):

$$f : \mathbb{R}^{n\times m} \to \mathbb{R}^n, \quad f\left(\begin{bmatrix} w_{11} & \dots & w_{1m} \\ \vdots & \ddots & \vdots \\ w_{n1} & \dots & w_{mn} \end{bmatrix}\right) = \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}, \quad \frac{\partial f}{\partial w} = \begin{bmatrix} \begin{bmatrix} \frac{\partial f_1}{\partial w_{11}} & \dots & \frac{\partial f_1}{\partial w_{1m}} \\ \vdots & \ddots & \dots \\ \frac{\partial f_1}{\partial w_{n1}} & \dots & \frac{\partial f_1}{\partial w_{mn}} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} \frac{\partial f_n}{\partial w_{11}} & \dots & \frac{\partial f_n}{\partial w_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial w_{n1}} & \dots & \frac{\partial f_n}{\partial w_{mn}} \end{bmatrix} \end{bmatrix} \tag{8}$$

- What is a gradient? It is the derivative scalar-valued differentiable function by a vector or a matrix input.
- **Super important**: Neural networks in general could take any shape of input, but they all result in a loss function, that gives a scalar $L \in \mathbb{R}$. That means:
  - In the backpropogation step, the derivative of a learnable weight $w_{ij}$ is to be calculated as a **scalar derivative**:

$$\frac{\partial L}{\partial w_{u,v}} = \sum_i \sum_j \frac{\partial L}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial w_{u,v}} \tag{9}$$

    Where $i, j$ correspond to the rows and columns of $\frac{\partial L}{\partial Y}$ in some function that utilizes $w$, such in $Y = f(X) = XW + b$.
  - In the neural network backpropagation algorithm, we observe only the **current** layer at a time, as an abstraction. We do not try to think of the entire network at once, but step-by-step. Example:

    Toy-Network:

    * Affine()
    * ReLU()
    * **Affine()**
    * Sigmoid()
    * Loss()

    When it comes to think of how to derive the current Affine() layer, we observe it as if it was a function with its own scope.

```
def affine_backward(dout, cache):

    x, w, b = cache
    dx, dw, db = None, None, None

    # some math

    return dx, dw, db
```

Figure 3: Scope of a function

According to the chain rule, the derivative of the loss function value $L$ according to the weight matrix of our current affine layer $W$, would be:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \sigma(Y)} \oplus \frac{\partial \sigma(Y)}{\partial Y} \oplus \frac{\partial Y}{\partial W} \tag{10}$$

In this case, $\frac{\partial L}{\partial \sigma(Y)} \frac{\partial \sigma(Y)}{\partial Y}$ is what we call **dout**, or the **upstream gradient**, and we assume it is already calculated before, as in our current scope (according to the relevant functions, of course). Now it is sent to our current scope, to be calculated as a part of the chain-rule, and sent up the stream to the next layer.

Also, $\oplus$ in scalar derivatives represents a simple multiplication. However, in multidimensional derivatives, $\oplus$ represents some unknown function, which we need to figure out.

# 3 Stanford Article

- Original article by Stanford (cs231n): Link

Let's follow their example:

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix}_{2\times 2} \quad W = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix}_{2\times 3} \tag{11}$$

$$Y = XW = \begin{pmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{pmatrix} \tag{12}$$

Given a loss function $Loss(Y) = L$, we want to calculate $\frac{\partial L}{\partial X}$ or $\frac{\partial L}{\partial W}$.

As seen in (6), the derivative of a scalar by a matrix, is a gradient / Jacobian matrix that has the same shape as the input. Moreover, we saw in (9), that the final derivative of the loss value $L$ by **any** entry of any matrix in the whole neural network is just a **scalar**. For better understanding we could look at the computational graph of the network in, Figure 2, to clearly see that each neuron collects and sums the upstream derivatives (from the loss up to it) - that it took part in calculation of, during the forward pass.

$$\frac{\partial L}{\partial Y} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} \tag{13}$$

So, from (6) we know that the gradient of $Y$ will have the same shape of $Y$, because $L$ is a scalar, and it is calculated as a part of the chain-rule. This is the abstraction notion that is discussed above.

Let's derive $W$. Eventually, after the chain-rule, the derivative of $W$ would have the same shape:

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial w_{1,1}} & \frac{\partial L}{\partial w_{1,2}} & \frac{\partial L}{\partial w_{1,3}} \\ \frac{\partial L}{\partial w_{2,1}} & \frac{\partial L}{\partial w_{2,2}} & \frac{\partial L}{\partial w_{2,3}} \end{pmatrix} \tag{14}$$

Now, this is important. We **do not (!!)** want to calculate the Jacobians. For a better explanation why, refer to the attached article. We have also learned that each entry of $\frac{\partial L}{\partial W}$ is a scalar, that is computed as in (9).

So let's divide and conquer. It is always a better practice, because it's hard to wrap our minds on something bigger than scalars.

$$\frac{\partial L}{\partial w_{11}} = \sum_{i=1}^{2}\sum_{j=1}^{3}\frac{\partial L}{\partial y_{ij}}\frac{\partial y_{ij}}{\partial w_{11}} \tag{15}$$

For better visualization, we could look at it as a **dot product**, which is elementwise multiplication and then summation off all cells (Not what we know as np.dot() - this is confusing). Remember: when deriving a function, it is by the input variable (at least one):

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial Y}\cdot\frac{\partial Y}{\partial w_{11}} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} \begin{pmatrix} \frac{\partial y_{1,1}}{\partial w_{1,1}} & \frac{\partial y_{1,2}}{\partial w_{1,1}} & \frac{\partial y_{1,3}}{\partial w_{1,1}} \\ \frac{\partial y_{2,1}}{\partial w_{1,1}} & \frac{\partial y_{2,2}}{\partial w_{1,1}} & \frac{\partial y_{2,3}}{\partial w_{1,1}} \end{pmatrix} \tag{16}$$

If we go back to (23), we get:

$$\frac{\partial L}{\partial w_{11}} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} \cdot \begin{pmatrix} x_{1,1} & 0 & 0 \\ x_{2,1} & 0 & 0 \end{pmatrix} \tag{17}$$

Now let's perform the dot product, and we get:

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial y_{1,1}}x_{1,1} + \frac{\partial L}{\partial y_{2,1}}x_{2,1} \tag{18}$$

We can do that for every entry $w_{i,j}$ in W, and we get:

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}}x_{1,1} + \frac{\partial L}{\partial y_{2,1}}x_{2,1} & \frac{\partial L}{\partial y_{1,2}}x_{1,1} + \frac{\partial L}{\partial y_{2,2}}x_{2,1} & \frac{\partial L}{\partial y_{1,3}}x_{1,1} + \frac{\partial L}{\partial y_{2,3}}x_{2,1} \\ \frac{\partial L}{\partial y_{1,1}}x_{1,2} + \frac{\partial L}{\partial y_{2,1}}x_{2,2} & \frac{\partial L}{\partial y_{1,2}}x_{1,2} + \frac{\partial L}{\partial y_{2,2}}x_{2,2} & \frac{\partial L}{\partial y_{1,3}}x_{1,2} + \frac{\partial L}{\partial y_{2,3}}x_{2,2} \end{pmatrix} \tag{19}$$

From this matrix, with a little experience, we could derive

$$\frac{\partial L}{\partial W} = \begin{pmatrix} x_{1,1} & x_{2,1} \\ x_{1,2} & x_{2,2} \end{pmatrix} \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} = X^T \cdot \frac{\partial L}{\partial Y} \tag{20}$$

We could, of course, do the exact same thing in order to derive $X$, and we will see that:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\cdot W^T \tag{21}$$

**Note**: This is only true, because $L$ is a scalar. If we just looked at $Y = XW \rightarrow \frac{\partial Y}{\partial W}$ would be a Jacobian.

### 3.1 What about the bias $b$ term in the affine layer?

We could, or course, do the trick of merging it into $X$ and $W$, as we saw in the lecture.

If not:

1. $$Y = XW + b$$

   where $X_{NxD}$, $W_{DxM}$, $b_{1xM}$ $XW_{NxM}$

   That means that each $b_i$ in $b$ corresponds to one feature in a row of $XW$ - but to add them like that, it is quite impossible mathematically, right?

   - NumPy uses **broadcasting** to duplicate the row vector $b$ to be a matrix $B_{NxM}$, by simply copying $b$ $N$ times and stacking them together along the rows, or the 0-axis. But that's just the programming application.
   - Mathematically speaking, it's not $Y = XW + b$, but $Y = XW + 1^N b$, where $1^N$ is a column vector, such that
     $$\begin{bmatrix} 1_1 \\ \vdots \\ 1_N \end{bmatrix} \begin{bmatrix} b_1 & \dots & b_M \end{bmatrix} = \begin{bmatrix} b_1 & \dots & b_M \\ \vdots & \ddots & \vdots \\ b_1 & \dots & b_M \end{bmatrix}_{N \times M}$$
     That gives us the broadcast that python does by itself, and allows us to actually sum those matrices together.

   Now, one can simply follow the exact same paradigm that we've shown above to solve for $b$, or we could just look at $1^N b$ as another $XW$, and do the exact same thing as you did for $XW$, where $1^N$ was $X$ and $b$ was $W$.

2. We see that the derivative of $\frac{\partial L}{\partial b}$ is,
   $$\frac{\partial L}{\partial b} = (1^N)^T \cdot \frac{\partial L}{\partial Y} = \left[ \sum_{i=1}^{N} \frac{\partial L}{\partial y_{i,1}}, \dots, \sum_{i=1}^{N} \frac{\partial L}{\partial y_{i,M}} \right]$$
   Which in NumPy translates into:
   $$np.sum(dout, axis = 0)$$

# 4 Exercise

Given a simple neural network as above:

Toy-Network:

- **Affine()**
- Sigmoid()
- Loss()

Given again that:

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix}_{2 \times 2} \quad W = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix}_{2 \times 3} \tag{22}$$

$$Y = XW = \begin{pmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{pmatrix} \tag{23}$$

And $Y = Affine(X)$

1. Show a solution to compute the graident of the **Sigmoid** layer, w.r.t to the upstream graident.