

Lecture 04

Linear Algebra

for Deep Learning

STAT 479: Deep Learning, Spring 2019

Sebastian Raschka

<http://stat.wisc.edu/~sraschka/teaching/stat479-ss2019/>

Tensors

Vectors, Matrices, and Tensors -- Notational Conventions

Scalar

(rank-0 tensor)

$$x \in \mathbb{R}$$

e.g.,

$$x = 1$$

Vector

(rank-1 tensor)

$$\mathbf{x} \in \mathbb{R}^n$$

but in this lecture,
we will assume

$$\mathbf{x} \in \mathbb{R}^{n \times 1}$$

e.g.,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Matrix

(rank-2 tensor)

$$\mathbf{X} \in \mathbb{R}^{m \times n}$$

e.g.,

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix}$$

$$\mathbf{x}^\top = [x_1 \quad x_2 \quad \dots \quad x_n], \text{ where } \mathbf{x}^\top \in \mathbb{R}^{1 \times n}$$

Vectors, Matrices, and Tensors -- Notational Conventions

We will often use \mathbf{X} as a special convention to refer to the "design matrix." That is, the matrix containing the training examples and features (inputs)

and assume the structure $\mathbf{X} \in \mathbb{R}^{n \times m}$

because n is often used to refer to the number of examples in literature across many disciplines.

E.g.,

$$\mathbf{X} = \begin{bmatrix} x_1^{[1]} & x_2^{[1]} & \dots & x_m^{[1]} \\ x_1^{[2]} & x_2^{[2]} & \dots & x_m^{[2]} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{[n]} & x_2^{[n]} & \dots & x_m^{[n]} \end{bmatrix}$$

$x_2^{[1]}$ = 2nd feature value of the 1st
training example

Why the "ugly" superscript?

Even in context, \mathbf{x}_i may not be always clear:

- does it refer to the feature vector of the i th training example?
- does it refer to i th feature column across training examples?

$$\mathbf{X} = \begin{bmatrix} | & | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \dots & \mathbf{x}_m \\ | & | & | & & | \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} - & \mathbf{x}_1^\top & - \\ - & \mathbf{x}_2^\top & - \\ \vdots & & \\ - & \mathbf{x}_n^\top & - \end{bmatrix}$$

Why the "ugly" superscript?

\mathbf{x}_i and $\mathbf{x}^{[j]}$ are less ambiguous

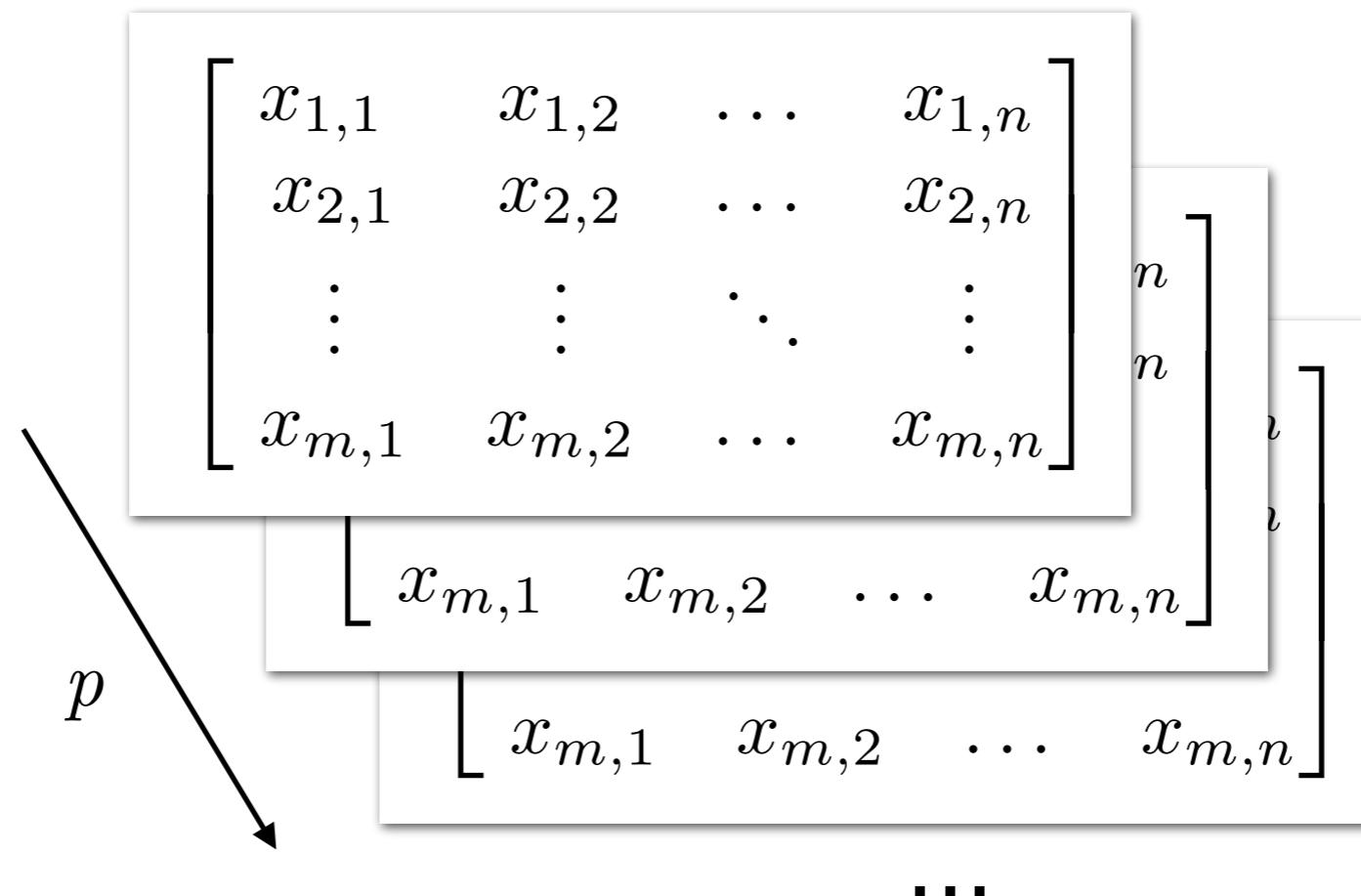
$$\mathbf{X} = \begin{bmatrix} | & & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \dots & \mathbf{x}_m \\ | & & | & & | \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} - & (\mathbf{x}^{[1]})^\top & - \\ - & (\mathbf{x}^{[2]})^\top & - \\ \vdots & & \vdots \\ - & (\mathbf{x}^{[n]})^\top & - \end{bmatrix}$$

Vectors, Matrices, and Tensors -- Notational Conventions

3D Tensor

(rank-3 tensor)

$$\mathbf{X} \in \mathbb{R}^{m \times n \times p}$$



An Example of a 3D Tensor in DL

Single color image

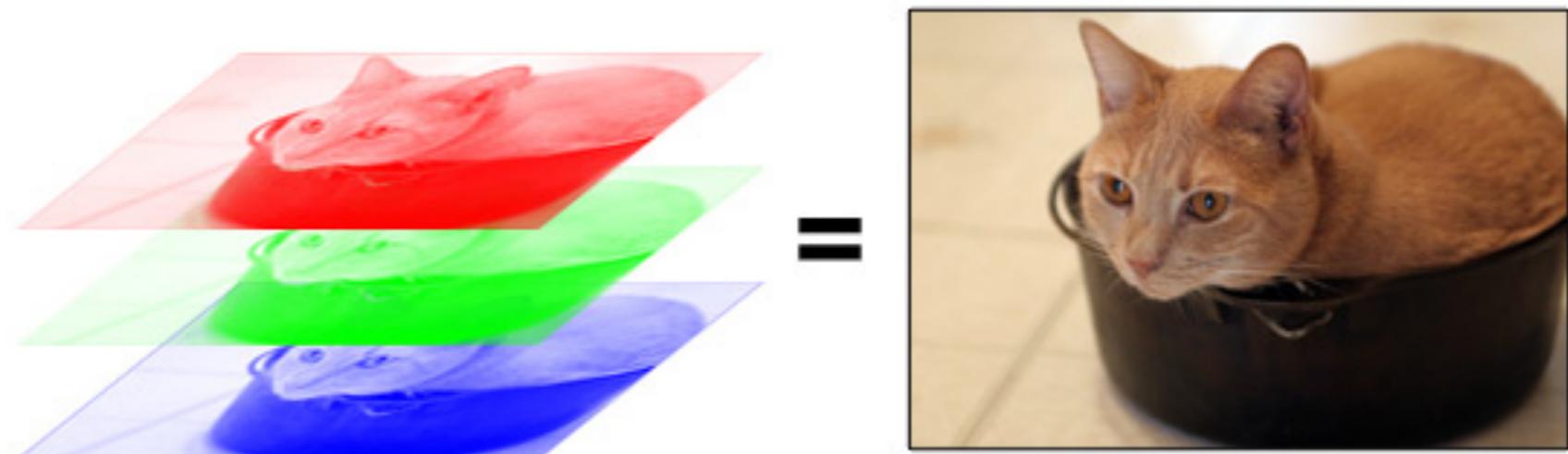
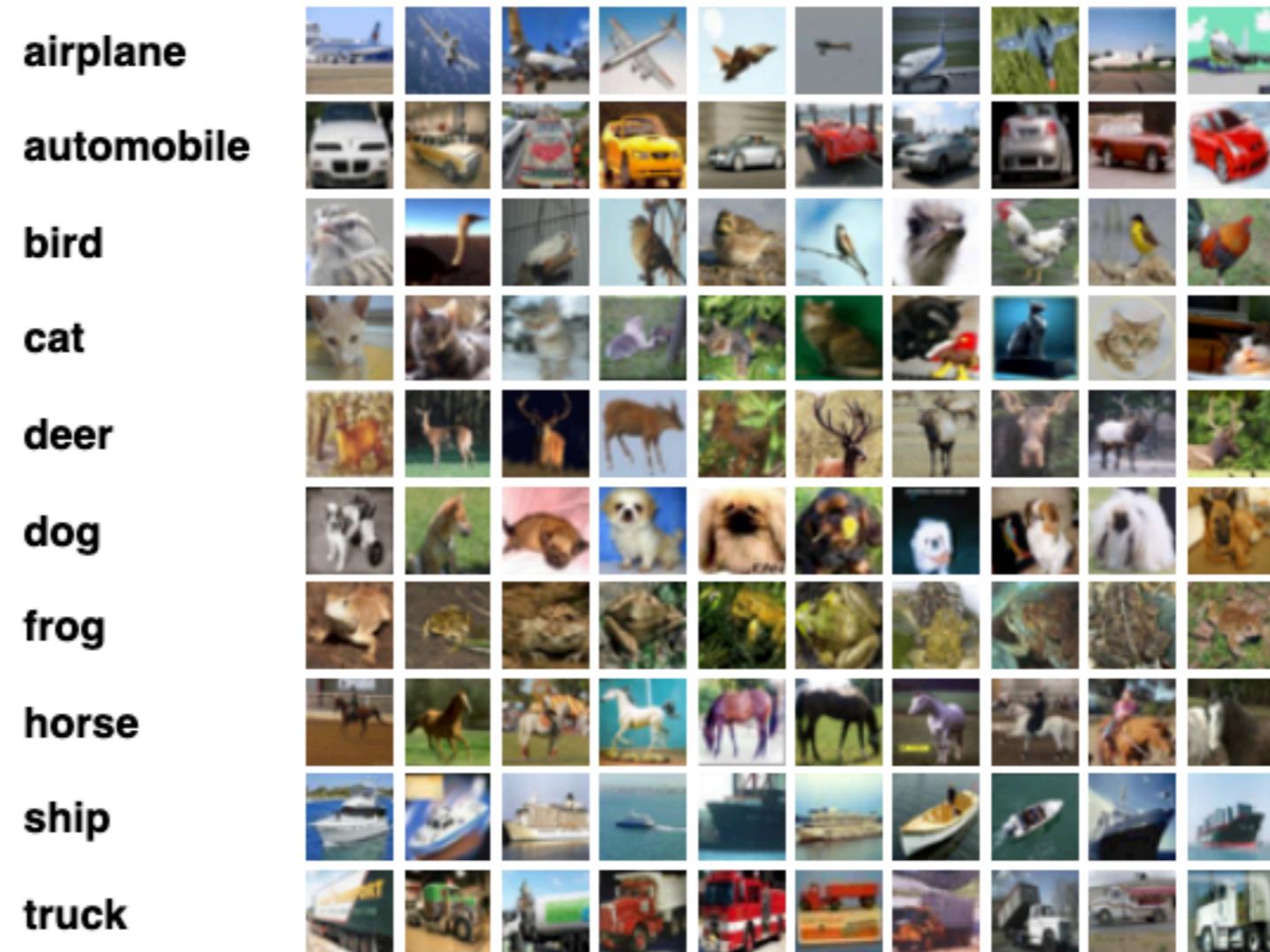


Image Source: <https://code.tutsplus.com/tutorials/create-a-retro-crt-distortion-effect-using-rgb-shifting--active-3359>

(3D tensor for "multidimensional-array" storage and parallel computing purpose,
we still use regular vector and matrix math)

An Example of a 4D Tensor in DL

Batch of images
(as neural network input,
more later)



<https://www.cs.toronto.edu/~kriz/cifar.html>

(4D tensor for "multidimensional-array" storage and parallel computing purpose,
we still use regular vector and matrix math)

Interlude: Multidimensional Arrays as Tensors

`numpy.array / numpy.ndarray =`
`(data structure representation of a tensor)`

`pytorch.tensor / pytorch.Tensor =`
`(data structure representation of a tensor)`

Example:

```
In [1]: import numpy as np
...: import torch
...:
...: a = np.array([1., 2., 3.])
...: b = torch.tensor([1., 2., 3.])

In [2]: print(a.dtype)
...: print(b.dtype)
...:
...: print(a.shape)
...: print(b.shape)
float64
torch.float32
(3,)
torch.Size([3])
```

NumPy and PyTorch Syntax is Very Similar

Example:

```
In [1]: import numpy as np  
...: import torch  
...:  
...: a = np.array([1., 2., 3.])  
...: b = torch.tensor([1., 2., 3.])
```

```
In [2]: print(a.dot(a))  
14.0
```

Note: "dot" vs "matmul"

```
In [3]: print(b.matmul(b))  
tensor(14.)
```

```
In [4]: b.numpy()  
Out[4]: array([1., 2., 3.], dtype=float32)
```

We can convert,
but pay attention to
default types

```
In [5]: torch.tensor(a)  
Out[5]: tensor([1., 2., 3.], dtype=torch.float64)
```

Data Types to Memorize

NumPy data type	Tensor data type	
<code>numpy.uint8</code>	<code>torch.ByteTensor</code>	
<code>numpy.int16</code>	<code>torch.ShortTensor</code>	
<code>numpy.int32</code>	<code>torch.IntTensor</code>	
<code>numpy.int</code>	<code>torch.LongTensor</code>	
<code>numpy.int64</code>	<code>torch.LongTensor</code>	default int in NumPy & PyTorch
<code>numpy.float16</code>	<code>torch.HalfTensor</code>	
<code>numpy.float32</code>	<code>torch.FloatTensor</code>	default float in PyTorch
<code>numpy.float</code>	<code>torch.DoubleTensor</code>	
<code>numpy.float64</code>	<code>torch.DoubleTensor</code>	default float in NumPy

- E.g., `int32` stands for 32 bit integer
- 32 bit floats are less precise than 64 floats, but for neural nets, it doesn't matter much
- For regular GPUs, we usually want 32 bit floats (vs 64 bit floats) for fast performance

PyTorch is Picky about Types

```
In [20]: c = torch.tensor([1., 2., 3.])
....: d = torch.tensor([1, 2, 3])
....:
....: print(c - d)
-----
RuntimeError                                     Traceback (most recent call last)
<ipython-input-20-b04feb3ca8b4> in <module>
      2 d = torch.tensor([1, 2, 3])
      3
----> 4 print(c - d)

RuntimeError: expected type torch.FloatTensor but got torch.LongTensor
```

PyTorch is Picky about Types

Specify the type upon construction based on your main use case:

```
In [21]: c = torch.tensor([1., 2., 3.], dtype=torch.float)
...: d = torch.tensor([1, 2, 3], dtype=torch.float)
...
...: print(c - d)
tensor([0., 0., 0.])
```

You can also change types later/on the fly if you must

```
In [22]: c = torch.tensor([1., 2., 3.])
...: d = torch.tensor([1, 2, 3])
...
...: print(c.float() - d.float())
...: print(c.double() - d.double())
...: print(c.int() - d.int())
...: print(c.long() - d.long())
tensor([0., 0., 0.])
tensor([0., 0., 0.], dtype=torch.float64)
tensor([0, 0, 0], dtype=torch.int32)
tensor([0, 0, 0])
```

So, Why Not Just Using NumPy?

- PyTorch has GPU support:
 - A. we can load the dataset and model parameters into GPU memory
 - B. on the GPU we then have better parallelism for computing (many) matrix multiplications
- Also, PyTorch has automatic differentiation (more later)
- Moreover, PyTorch implements many DL convenience functions (more later)

Loading Data onto the GPU is Easy!

```
In [23]: print(torch.cuda.is_available())
```

```
True
```

```
In [24]: b = b.to(torch.device('cuda:0'))
```

```
...: print(b)
```

```
tensor([1., 2., 3.], device='cuda:0')
```

```
In [25]: b = b.to(torch.device('cpu'))
```

```
...: print(b)
```

```
tensor([1., 2., 3.])
```

How to Check Your CUDA Devices

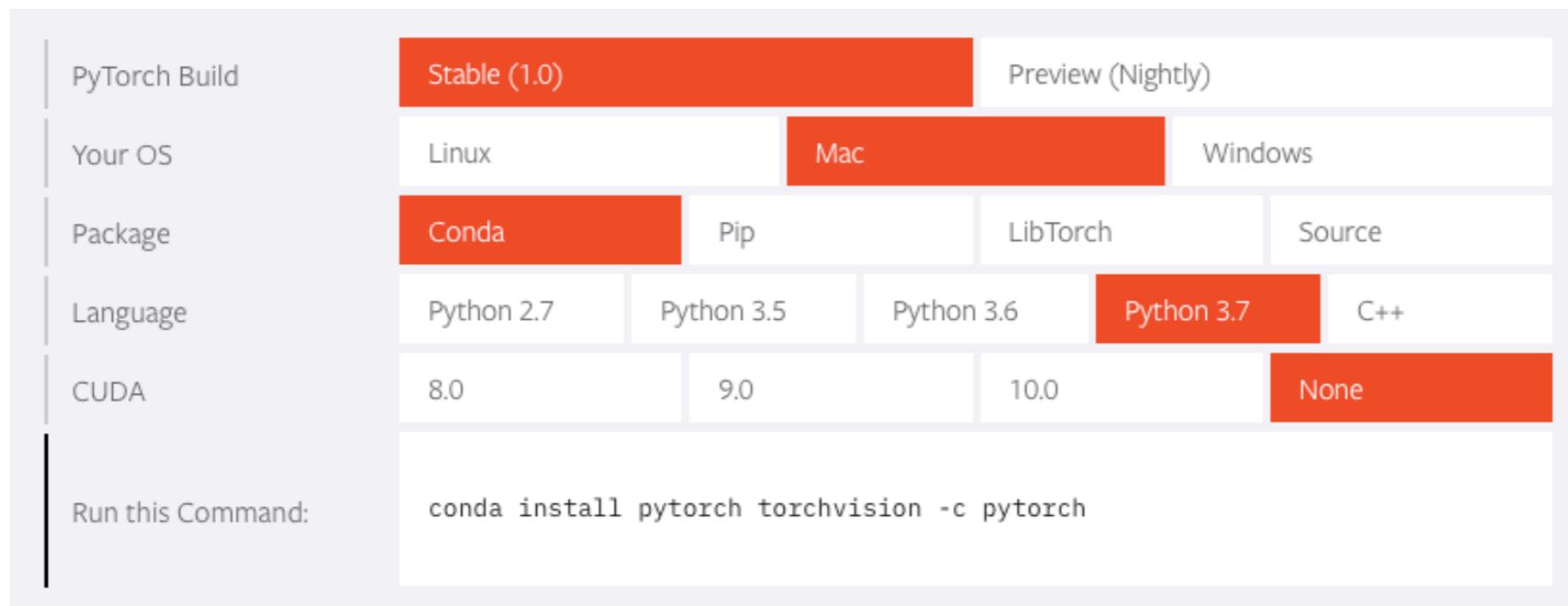
- If you have CUDA installed, you should have access to nvidia-smi
- However, if you are using a laptop, you probably don't have CUDA compatible graphics cards (my laptops don't)
- We will discuss GPU cloud computing later ...

Sun Feb 3 17:57:02 2019									
NVIDIA-SMI 410.78			Driver Version: 410.78			CUDA Version: 10.0			
GPU	Name	Persistence-M	Bus-Id	Disp.A		Volatile	Uncorr.	ECC	
	Fan	Temp		Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.		
0	GeForce GTX 108...	Off	00000000:05:00.0	On				N/A	
32%	55C	P8	19W / 250W		751MiB / 11175MiB	0%	Default		
1	GeForce GTX 108...	Off	00000000:06:00.0	Off				N/A	
29%	51C	P8	17W / 250W		12MiB / 11178MiB	0%	Default		
2	GeForce GTX 108...	Off	00000000:09:00.0	Off				N/A	
28%	50C	P8	17W / 250W		12MiB / 11178MiB	0%	Default		
3	GeForce GTX 108...	Off	00000000:0A:00.0	Off				N/A	
25%	42C	P8	19W / 250W		12MiB / 11178MiB	0%	Default		

About Installing PyTorch

If you want to install PyTorch later (after the lecture) ...

- If you use it on a laptop, you likely don't have a CUDA compatible GPU
- Recommend using CPU version for your laptop (no CUDA)
- Installation on GPU-cloud later ...
- Also, use this selector tool from <https://pytorch.org> (conda is recommended):



Vectors

Vectors

How do we call this again in the context of neural nets?

$$\mathbf{w}^\top \mathbf{x} + b = z$$

where $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$ $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$



Basic vector operations

- Addition (/subtraction)
- Inner products (e.g., dot product)
- Scalar multiplication

TensorFlow and PyTorch Tensors are not Real Tensors

```
In [2]: a = torch.tensor([1, 2, 3])
```

```
In [3]: b = torch.tensor([4, 5, 6])
```

```
In [4]: a * b  
Out[4]: tensor([ 4, 10, 18])
```

```
In [5]: torch.tensor([1, 2, 3]) + 1  
Out[5]: tensor([2, 3, 4])
```

While not equivalent to the mathematical definitions, very useful for computing!

(these "extensions" are now also commonly used in mathematical notation in computer science literature as they are quite convenient)

Matrices

Computing the Output From Multiple Training Examples at Once

- The perceptron algorithm is typically considered an "online" algorithm (i.e., it updates the weights after each training example)
- However, during prediction (e.g., test set evaluation), we could pass all data points at once (so that we can get rid of the "for-loop")

$$\mathbf{X} = \begin{bmatrix} x_1^{[1]} & x_2^{[1]} & \dots & x_m^{[1]} \\ x_1^{[2]} & x_2^{[2]} & \dots & x_m^{[2]} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{[n]} & x_2^{[n]} & \dots & x_m^{[n]} \end{bmatrix}$$

- Two opportunities for parallelism:
multiplying elements to compute the dot product
- computing multiple dot products

Computing the Output From Multiple Training Examples at Once

- Two opportunities for parallelism:
 1. computing the dot product in parallel
 2. computing multiple dot products at once

$$\mathbf{X}\mathbf{w} + b = \mathbf{z}$$


where $\mathbf{X} = \begin{bmatrix} x_1^{[1]} & x_2^{[1]} & \dots & x_m^{[1]} \\ x_1^{[2]} & x_2^{[2]} & \dots & x_m^{[2]} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{[n]} & x_2^{[n]} & \dots & x_m^{[n]} \end{bmatrix}, \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$

(this is why \mathbf{W} is not a "vector" but an $m \times 1$ matrix)

$$\mathbf{z} = \begin{bmatrix} \mathbf{w}^\top \mathbf{x}^{[1]} + b \\ \mathbf{w}^\top \mathbf{x}^{[2]} + b \\ \vdots \\ \mathbf{w}^\top \mathbf{x}^{[n]} + b \end{bmatrix} = \begin{bmatrix} z^{[1]} \\ z^{[2]} \\ \vdots \\ z^{[n]} \end{bmatrix}$$

Computing the Output From Multiple Training Examples at Once

$$\mathbf{X}\mathbf{w} + b = \mathbf{z}$$

(this is why \mathbf{W} is not a "vector"
but an $m \times 1$ matrix)

But NumPy and PyTorch
are not very picky about that:

In [1]: `import torch`

In [2]: `X = torch.arange(6).view(2, 3)`

In [3]: `X`

Out[3]:
`tensor([[0, 1, 2],
 [3, 4, 5]])`

In [4]: `w = torch.tensor([1, 2, 3])`

In [5]: `X.matmul(w)`

Out[5]: `tensor([8, 26])`

In [6]: `w = w.view(-1, 1)`

same as `reshape`
(historic reasons)

In [7]: `X.matmul(w)`

Out[7]:

`tensor([[8],
 [26]])`

Computing the Output From Multiple Training Examples at Once

- Two opportunities for parallelism:
 1. computing the dot product in parallel
 2. computing multiple dot products at once

$$\mathbf{X}\mathbf{w} + b = \mathbf{z}$$

where

$$\mathbf{X} = \begin{bmatrix} x_1^{[1]} & x_2^{[1]} & \dots & x_m^{[1]} \\ x_1^{[2]} & x_2^{[2]} & \dots & x_m^{[2]} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{[n]} & x_2^{[n]} & \dots & x_m^{[n]} \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

(this is why \mathbf{W} is not a "vector" but an $m \times 1$ matrix)

Can you spot the error on this slide?

$$\mathbf{z} = \begin{bmatrix} \mathbf{w}^\top \mathbf{x}^{[1]} + b \\ \mathbf{w}^\top \mathbf{x}^{[2]} + b \\ \vdots \\ \mathbf{w}^\top \mathbf{x}^{[n]} + b \end{bmatrix} = \begin{bmatrix} z^{[1]} \\ z^{[2]} \\ \vdots \\ z^{[n]} \end{bmatrix}$$

Computing the Output From Multiple Training Examples at Once

$$\mathbf{X}\mathbf{w} + b = \mathbf{z}$$

Can you spot the error on this slide?

This should be

$$\mathbf{X}\mathbf{w} + \mathbf{1}_m b = \mathbf{z}$$

but we deep learning researchers are lazy! :)

Broadcasting

- In PyTorch, it works just fine.
- This (general) feature is called "broadcasting"

```
In [4]: torch.tensor([1, 2, 3]) + 1  
Out[4]: tensor([2, 3, 4])
```

```
In [5]: t = torch.tensor([[4, 5, 6], [7, 8, 9]])
```

```
In [6]: t  
Out[6]:  
tensor([[4, 5, 6],  
       [7, 8, 9]])
```

```
In [7]: t + torch.tensor([1, 2, 3])  
Out[7]:  
tensor([[ 5,  7,  9],  
       [ 8, 10, 12]])
```

Broadcasting

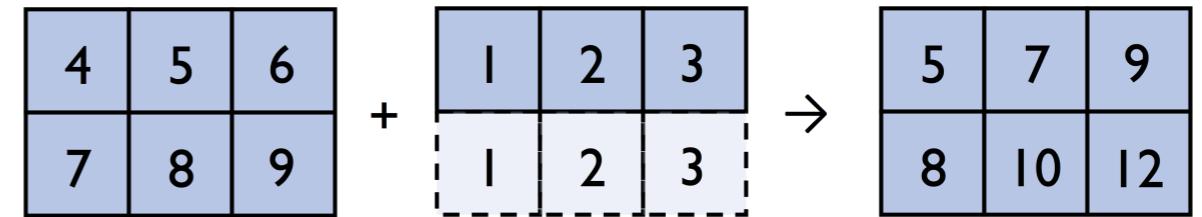
- In PyTorch, it works just fine.
- This (general) feature is called "broadcasting"

```
In [4]: torch.tensor([1, 2, 3]) + 1  
Out[4]: tensor([2, 3, 4])
```



```
In [5]: t = torch.tensor([[4, 5, 6], [7, 8, 9]])
```

```
In [6]: t  
Out[6]:  
tensor([[4, 5, 6],  
       [7, 8, 9]])
```

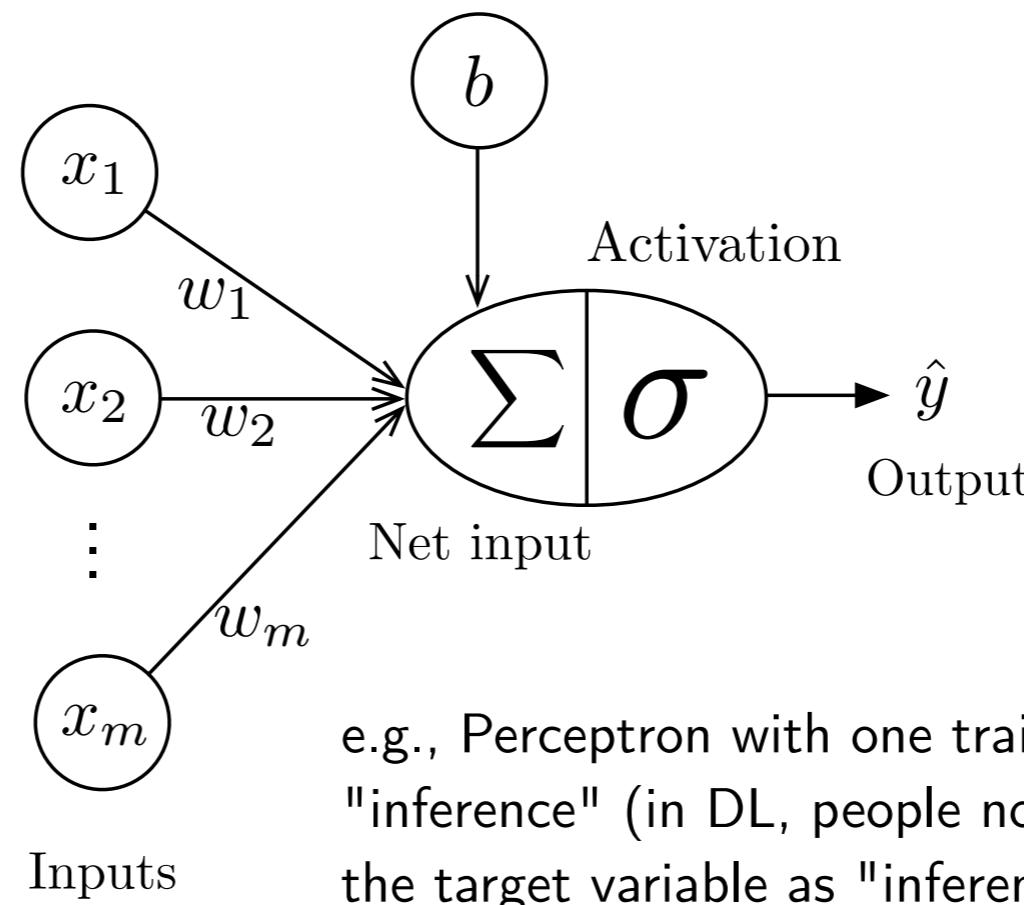


```
In [7]: t + torch.tensor([1, 2, 3])  
Out[7]:  
tensor([[ 5,  7,  9],  
       [ 8, 10, 12]])
```

Implicit dimensions get added,
elements are implicitly duplicated

Connections We Have Seen Before ...

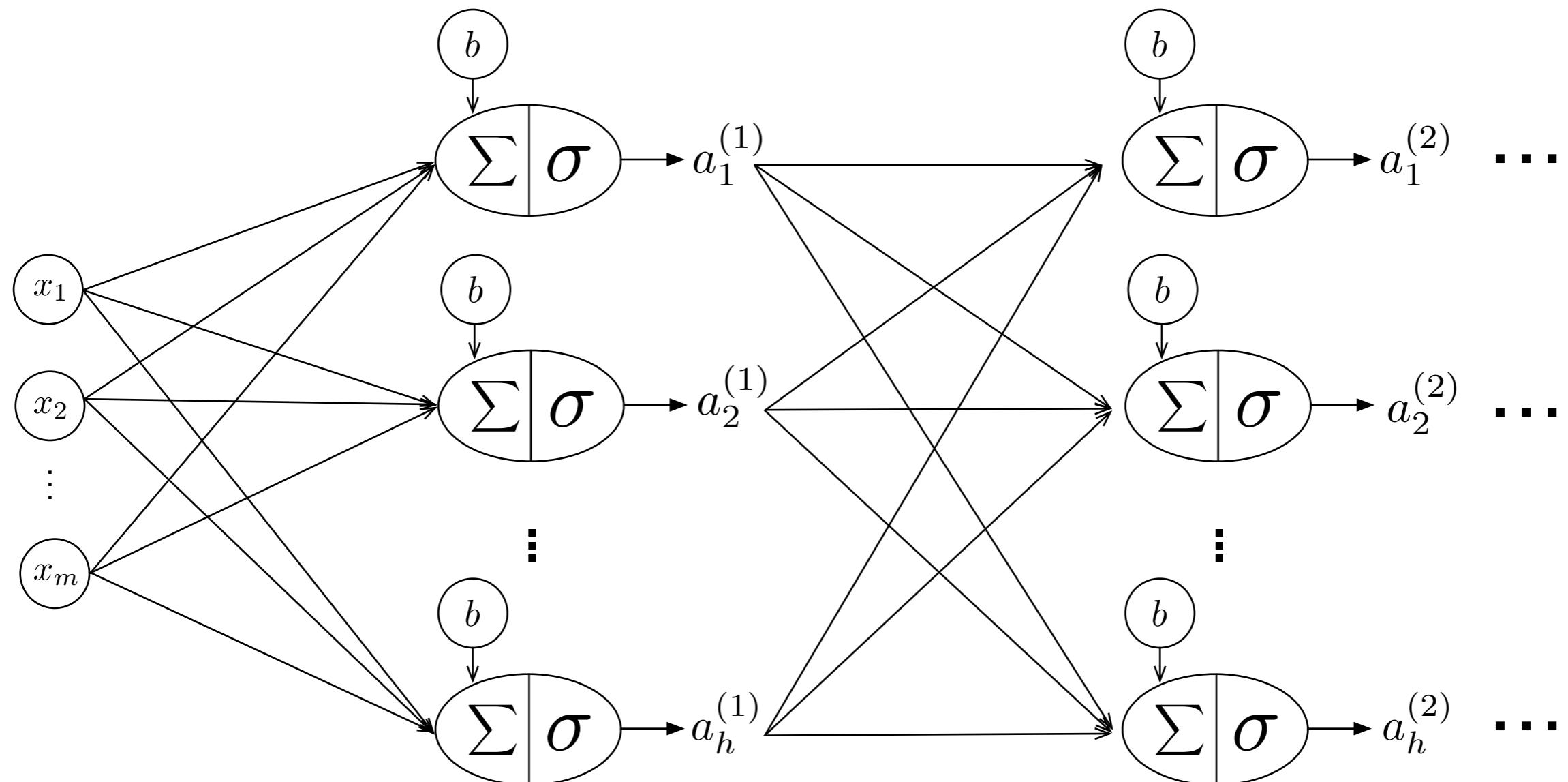
$$\mathbf{x}^\top \mathbf{w} + b = z$$



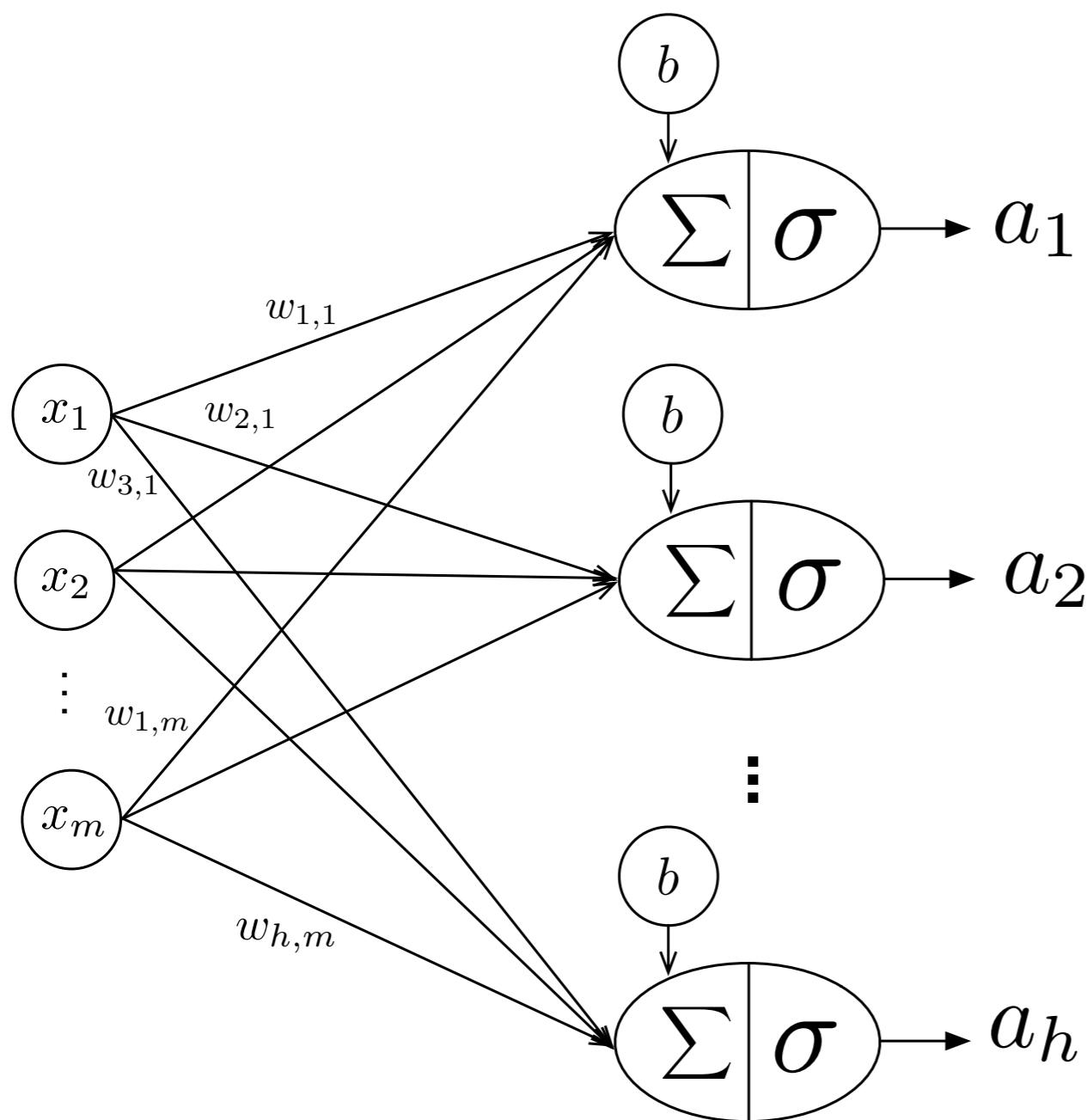
If we have n training examples, $\mathbf{X} \in \mathbb{R}^{n \times m}$, $\mathbf{z} \in \mathbb{R}^{n \times 1}$

$$\mathbf{X}\mathbf{w} + b = \mathbf{z}$$

Connections We Will Encounter Later ...



A Fully Connected Layer



where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

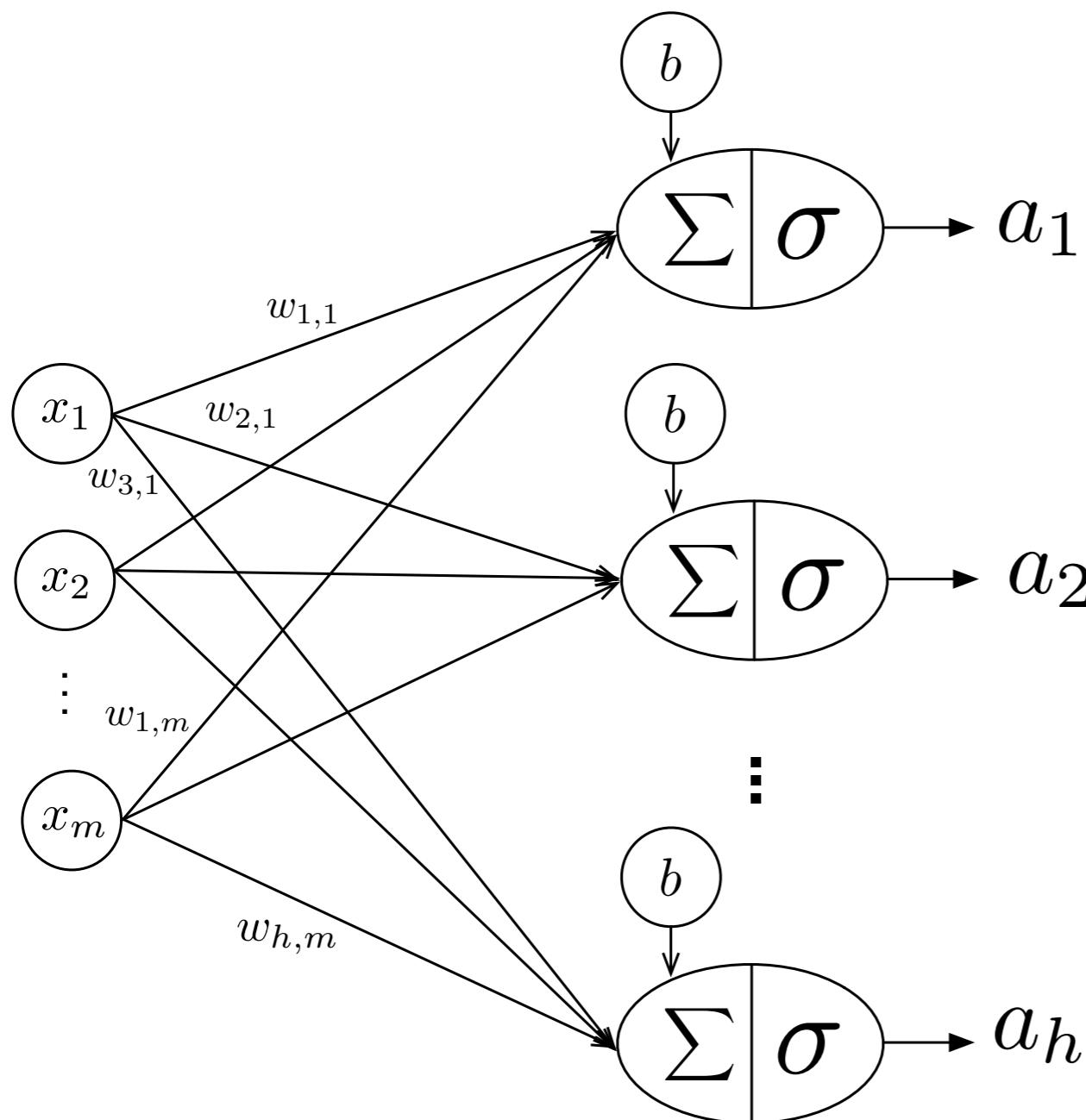
$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,m} \\ w_{2,1} & w_{2,2} & \dots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{h,1} & w_{h,2} & \dots & w_{h,m} \end{bmatrix}$$

Layer activations for 1 training example

$$\sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) = \mathbf{a}$$
$$\mathbf{a} \in \mathbb{R}^{h \times 1}$$

note that $w_{i,j}$ refers to the weight connecting the j -th input to the i -th output.

A Fully Connected Layer



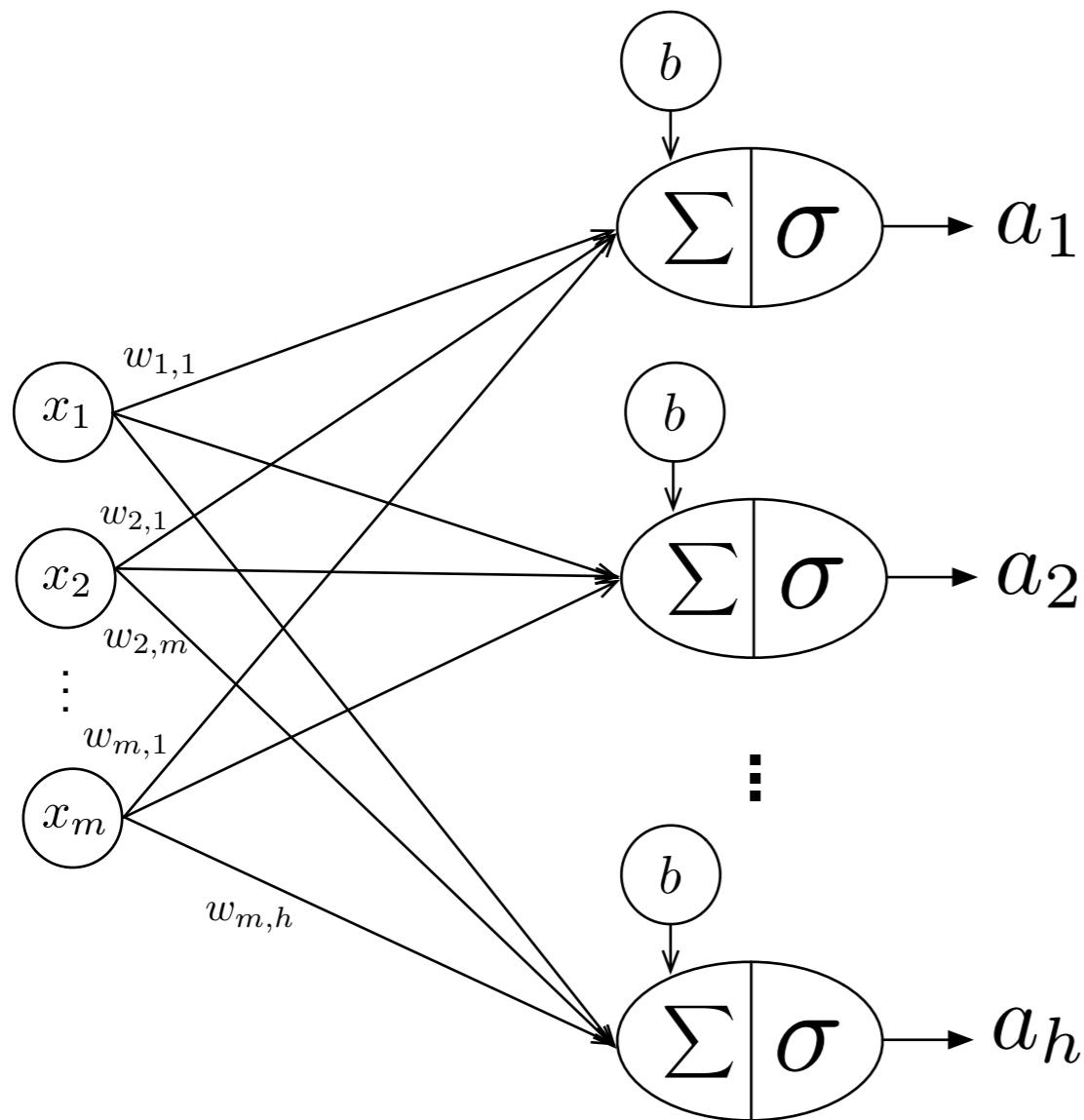
Layer activations for n training examples

$$\sigma([\mathbf{W}\mathbf{X}^\top + \mathbf{b}]^\top) = \mathbf{A}$$

$$\mathbf{A} \in \mathbb{R}^{n \times h}$$

Machine learning textbooks usually represent training examples over columns, and features over rows (instead of using the "design matrix") -- in that case, we could drop the transpose.

Another Common Convention



note that $w_{i,j}$ refers to the weight connecting the i -th input to the j -th output.

where $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,h} \\ w_{2,1} & w_{2,2} & \dots & w_{2,h} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \dots & w_{m,h} \end{bmatrix}$$

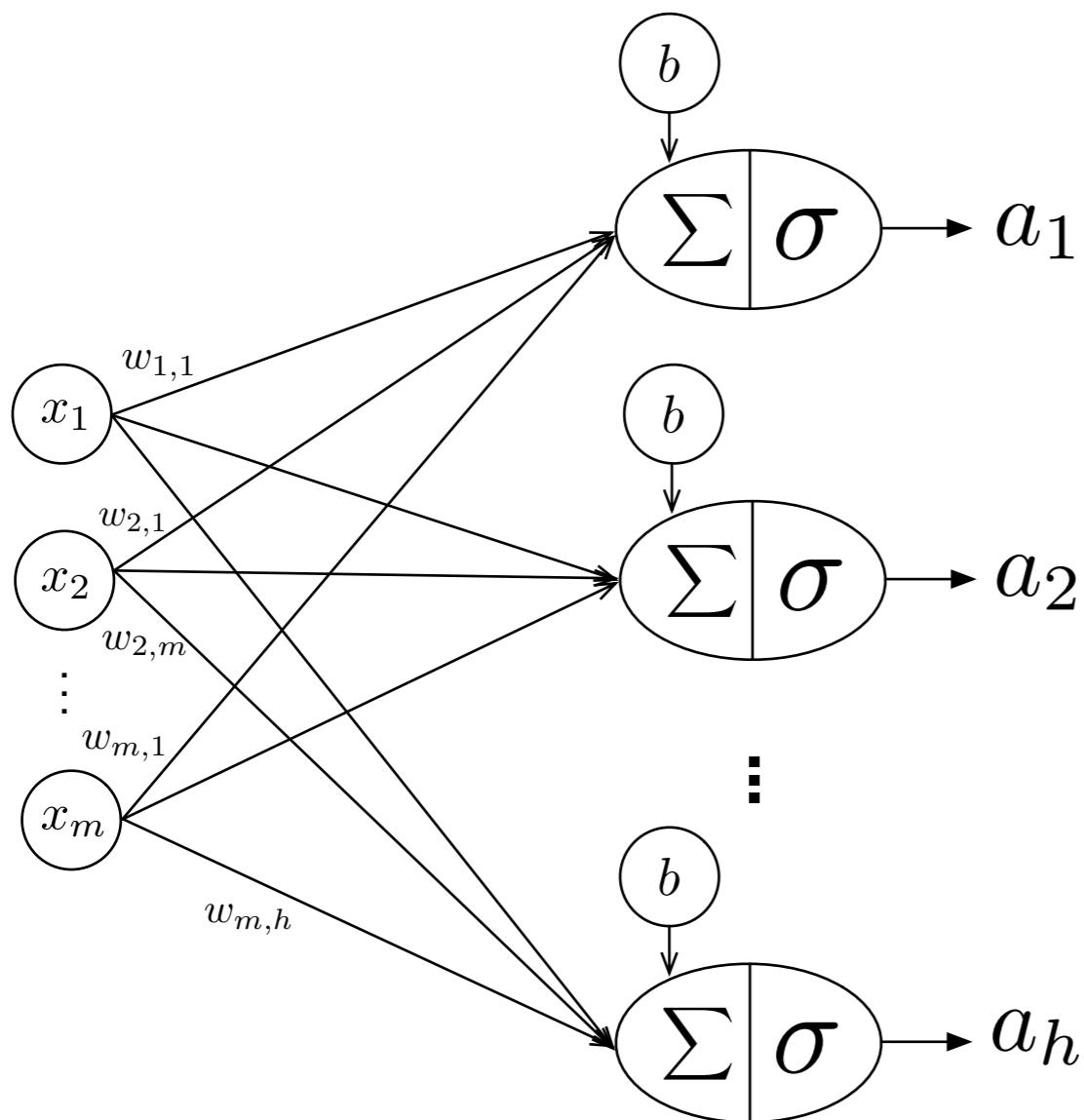
Layer activations for 1 training example

$$\sigma([\mathbf{x}^\top \mathbf{W}]^\top + \mathbf{b}) = \mathbf{a}$$
$$\mathbf{a} \in \mathbb{R}^{h \times 1}$$

In code, we don't need the transpose, since with NumPy and PyTorch, we can multiply the following with matrices:

$$\mathbf{x} \in \mathbb{R}^h \quad \mathbf{a} \in \mathbb{R}^h$$

Another Common Convention



Layer activations for n training example

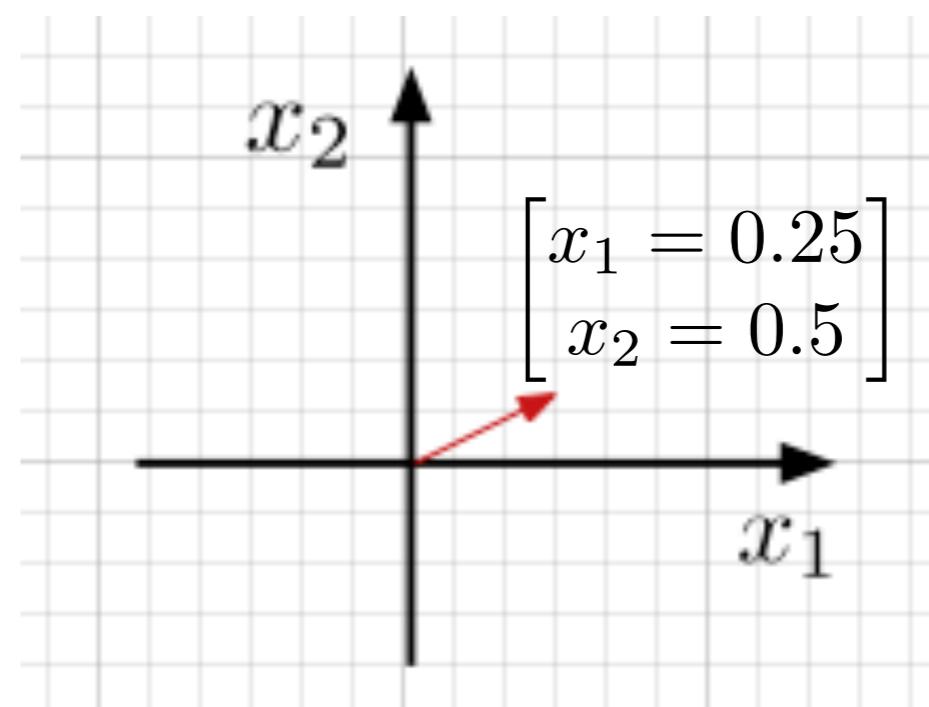
$$\sigma(\mathbf{XW} + \mathbf{b}) = \mathbf{A}$$
$$\mathbf{A} \in \mathbb{R}^{n \times h}$$

In practice, we will almost always be working with multiple training examples as inputs

But Why is the Wx Notation Intuitive?

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

↑
Transformation matrix



But Why is the Wx Notation Intuitive?

Two ways to think about calculating $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

- 1) "Row dot column"

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} ax_1 + bx_2 \\ cx_1 + dx_2 \end{bmatrix}$$

- 2) "Geometrical Intuition"

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 \begin{bmatrix} a \\ c \end{bmatrix} + x_2 \begin{bmatrix} b \\ d \end{bmatrix}$$

But Why is the Wx Notation Intuitive?

The first column affects the first dimension, the second column the second dimension and so forth.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \left(x \begin{bmatrix} 1 \\ 0 \end{bmatrix} + y \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right)$$

Now, applying the distributivity law:

$$= \begin{bmatrix} a & b \\ c & d \end{bmatrix} x \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} a & b \\ c & d \end{bmatrix} y \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Next, the associative property for scalar multiplication:

$$= x \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + y \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Continuing with the matrix multiplication:

$$= x \begin{bmatrix} a \\ c \end{bmatrix} + y \begin{bmatrix} b \\ d \end{bmatrix}$$

But Why is the Wx Notation Intuitive?

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = x \begin{bmatrix} a \\ d \end{bmatrix} + y \begin{bmatrix} b \\ c \end{bmatrix}$$

scales the x coordinate

moves y into x direction

moves x in y direction

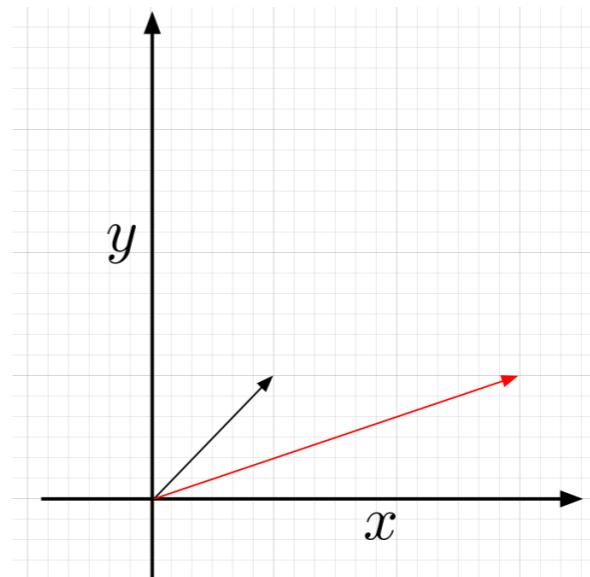
scales the y coordinate

The diagram illustrates the decomposition of a 2x2 matrix multiplication into scaling and shifting components. It shows the equation $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = x \begin{bmatrix} a \\ d \end{bmatrix} + y \begin{bmatrix} b \\ c \end{bmatrix}$. Four arrows point from text labels to specific parts of the equation: one arrow points to the term $x \begin{bmatrix} a \\ d \end{bmatrix}$ with the label "scales the x coordinate"; another arrow points to the term $y \begin{bmatrix} b \\ c \end{bmatrix}$ with the label "moves y into x direction"; a third arrow points to the term $x \begin{bmatrix} a \\ d \end{bmatrix}$ with the label "moves x in y direction"; and a fourth arrow points to the term $y \begin{bmatrix} b \\ c \end{bmatrix}$ with the label "scales the y coordinate".

But Why is the Wx Notation Intuitive?

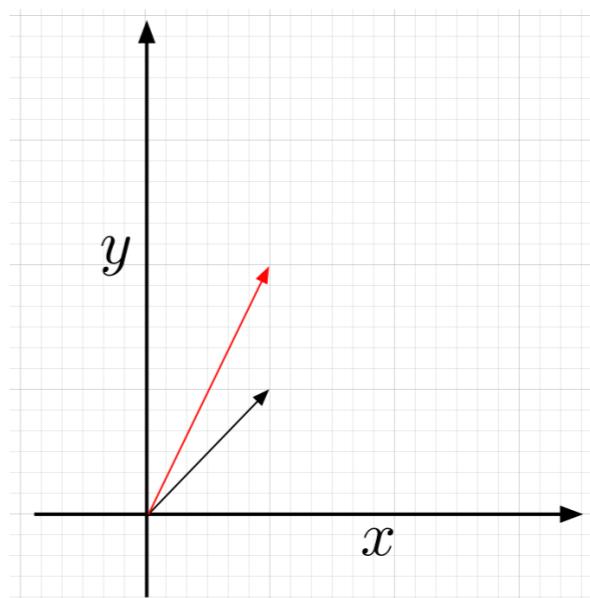
Stretching x-axis by factor of 3:

$$\begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ y \end{bmatrix}$$



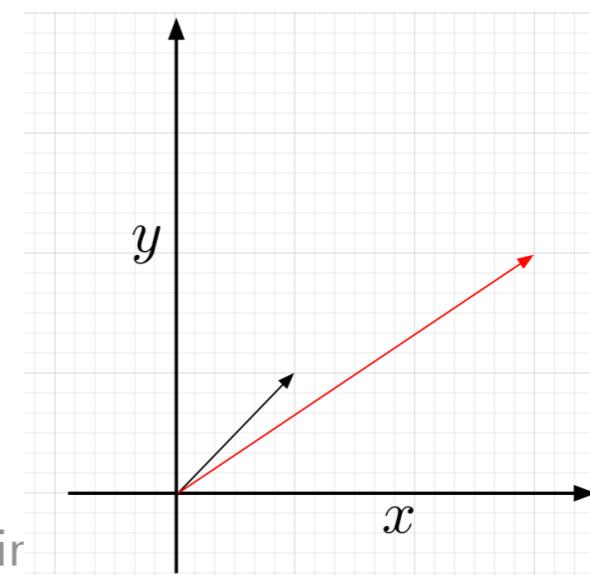
Stretching y-axis by factor of 2:

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ 2y \end{bmatrix}$$



Stretching x-axis by factor of 3 and y-axis by a factor of 2:

$$\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ 2y \end{bmatrix}$$



But Why is the Wx Notation Intuitive?

Take-home exercise:

think about transformation matrices for

- 1) Mirroring
- 2) Shearing (rectangle -> parallelogram)
- 3) Rotation

Fully Connected Layer in PyTorch

```
[1]: import torch

[2]: X = torch.arange(50, dtype=torch.float).view(10, 5)
     # .view() and .reshape() are equivalent
     X

[2]: tensor([[ 0.,  1.,  2.,  3.,  4.],
            [ 5.,  6.,  7.,  8.,  9.],
            [10., 11., 12., 13., 14.],
            [15., 16., 17., 18., 19.],
            [20., 21., 22., 23., 24.],
            [25., 26., 27., 28., 29.],
            [30., 31., 32., 33., 34.],
            [35., 36., 37., 38., 39.],
            [40., 41., 42., 43., 44.],
            [45., 46., 47., 48., 49.]])
```

```
[3]: fc_layer = torch.nn.Linear(in_features=5,
                               out_features=3)
```

```
[4]: fc_layer.weight
```

```
[4]: Parameter containing:
tensor([[-0.1706,  0.1684,  0.3509,  0.1649,  0.1903],
       [-0.1356,  0.0663, -0.4357,  0.2710,  0.1179],
       [-0.0736,  0.0413, -0.0186,  0.4032,  0.0992]], requires_grad=True)
```

```
[5]: fc_layer.bias
```

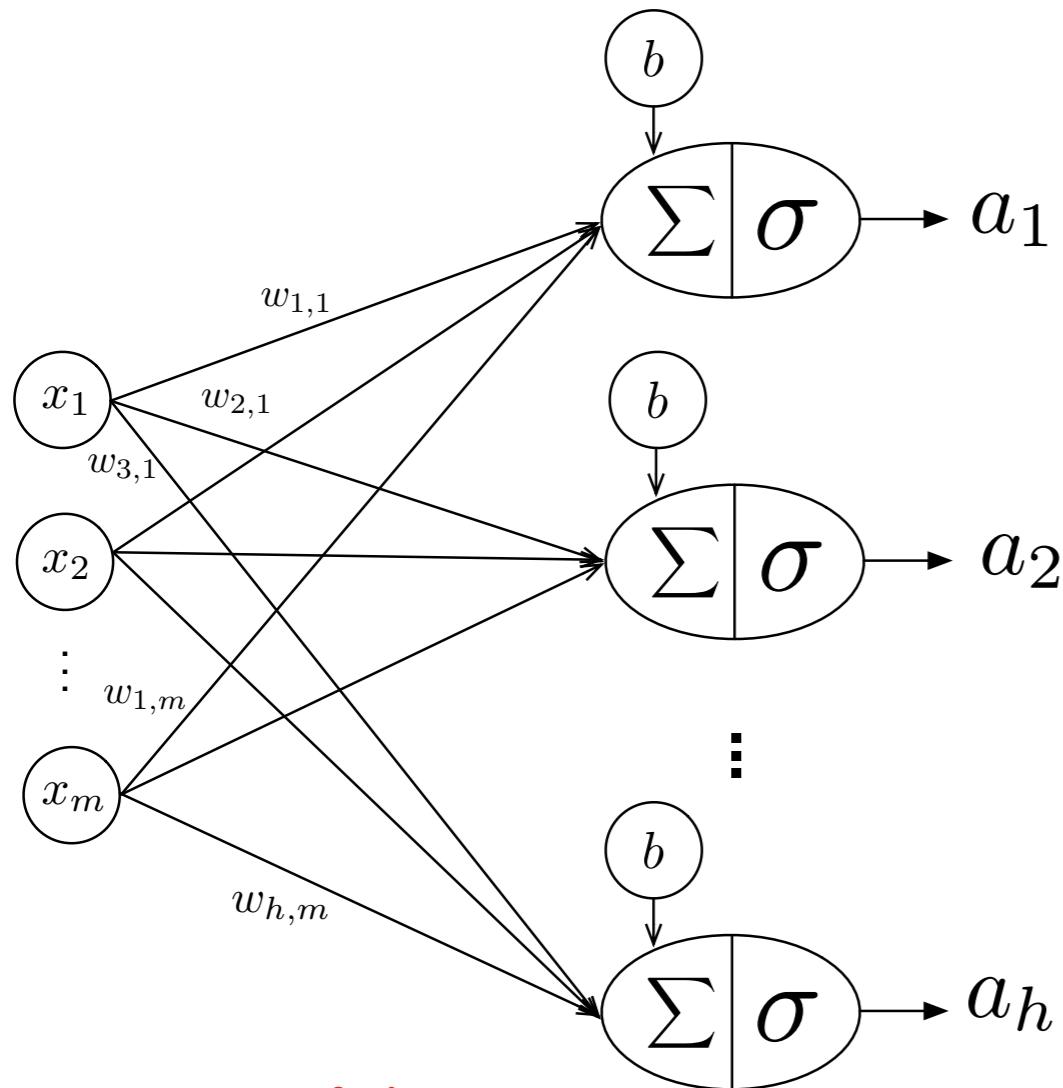
```
[5]: Parameter containing:
tensor([-0.2552,  0.3918,  0.2693], requires_grad=True)
```

Fully Connected Layer in PyTorch

```
[6]: print('X dim:', X.size())
      print('W dim:', fc_layer.weight.size())
      print('b dim:', fc_layer.bias.size())
      # .size() is equivalent to .shape
      A = fc_layer(X)
      print('A:', A)
      print('A dim:', A.size())
```

X dim: torch.Size([10, 5])
W dim: torch.Size([3, 5])
b dim: torch.Size([3])
A: tensor([[1.2004, 2.3291, 2.0036],
 [4.5367, 7.7858, 5.4519],
 [7.8730, 13.2424, 8.9003],
 [11.2093, 18.6991, 12.3486],
 [14.5457, 24.1557, 15.7970],
 [17.8820, 29.6123, 19.2453],
 [21.2183, 35.0690, 22.6937],
 [24.5546, 40.5256, 26.1420],
 [27.8910, 45.9823, 29.5904],
 [31.2273, 51.4389, 33.0387]], grad_fn=<ThAddmmBackward>)
A dim: torch.Size([10, 3])

Based on PyTorch, We Have Another Convention ...



note that $w_{i,j}$ refers to the weight connecting the j -th input to the i -th output.

where

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,m} \\ w_{2,1} & w_{2,2} & \dots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{h,1} & w_{h,2} & \dots & w_{h,m} \end{bmatrix}$$

$$\mathbf{x} = [x_1 \quad x_2 \dots \quad x_m]$$

Layer activations for 1 training example

$$\sigma(\mathbf{x}\mathbf{W}^\top + \mathbf{b}) = \mathbf{a}$$

$$\mathbf{a} \in \mathbb{R}^{1 \times h}$$

Layer activations for n training example

$$\sigma(\mathbf{X}\mathbf{W}^\top + \mathbf{b}) = \mathbf{a}$$

$$\mathbf{a} \in \mathbb{R}^{n \times h}$$

You can find the source code here:

<https://github.com/pytorch/pytorch/blob/18edd3ab0828aca81dc052dba8644c874dc62db/torch/nn/functional.py#L1368>

Conclusion

- Always think about how the dot products are computed when writing and implementing matrix multiplication
- Theoretical intuition and convention does not always match up with practical convenience (coding)
- When switching between theory and code, these rules may be useful:

$$\mathbf{AB} = (\mathbf{B}^\top \mathbf{A}^\top)^\top$$

$$(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top$$

Our Convention (compromise betw. math convention & PyTorch)

(Transformation matrix should ideally be always in the front!)

where $\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,m} \\ w_{2,1} & w_{2,2} & \dots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{h,1} & w_{h,2} & \dots & w_{h,m} \end{bmatrix}$

note that $w_{i,j}$ refers to the weight connecting the j -th input to the i -th output.

Layer activations for 1 training example

$$\sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) = \mathbf{a}, \quad \mathbf{a} \in \mathbb{R}^{h \times 1} \quad \text{with } \mathbf{x} \in \mathbb{R}^{m \times 1}$$

$$\Leftrightarrow \sigma([\mathbf{x}^\top \mathbf{W}^\top]^\top + \mathbf{b}) = \mathbf{a} \quad \text{with } \mathbf{x} \in \mathbb{R}^{m \times 1}$$

$$\Leftrightarrow \sigma([\mathbf{x}\mathbf{W}^\top] + \mathbf{b}) = \mathbf{a} \quad \text{with } \mathbf{x} \in \mathbb{R}^{1 \times m} \text{ (PyTorch)}$$

Layer activations for n training examples

$$\sigma([\mathbf{W}\mathbf{X}^\top]^\top + \mathbf{b}) = \mathbf{a}, \quad \mathbf{a} \in \mathbb{R}^{n \times h} \quad \text{with } \mathbf{X} \in \mathbb{R}^{n \times m}$$

$$\Leftrightarrow \sigma([\mathbf{X}\mathbf{W}^\top] + \mathbf{b}) = \mathbf{a} \quad \text{with } \mathbf{X} \in \mathbb{R}^{n \times m}$$

Next Lecture: A better* learning algorithm for neural networks

* compared to the perceptron rule

Ungraded Homework Exercise

Revisit our Perceptron NumPy code:

https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/L03_perceptron/code/perceptron-numpy.ipynb

1. Without running the code, can you tell if the perceptron could predict the class labels if we feed an array of multiple training examples at once (i.e., via its forward method)?
 - If yes, why?
 - If no, what change would you need to make
2. Run the code to verify your intuition.
3. What about the train method? Can we have parallelism through matrix multiplication without affecting the perceptron learning rule?