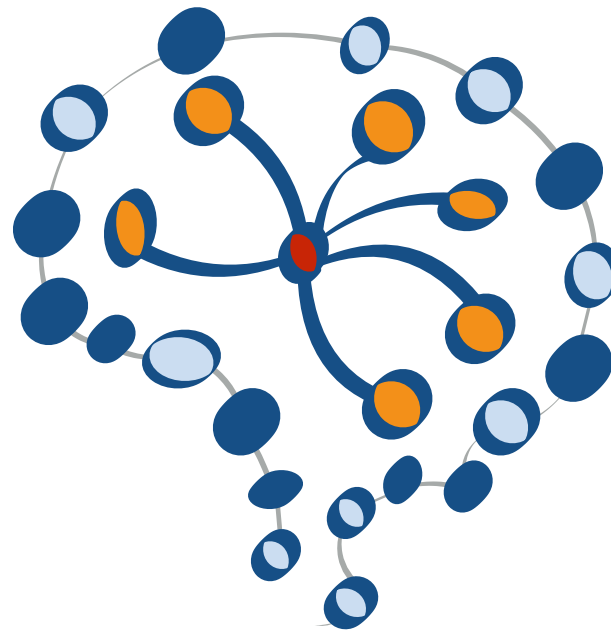


# STAT 453: Introduction to Deep Learning and Generative Models

Sebastian Raschka

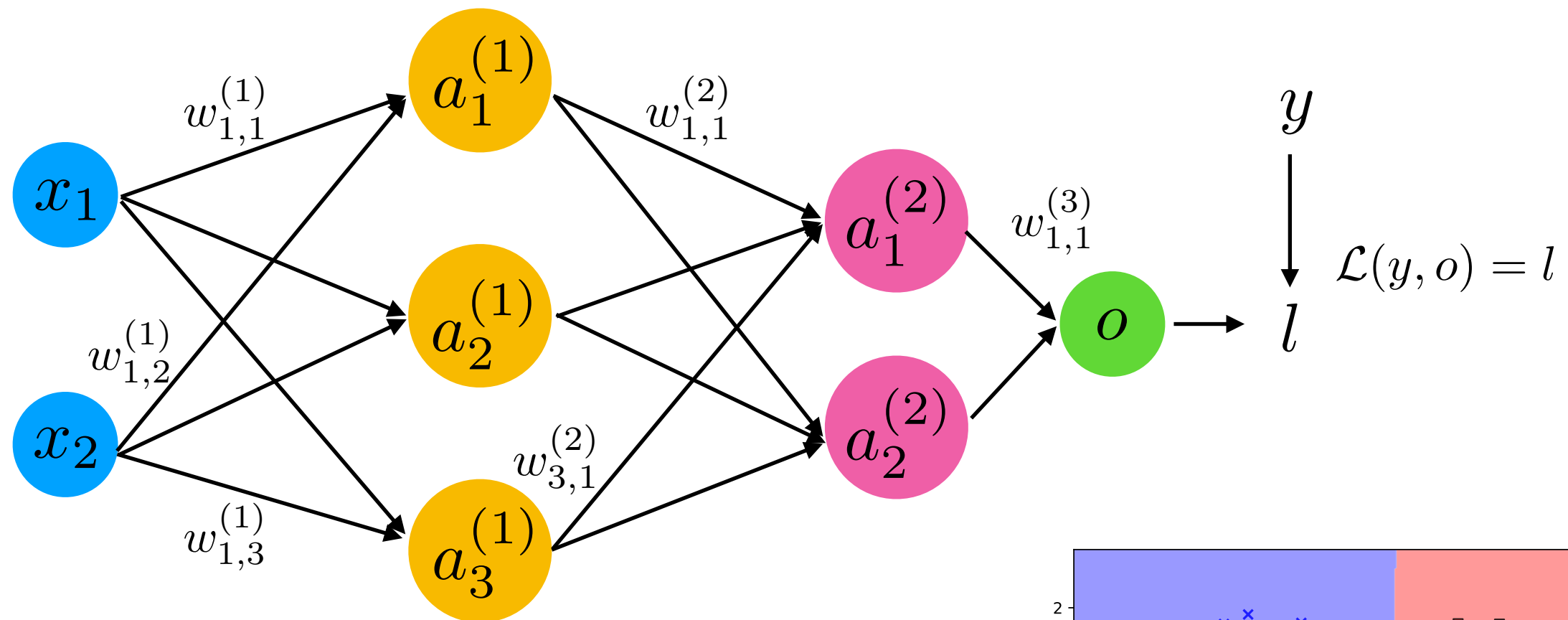
<http://stat.wisc.edu/~sraschka/teaching>



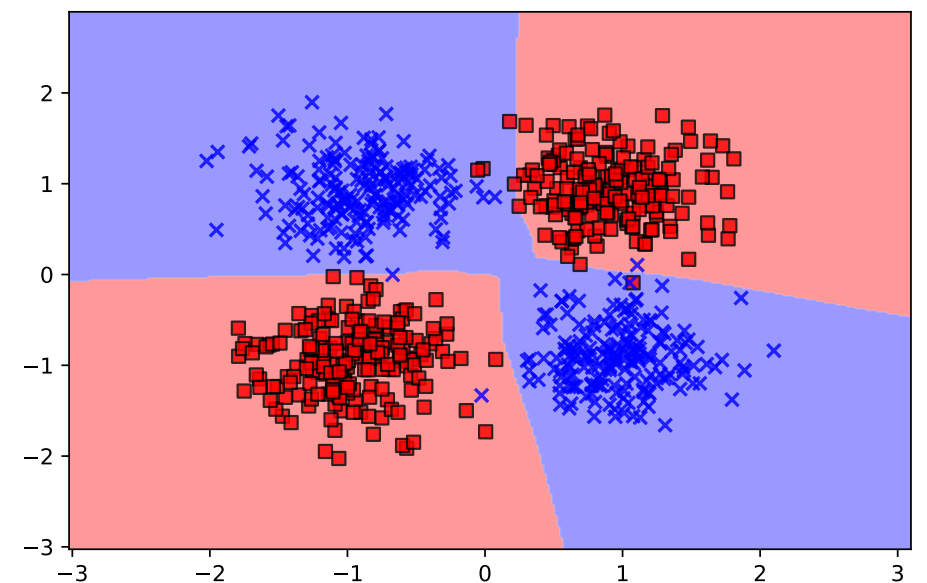
## Lecture 04

# Linear Algebra for Deep Learning

# Today: Fundamental Math Skills for DL



So that we can solve the XOR problem, among other things ...



1-hidden layer MLP  
with non-linear activation function (ReLU)

# Lecture Overview

1. Tensors in Deep Learning
2. Tensors and PyTorch
3. Vectors, Matrices, and Broadcasting
4. Notational Conventions for Neural Networks
5. A Fully Connected (Linear) Layer in PyTorch





# The Use of Tensors in Deep Learning

- 1. Tensors in Deep Learning**
2. Tensors and PyTorch
3. Vectors, Matrices, and Broadcasting
4. Notational Conventions for Neural Networks
5. A Fully Connected (Linear) Layer in PyTorch

# Vectors, Matrices, and Tensors -- Notational Conventions

## Scalar

(rank-0 tensor)

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

e.g.,

$$x = 1.23$$

## 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.,

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

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

# 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.

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

E.g.,

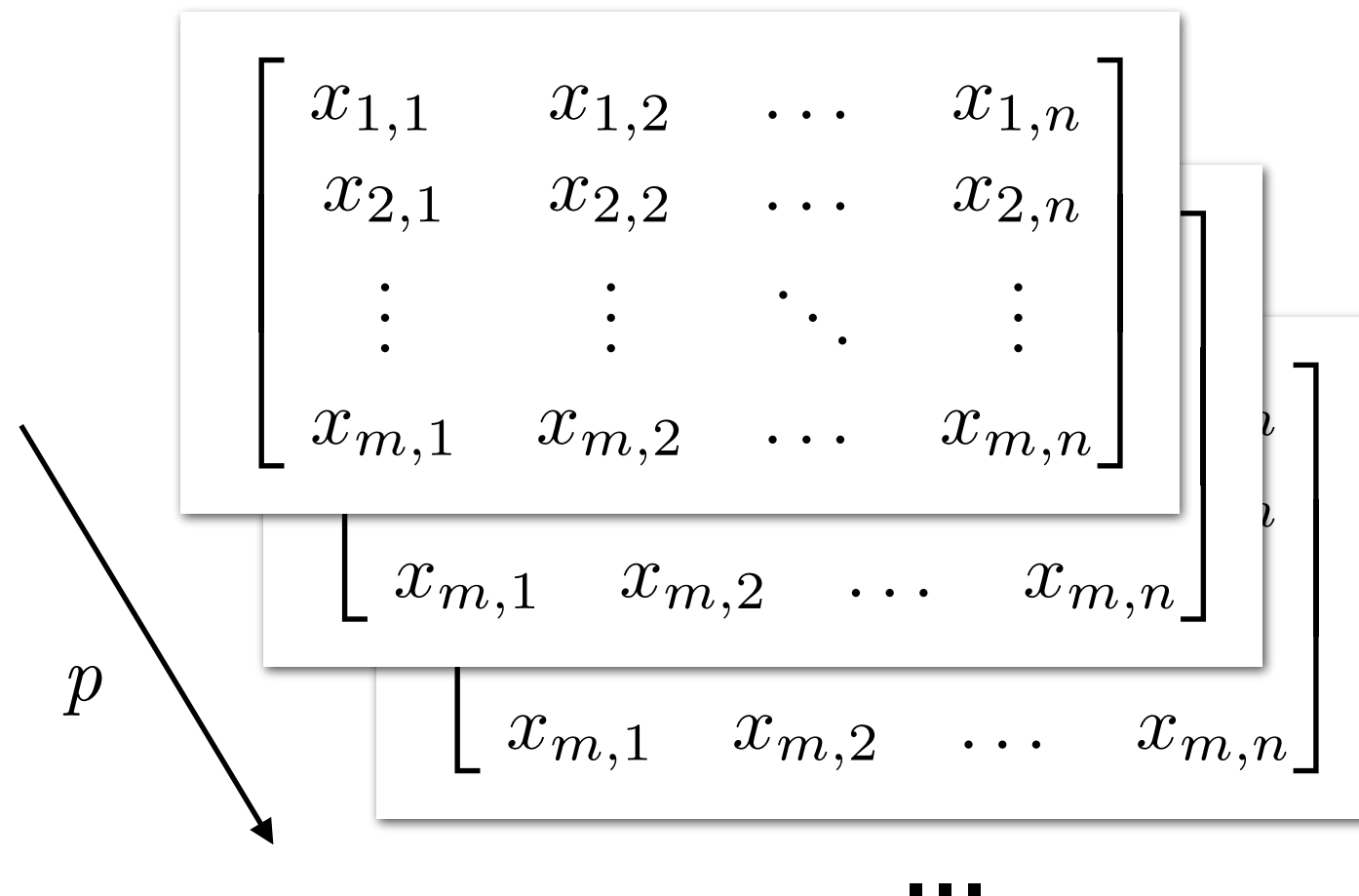
$x_2^{[1]}$  = 2nd feature value of the 1st training example

# Vectors, Matrices, and Tensors -- Notational Conventions

## 3D Tensor

(rank-3 tensor)

$$\mathbf{X} \in \mathbb{R}^{m \times n \times p} \quad (n \text{ and } m \text{ are generic indices here})$$



# An Example of a 3D Tensor in DL

Single color image

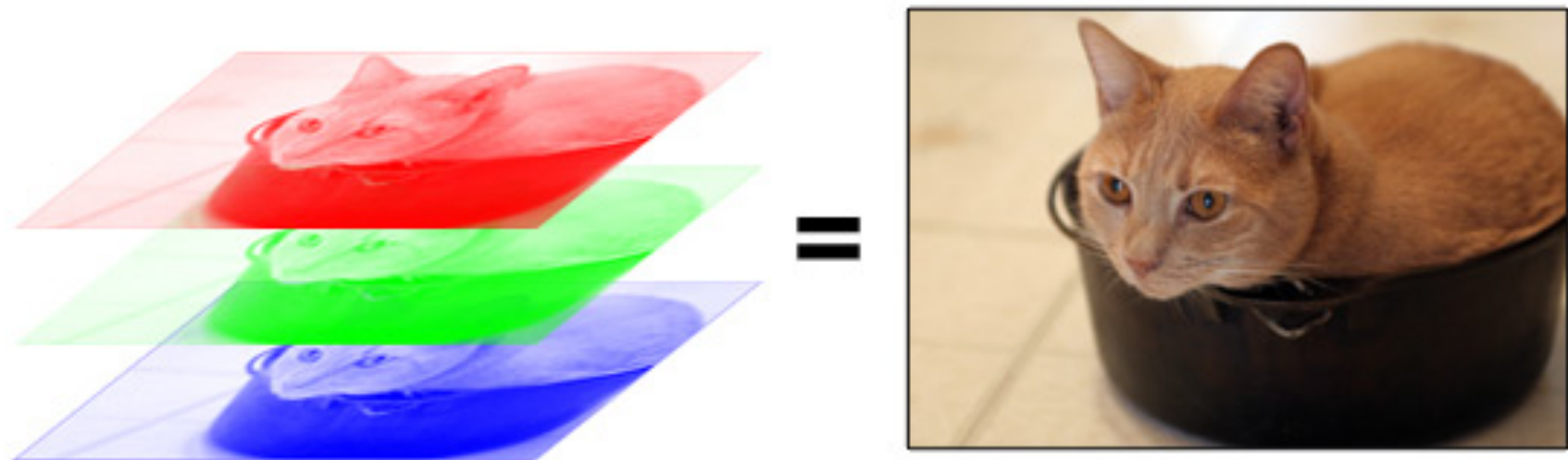


Image Source: <https://code.tutsplus.com/tutorials/create-a-retro-crt-distortion-effect-using-rgb-shifting--active-3359>

(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)

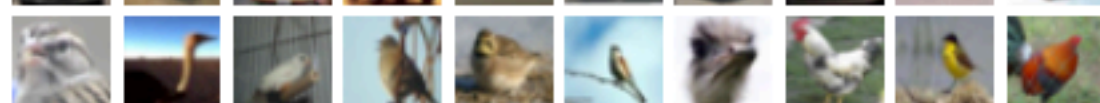
**airplane**



**automobile**



**bird**



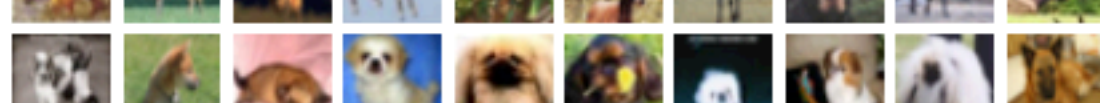
**cat**



**deer**



**dog**



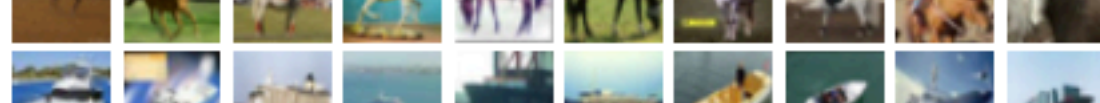
**frog**



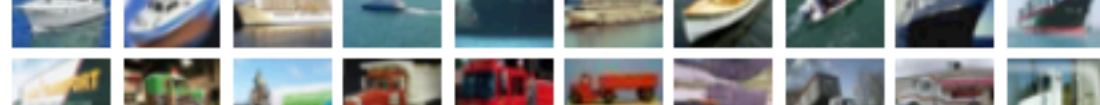
**horse**



**ship**



**truck**



<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)

In the context of TensorFlow, NumPy, PyTorch etc.,  
tensors = multidimensional arrays

dimensionality coincides with the number of indices of `.shape`

```
[In [1]: import torch
```

```
[In [2]: t = torch.tensor([[1, 2, 3], [4, 5, 6]])
```

```
[In [3]: t
```

```
Out[3]:
```

```
tensor([[1, 2, 3],  
        [4, 5, 6]])
```

```
[In [4]: t.shape
```

```
Out[4]: torch.Size([2, 3])
```

```
[In [5]: t.ndim
```

```
Out[5]: 2
```

```
In [6]: █
```

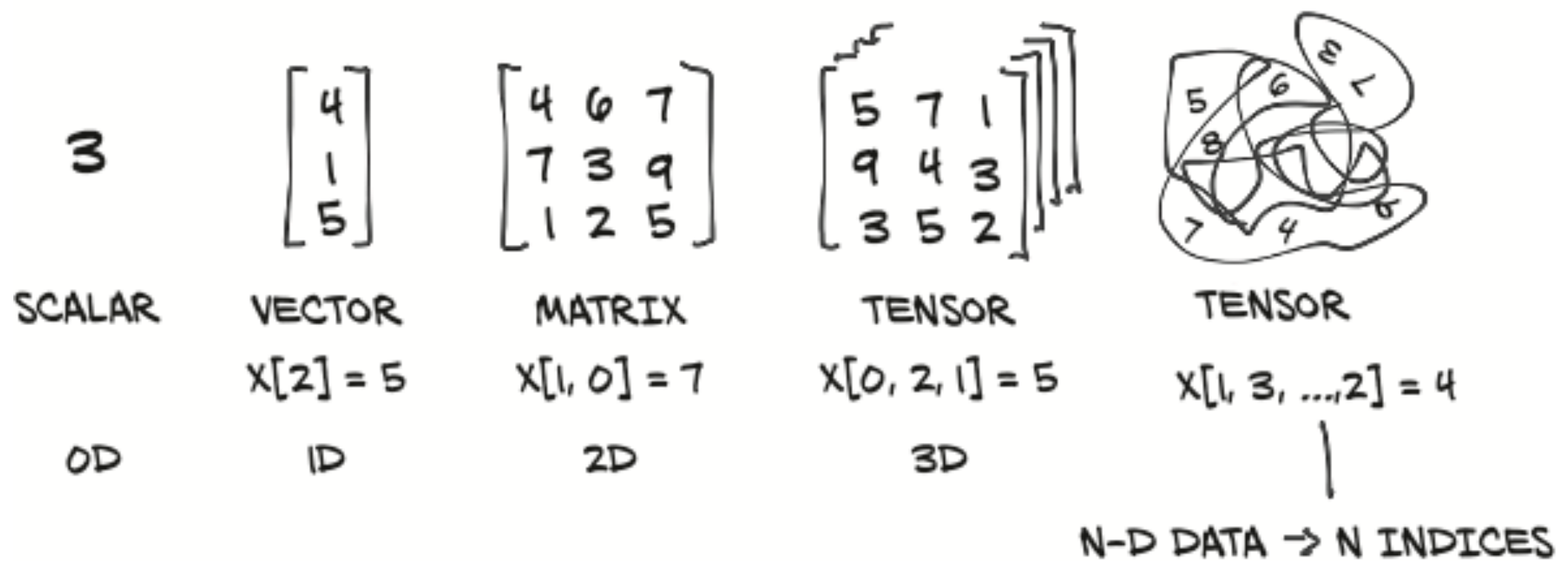


Figure 3.2 Tensors are the building blocks for representing data in PyTorch.

Image source: Stevens et al.'s "Deep Learning with PyTorch"





# Working with Tensors in PyTorch

1. Tensors in Deep Learning
- 2. Tensors and PyTorch**
3. Vectors, Matrices, and Broadcasting
4. Notational Conventions for Neural Networks
5. A Fully Connected (Linear) Layer in PyTorch

# 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
```

```
[In [2]: a = np.array([1., 2., 3.])
```

```
[In [3]: print(a.dtype)  
float64
```

```
[In [4]: print(a.shape)  
(3,)
```

```
[In [5]: import torch
```

```
[In [6]: b = torch.tensor([1., 2., 3.])
```

```
[In [7]: print(b.dtype)  
torch.float32
```

```
[In [8]: print(b.shape)  
torch.Size([3])
```

# NumPy and PyTorch Syntax is Very Similar

```
[In [9]: a = np.array([1., 2., 3.])
```

```
[In [10]: print(a.dot(a))  
14.0
```

```
[In [12]: print(b.matmul(b))  
tensor(14.)
```

```
[In [13]: b  
Out[13]: tensor([1., 2., 3.])
```

```
[In [14]: b.numpy()  
Out[14]: array([1., 2., 3.], dtype=float32)
```

We can convert,  
but pay attention to  
default types

Note: Traditionally, PyTorch used "matmul", but nowadays "dot" also works

```
[In [12]: print(b.matmul(b))  
tensor(14.)
```

```
[In [15]: print(b.dot(b))  
tensor(14.)
```

```
[In [16]: print(b @ b)  
tensor(14.)
```

# Data Types to Memorize

NumPy data	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>
<code>numpy.float16</code>	<code>torch.HalfTensor</code>
<code>numpy.float32</code>	<code>torch.FloatTensor</code>
<code>numpy.float</code>	<code>torch.DoubleTensor</code>
<code>numpy.float64</code>	<code>torch.DoubleTensor</code>

default int in NumPy & PyTorch

default float in PyTorch

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

# Specify the type upon construction

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

```
[In [22]: c.dtype
```

```
Out[22]: torch.float32
```

```
[In [23]: c = torch.tensor([1., 2., 3.], dtype=torch.double)
```

```
[In [24]: c.dtype
```

```
Out[24]: torch.float64
```

```
[In [25]: c = torch.tensor([1., 2., 3.], dtype=torch.float64)
```

```
[In [26]: c.dtype
```

```
Out[26]: torch.float64
```

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

```
[In [27]: d = torch.tensor([1, 2, 3])
```

```
[In [28]: d.dtype
```

```
Out[28]: torch.int64
```

```
[In [29]: e = d.double()
```

```
[In [30]: e.dtype
```

```
Out[30]: torch.float64
```

```
[In [31]: f = d.float64()
```

---

```
AttributeError                                Traceback (most recent call last)
<ipython-input-31-b3b070130d25> in <module>
----> 1 f = d.float64()
```

```
AttributeError: 'Tensor' object has no attribute 'float64'
```

```
[In [32]: f = d.to(torch.float64)
```

```
[In [33]: f.dtype
```

```
Out[33]: torch.float64
```



# 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 ...

```
[sraschka@gpu03:~$ nvidia-smi
```

```
Mon Feb  8 21:05:27 2021
```

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
NVIDIA-SMI 455.32.00				Driver Version: 455.32.00				CUDA Version: 11.1	
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
GPU	Name	Persistence-M		Bus-Id	Disp.A	Volatile Uncorr. ECC			
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.		
							MIG M.		
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
0	GeForce RTX 208...	Off		00000000:1A:00.0	Off			N/A	
24%	37C	P0	71W / 250W		0MiB / 11019MiB	0%	Default		
							N/A		
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									

# 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):

PyTorch Build	Stable (1.7.1)		Preview (Nightly)		
Your OS	Linux		Mac		Windows
Package	Conda		Pip	LibTorch	Source
Language	Python		C++ / Java		
CUDA	9.2	10.1	10.2	11.0	None
Run this Command:	<b>NOTE:</b> Python 3.9 users will need to add '-c=conda-forge' for installation <code>conda install pytorch torchvision torchaudio -c pytorch</code>				

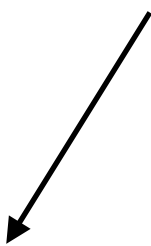


# Broadcasting semantics: Making Vector and Matrix computations more convenient

1. Tensors in Deep Learning
2. Tensors and PyTorch
- 3. Vectors, Matrices, and Broadcasting**
4. Notational Conventions for Neural Networks
5. A Fully Connected (Linear) Layer in PyTorch

# Vectors

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

$$\mathbf{w}^\top \mathbf{x} + b = z \quad \text{where} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$


## 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

Question for CS majors: What is the Big-O of matrix multiplication (assume 2 NxN matrices)?

# 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

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


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

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

# 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

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



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

where

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

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

Can you spot the error on this slide?

# 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

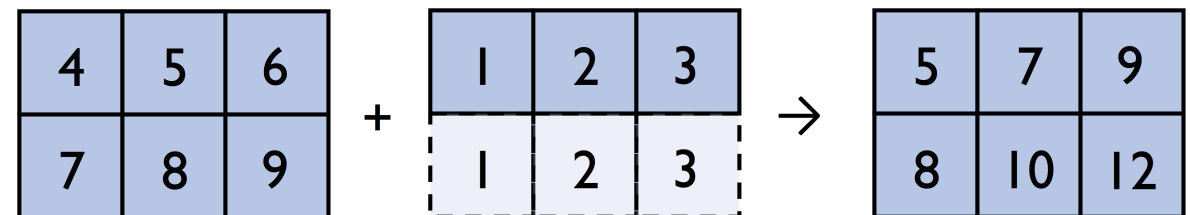
- In PyTorch, it works just fine.
- This (general) feature is called "broadcasting"

```
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

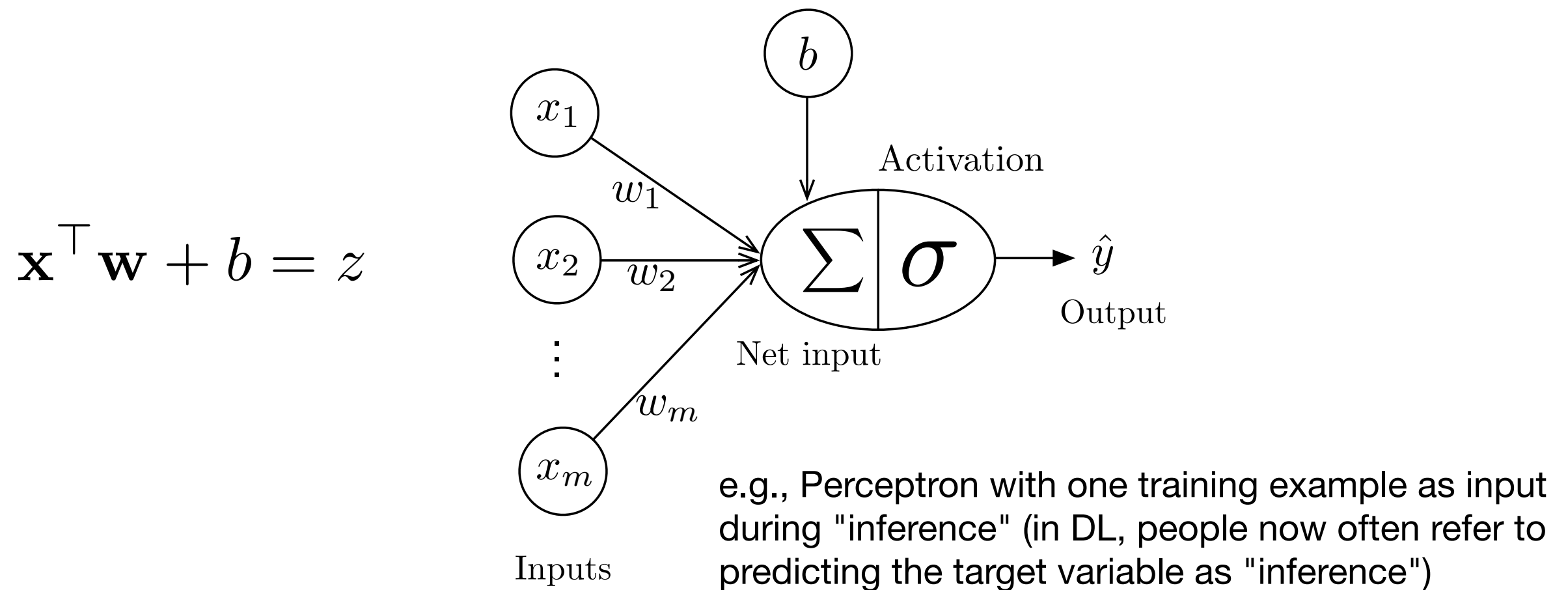




# Notational Linear Algebra Conventions in Deep Learning

1. Tensors in Deep Learning
2. Tensors and PyTorch
3. Vectors, Matrices, and Broadcasting
- 4. Notational Conventions for Neural Networks**
5. A Fully Connected (Linear) Layer in PyTorch

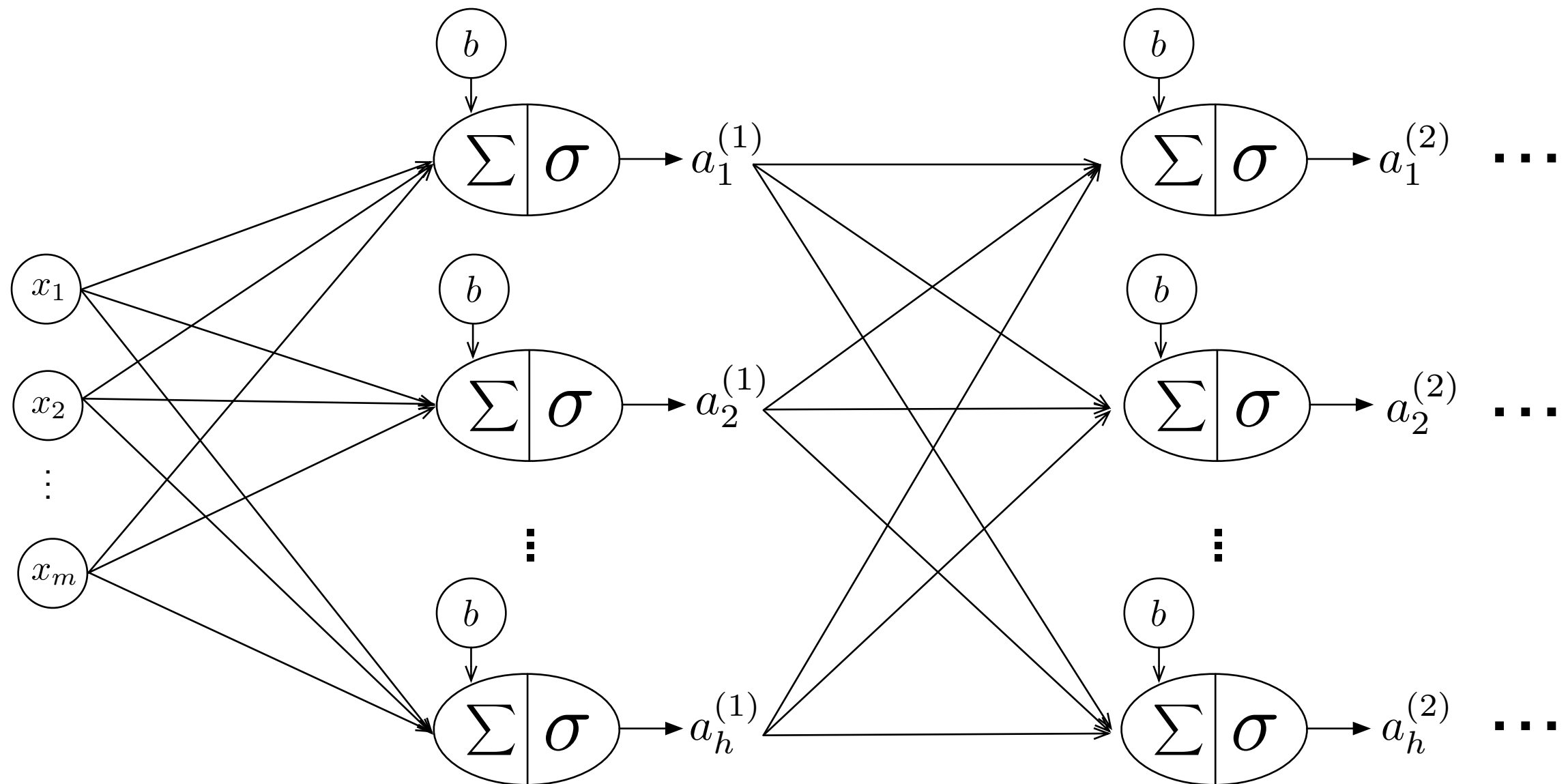
# Connections We Have Seen Before ...



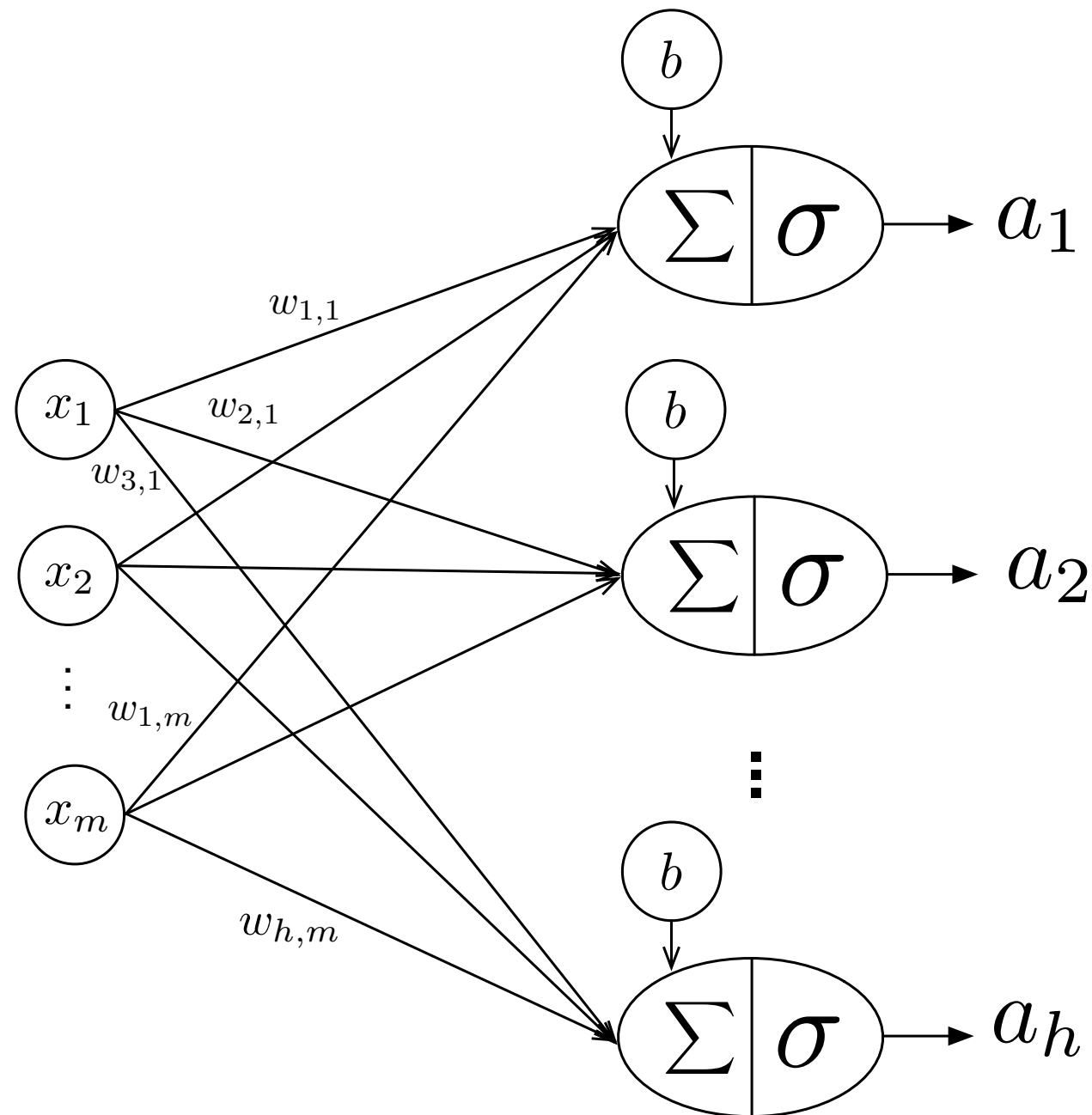
If we have  $n$  training examples,  $\mathbf{X} \in \mathbb{R}^{n \times m}$ ,  $\mathbf{z} \in \mathbb{R}^{n \times 1}$

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

# Connections We Will Encounter Later ...



# A Fully Connected Layer



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

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

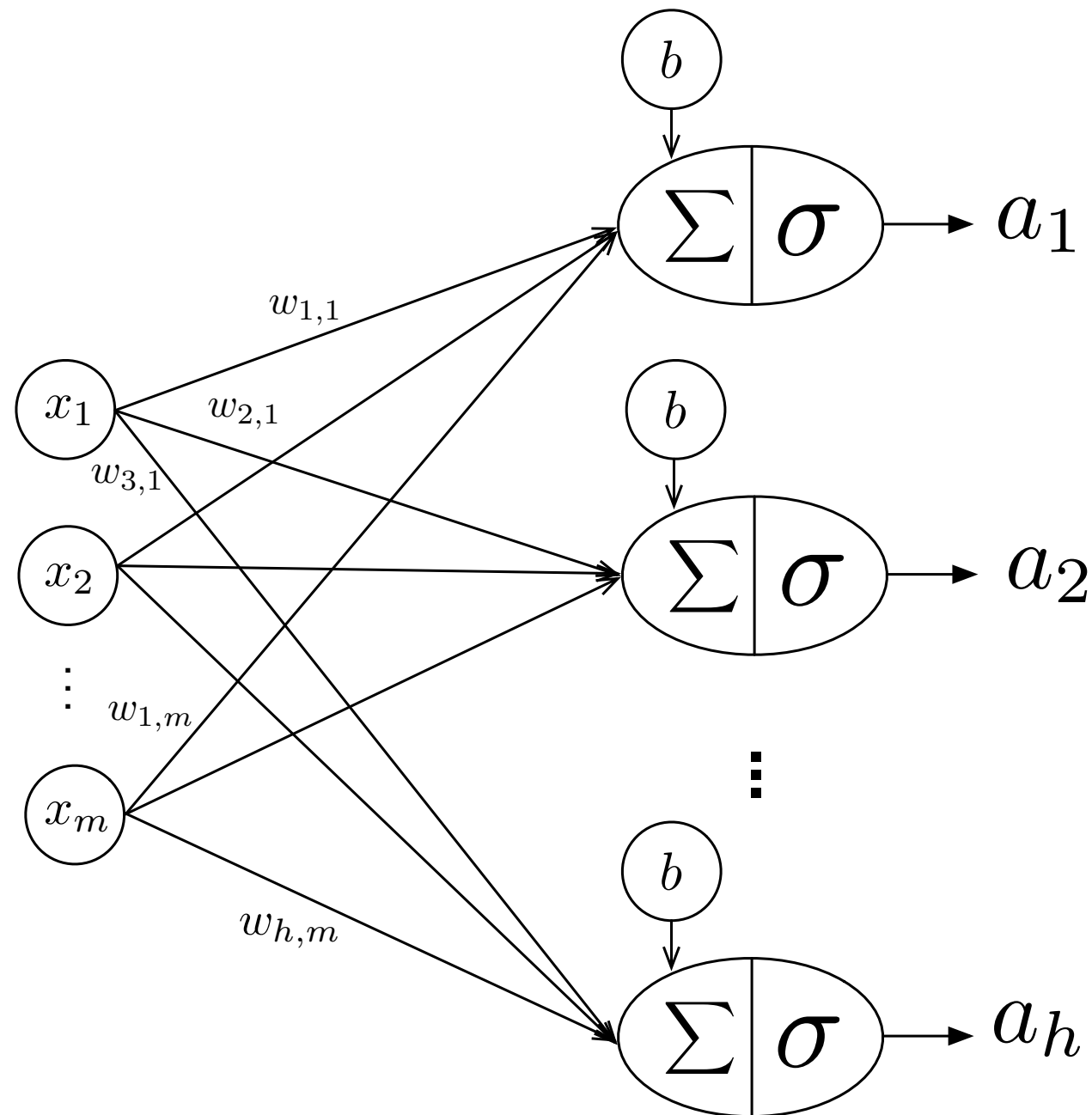
Layer activations for 1 training example

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

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

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

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

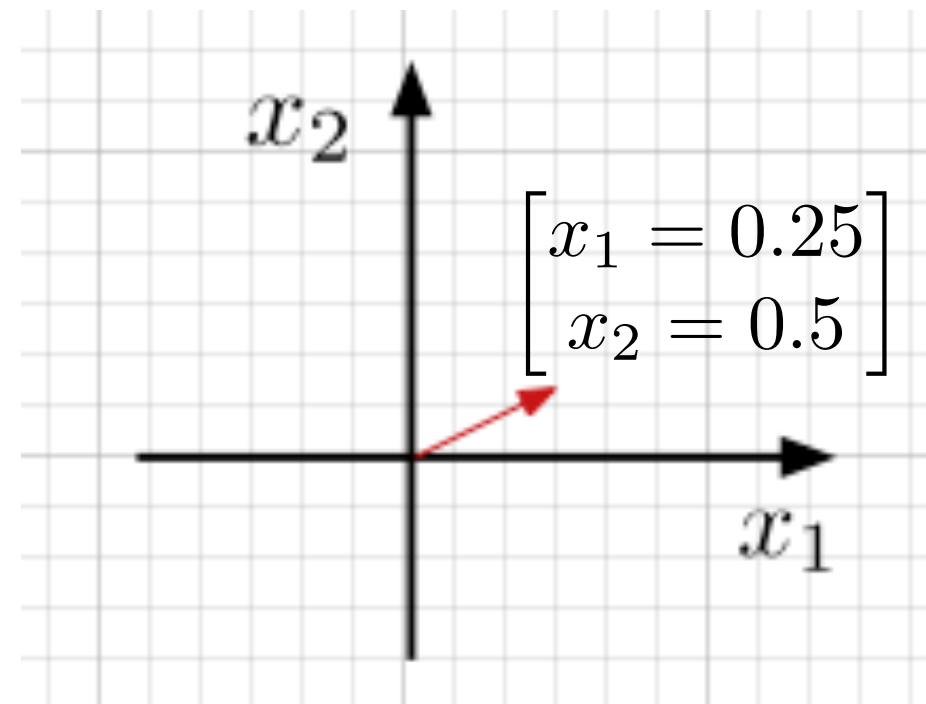
$$\mathbf{A} \in \mathbb{R}^{n \times h}$$

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.

# But Why is the $W_{\mathbf{x}}$ 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 $W_x$ Notation Intuitive?

scales the  $x$  coordinate

moves  $y$  into  $x$  direction

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

moves  $x$  in  $y$  direction

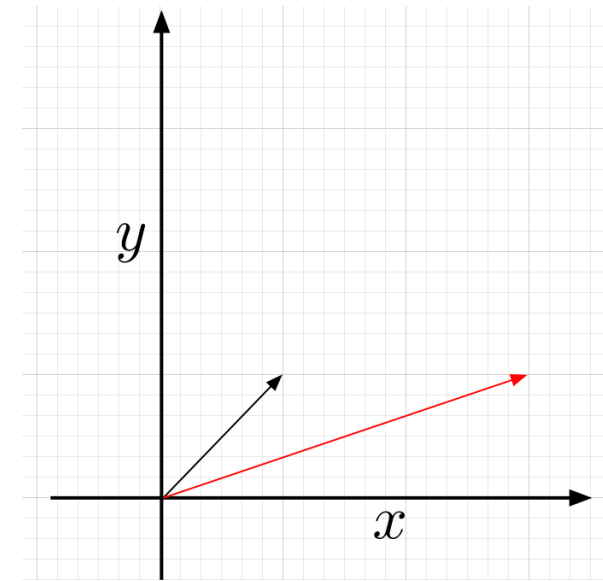
scales the  $y$  coordinate



# But Why is the $W_x$ Notation Intuitive?

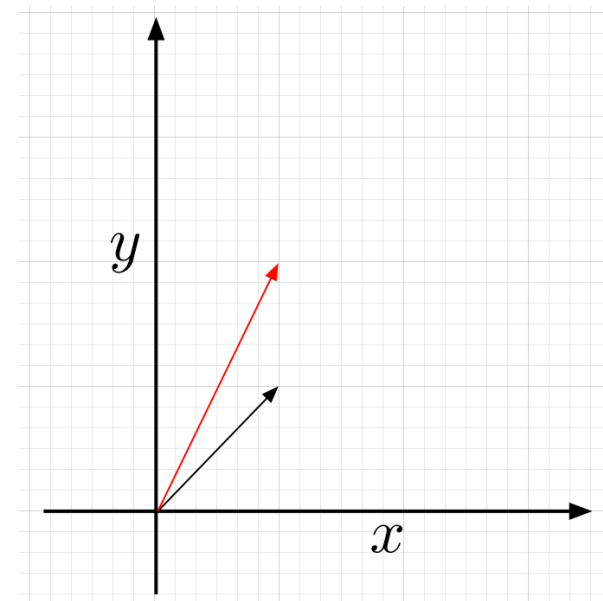
Stretching x-axis by factor of 3:

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



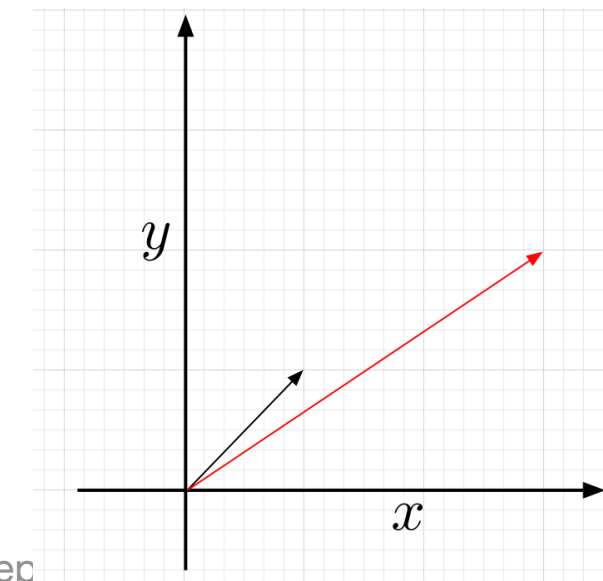
Stretching y-axis by factor of 2:

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



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

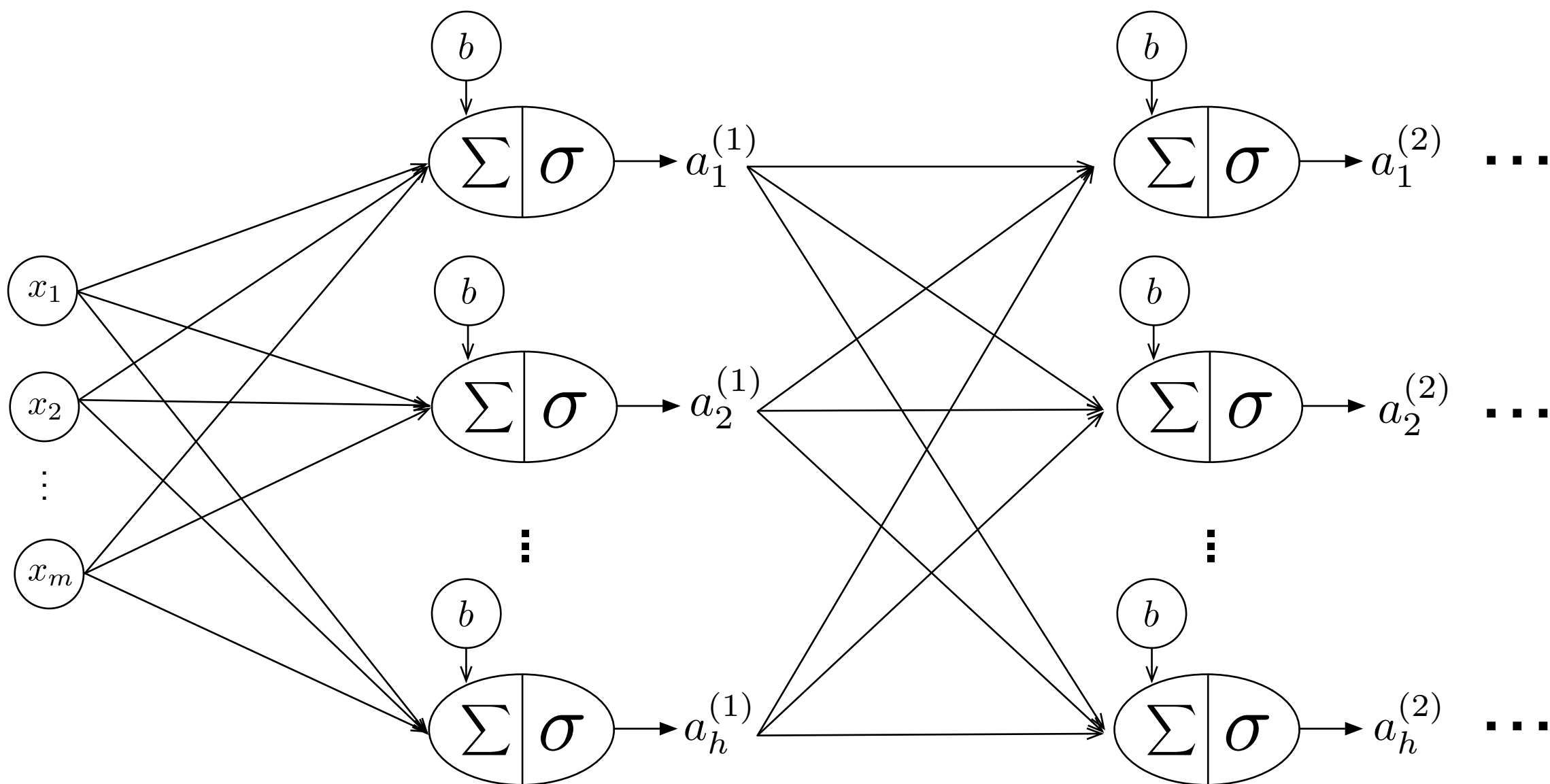
$$\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ 2y \end{bmatrix}$$





# A Fully Connected (Linear) Layer in PyTorch

1. Tensors in Deep Learning
2. Tensors and PyTorch
3. Vectors, Matrices, and Broadcasting
4. Notational Conventions for Neural Networks
- 5. A Fully Connected (Linear) Layer in PyTorch**



# 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
```

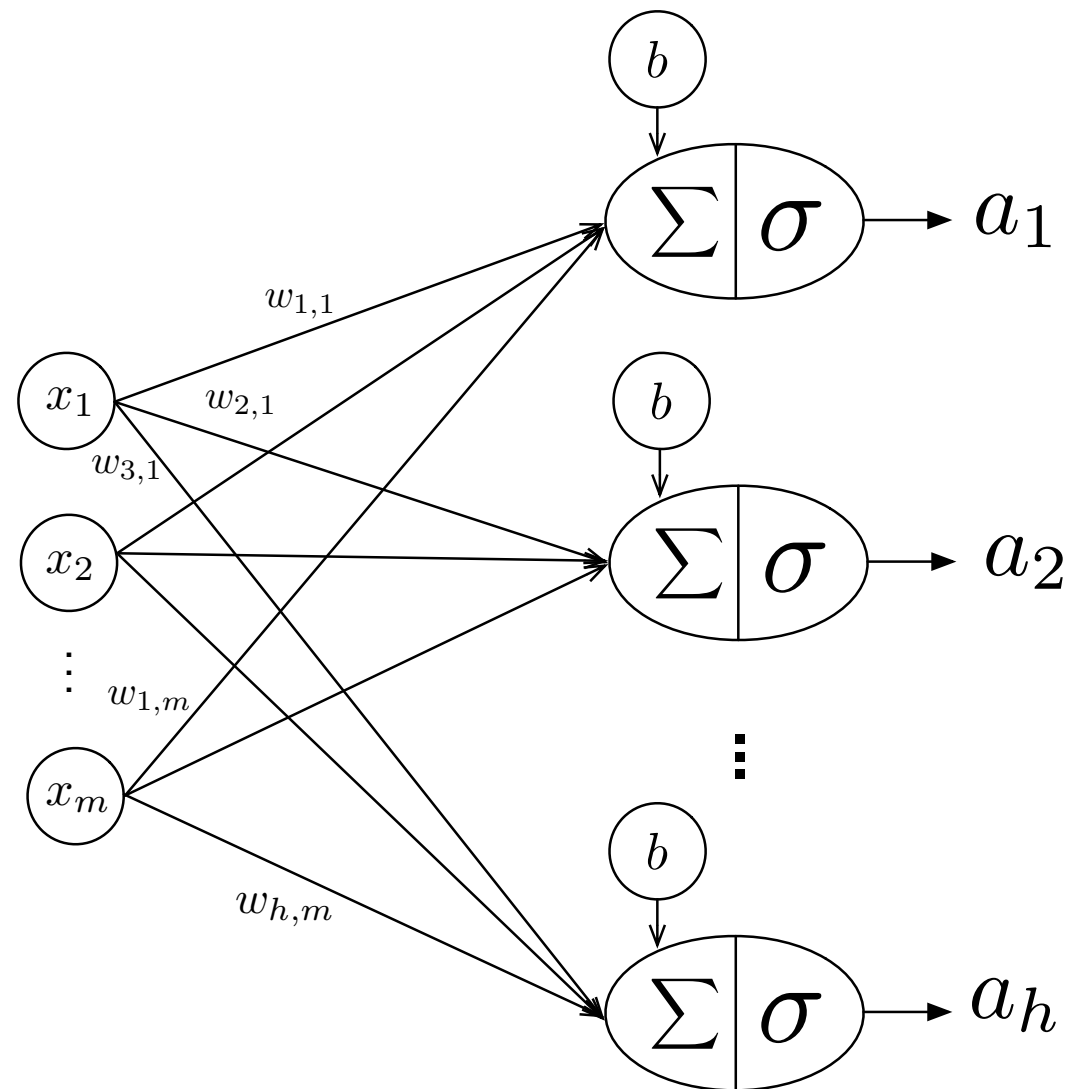
```
[5]: Parameter containing:
tensor([ -0.2552,  0.3918,  0.2693], requires_grad=True)
```

# 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.

You can find the source code here:

<https://github.com/pytorch/pytorch/blob/18edd3ab0828acaa81dc052dba8644c874dc62db/torch/nn/functional.py#L1368>

...

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

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

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

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

$$\mathbf{W}^\top \in \mathbb{R}^{m \times h}$$

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

# 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:

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

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



# Summary: Traditional vs 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

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

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

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

## Layer activations for $n$ training examples

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

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

# Ungraded Homework Exercise / Experiment

Revisit our Perceptron NumPy code:

<https://github.com/rasbt/stat453-deep-learning-ss20/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?

# Next Lecture:

## A better\* learning algorithm for neural networks

\* compared to the perceptron rule