STAT 453: Introduction to Deep Learning and Generative Models

Ben Lengerich

Lecture 04: Single-layer networks

September 15, 2025

Announcements

- HW1 Due this Friday via Canvas
- Enrollment / waitlist

Today: Single-layer neural networks

1. **Perceptrons**
2. Geometric Intuition
3. Notational Conventions for Neural Networks
4. A Fully Connected (Linear) Layer in PyTorch

Rosenblatt's Perceptron

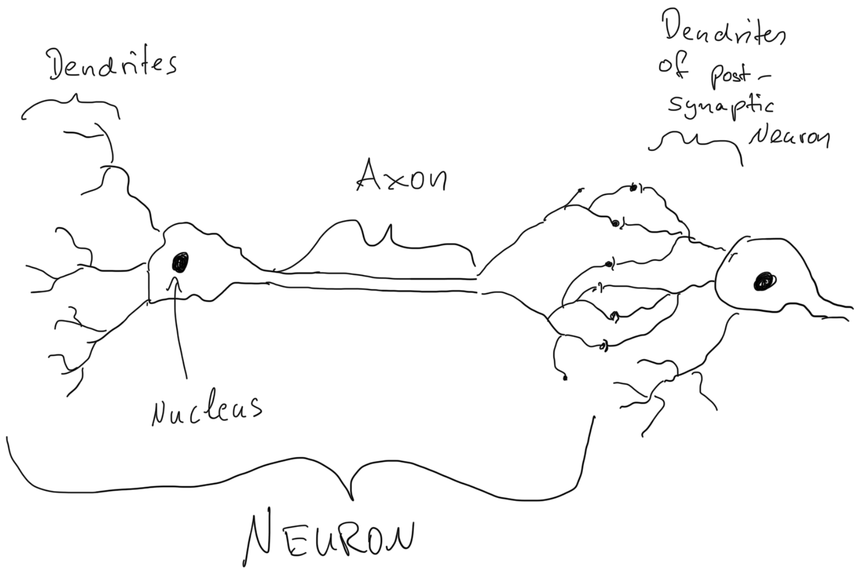
A learning rule for the computational/mathematical neuron model

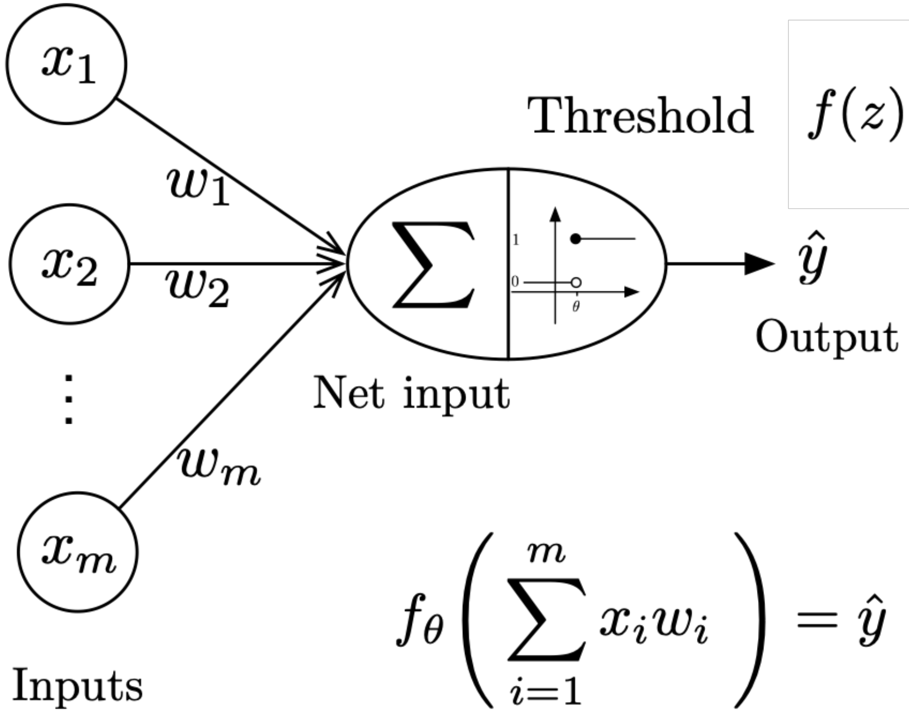
Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton. Project Para*. Cornell Aeronautical Laboratory.

Source: http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/Members/wilex4/Rosen-2.jpg

Rosenblatt’s Perceptron

A Computational Model of a Biological Neuron



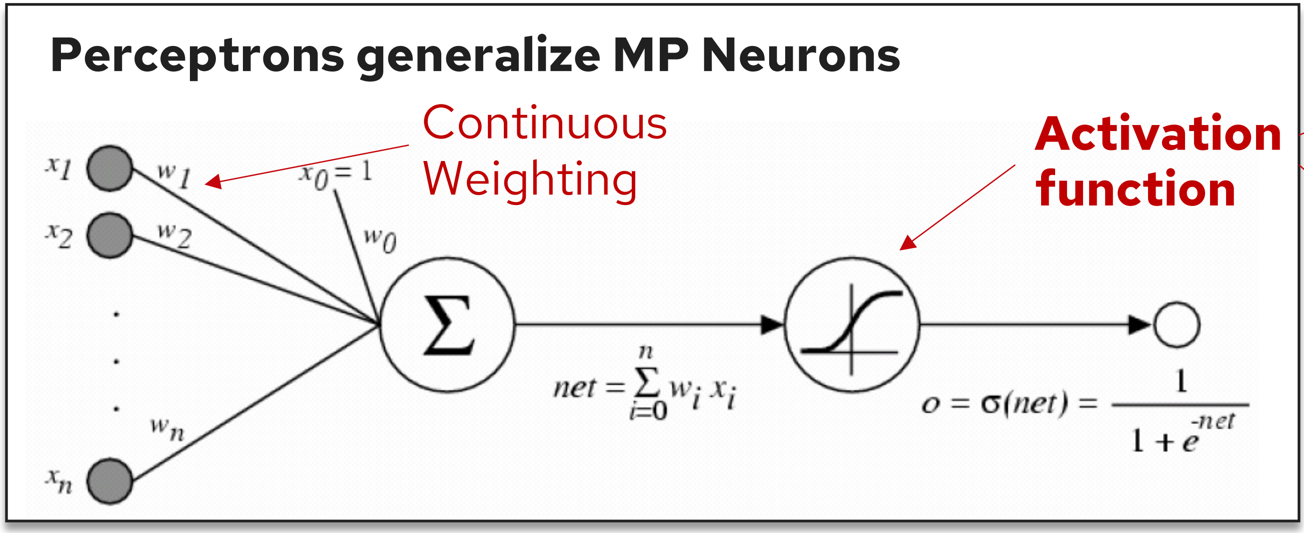


$$f(z) = \begin{cases} 0, & z \leq \theta \\ 1, & z > \theta \end{cases}$$

$$f_\theta\left(\sum_{i=1}^{m} x_i w_i\right) = \hat{y}$$

Rosenblatt’s Perceptron

- Note that Rosenblatt (and later others) proposed many variants of the Perceptron model and learning rule.
- We discuss a "basic" version; today
  "Perceptron" = "a classic Rosenblatt Perceptron”
- In our “brief history of DL” (Lecture 2), we were a bit loose:



threshold function → Classic Rosenblatt Perceptron

sigmoid → DL “Perceptron” / sigmoid unit

Many activation functions

- Threshold function (perceptron, 1950+)
- Sigmoid function (before 2000)
- ReLU function (popular since CNNs)
- Many variants of ReLU, e.g. leaky ReLU, GeLU

Terminology

General (logistic regression, multilayer nets, ...):

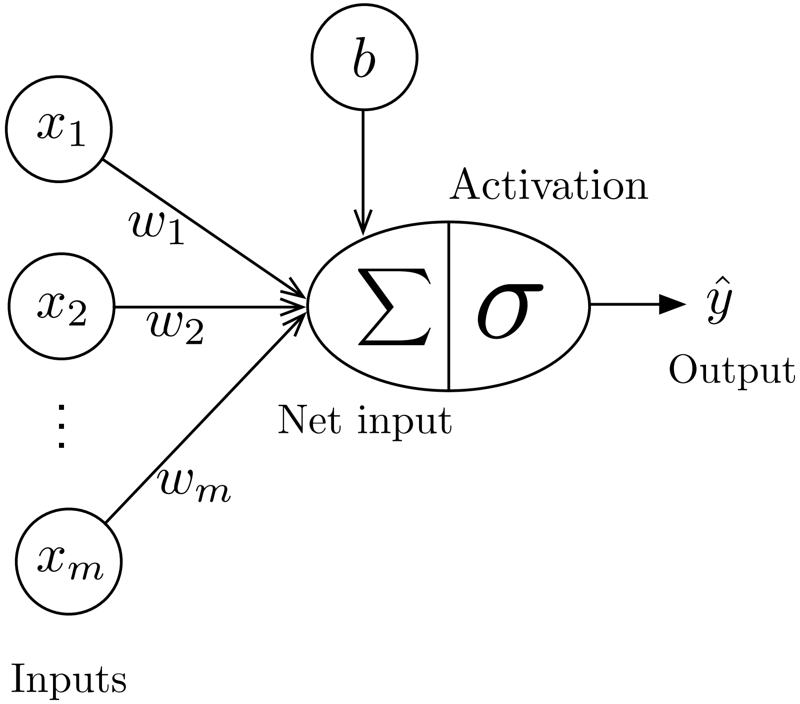
- Net input = pre-activation = weighted input, z
- Activations = activation function(net input); $a = \sigma(z)$
- Label output = threshold(activations of last layer); $\hat{y} = f(a)$

Special cases:

- In perceptron: activation function = threshold function
- In linear regression: activation = identify function, so net input = output

General Notation for Single-Layer Neural Networks

- Common notation (i.e. in most modern texts) to define the bias unit separately
- However, often inconvenient for mathematical notation



"separate" bias unit

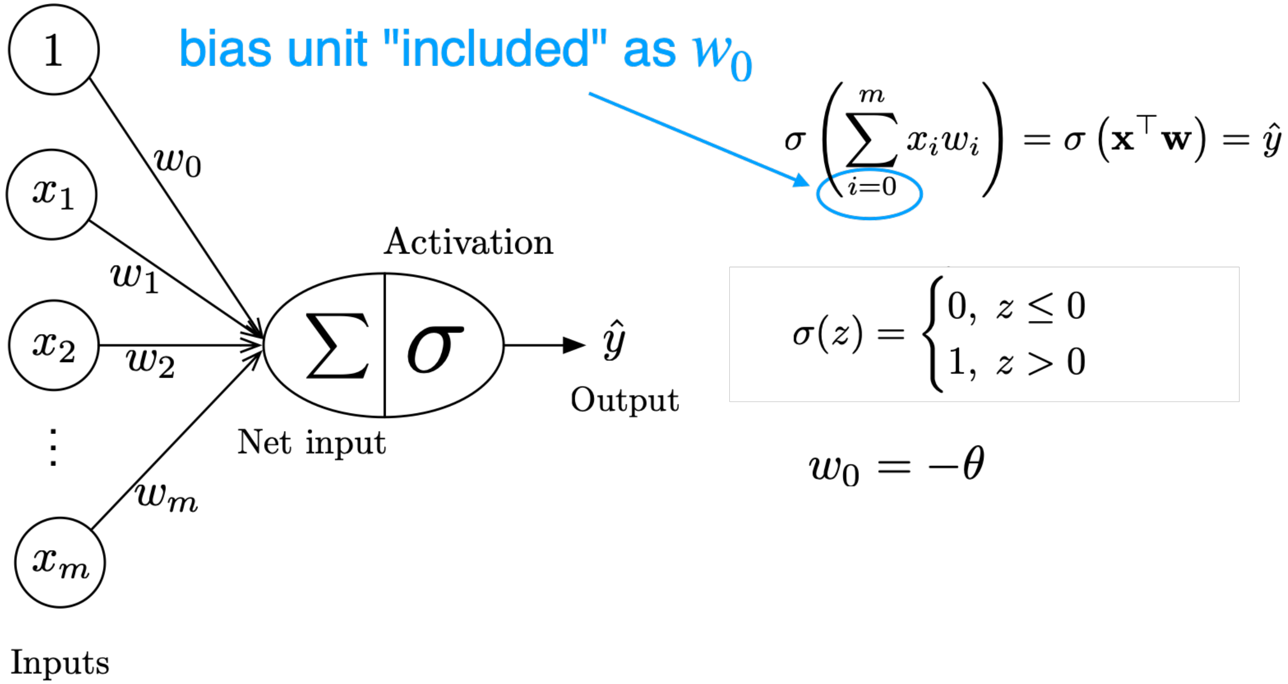
$$\sigma\left(\sum_{i=1}^{m} x_i w_i + b\right) = \sigma\left(\mathbf{x}^T\mathbf{w} + b\right) = \hat{y}$$

$$\sigma(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ if } z \geq 1 \end{cases}$$

$$b = -\theta$$

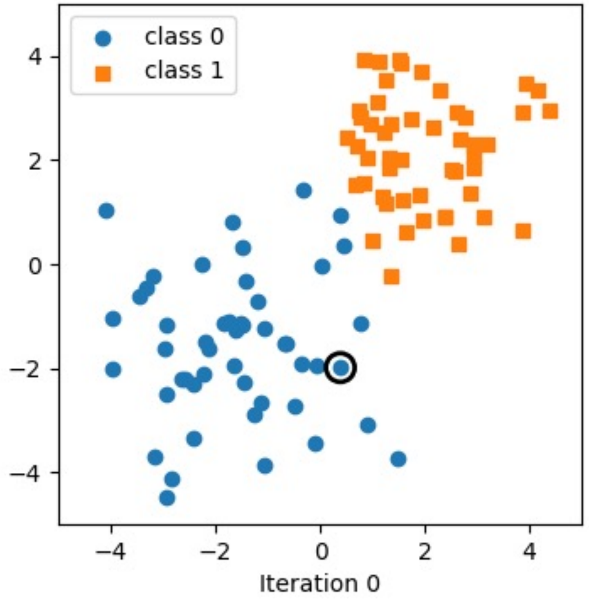
General Notation for Single-Layer Neural Networks

- Often more convenient notation: define bias unit as w_0 and prepend a 1 to each input vector as an additional "feature"
- Modifying input vectors is more inconvenient coding-wise



Perceptron Learning Algorithm

- Assume binary classification task
- Perceptron finds decision boundary is classes are separable



[animated GIF]

Code at https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L03-perceptron/code/perceptron-animation.ipynb

Perceptron Learning Algorithm

- If correct: Do nothing
- If incorrect
  - If output is 0 (target is 1), then add input vector to weight vector
  - If output is 1 (target is 0), then subtract input vector from weight vector

Guaranteed to converge if a solution exists (more about that later…)

Perceptron Learning Algorithm (pseudocode)

Let

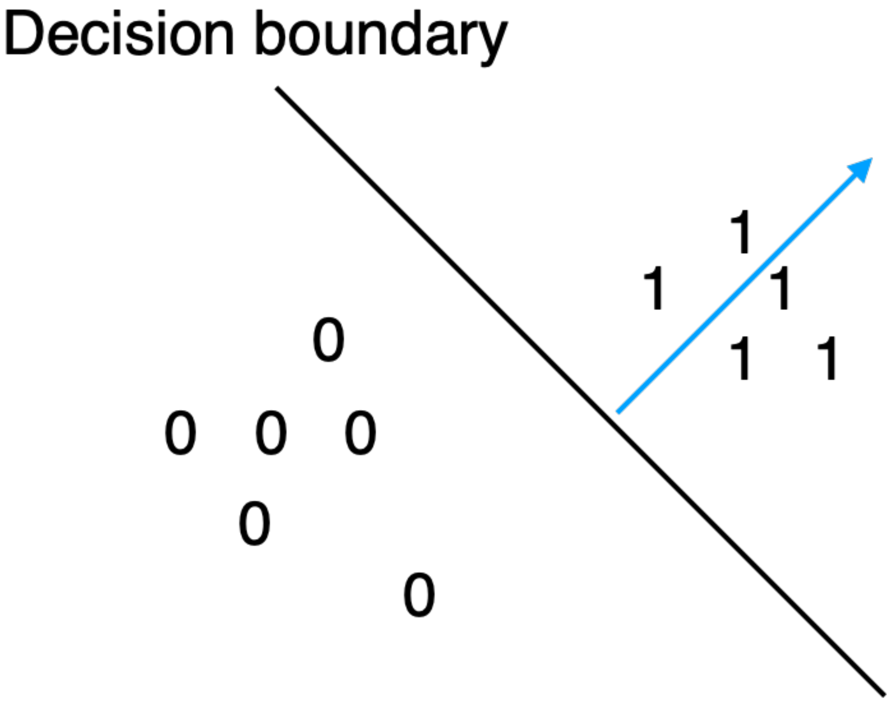
$$\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, \dots, \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$$

1. Initialize $\boldsymbol{w} \coloneqq 0^m$ (assume weight incl. bias)
2. For every training epoch:
   1. For every $\langle \boldsymbol{x}^{[i]}, y^{[i]} \rangle \in D$:
      1. $\hat{y}^{[i]} \coloneqq \sigma(\boldsymbol{x}^{[i]T} \boldsymbol{w})$ ← Only -0 or 1
      2. $err \coloneqq (y^{[i]} - \hat{y}^{[i]})$ ← Only -1, 0, or 1
      3. $\boldsymbol{w} \coloneqq \boldsymbol{w} + err \times \boldsymbol{x}^{[i]}$

Today: Single-layer neural networks

1. Perceptrons
2. **Geometric Intuition**
3. Notational Conventions for Neural Networks
4. A Fully Connected (Linear) Layer in PyTorch

Geometric Intuition



Weight vector is perpendicular to the boundary. Why?

Remember,

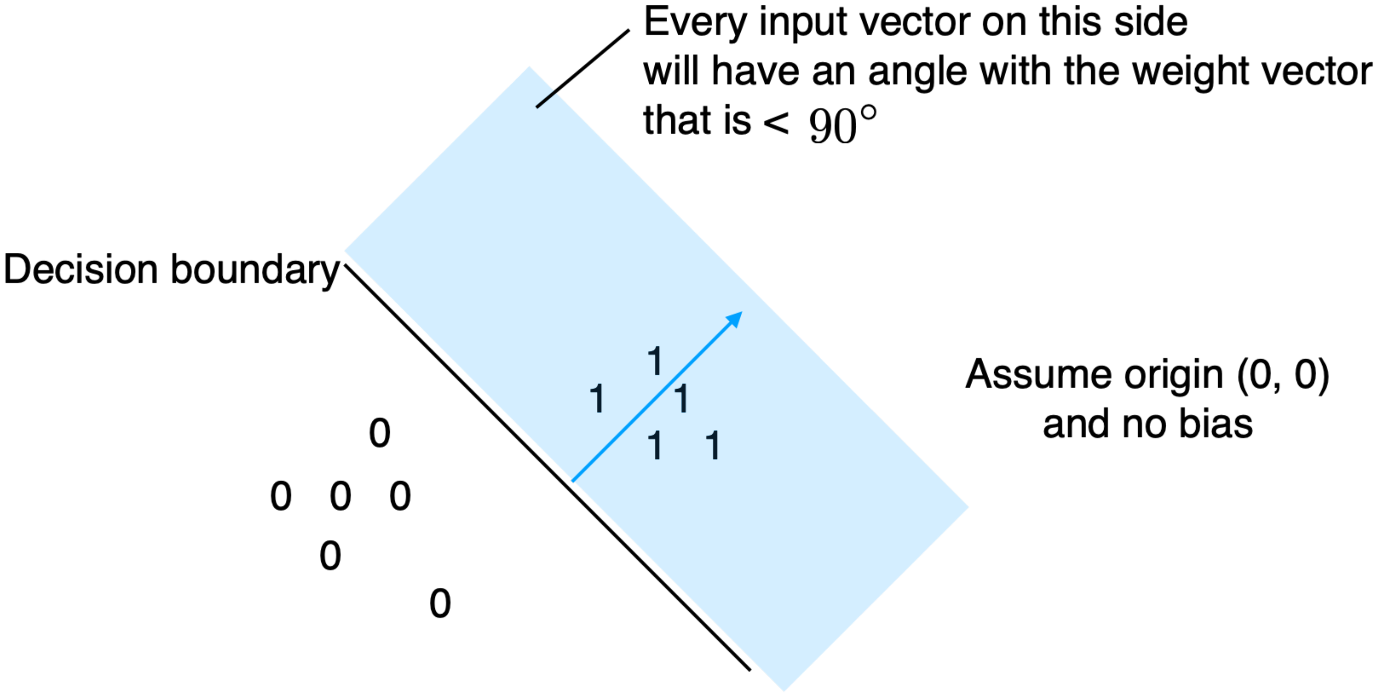
$$\hat{y} = \begin{cases} 0, & \mathbf{w}^T\mathbf{x} \leq 0 \\ 1, & \mathbf{w}^T\mathbf{x} > 0 \end{cases}$$

$$\mathbf{w}^T\mathbf{x} = ||\mathbf{w}|| \cdot ||\mathbf{x}|| \cdot \underbrace{\cos(\theta)}$$

So this needs to be 0 at the boundary, and it is zero at 90°

Geometric Intuition

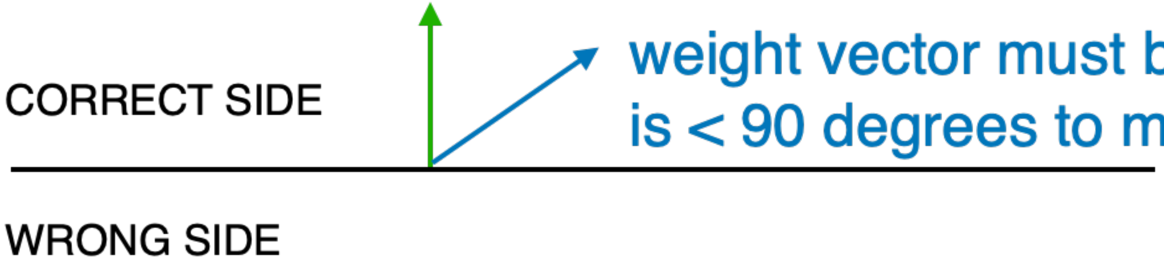
What else does this mean?



So, we could scale the weights and/or inputs by an arbitrary factor and still get the same classification results

Geometric Intuition

input vector for an example with label 1



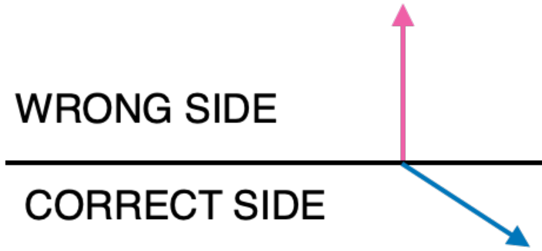
weight vector must be somewhere such that the angle is < 90 degrees to make a correct prediction

The dot product will then be positive, i.e., > 0, since

$$\mathbf{w}^T \mathbf{x} = ||\mathbf{w}|| \cdot ||\mathbf{x}|| \cdot \cos(\theta)$$

Geometric Intuition

input vector for an example with label 0

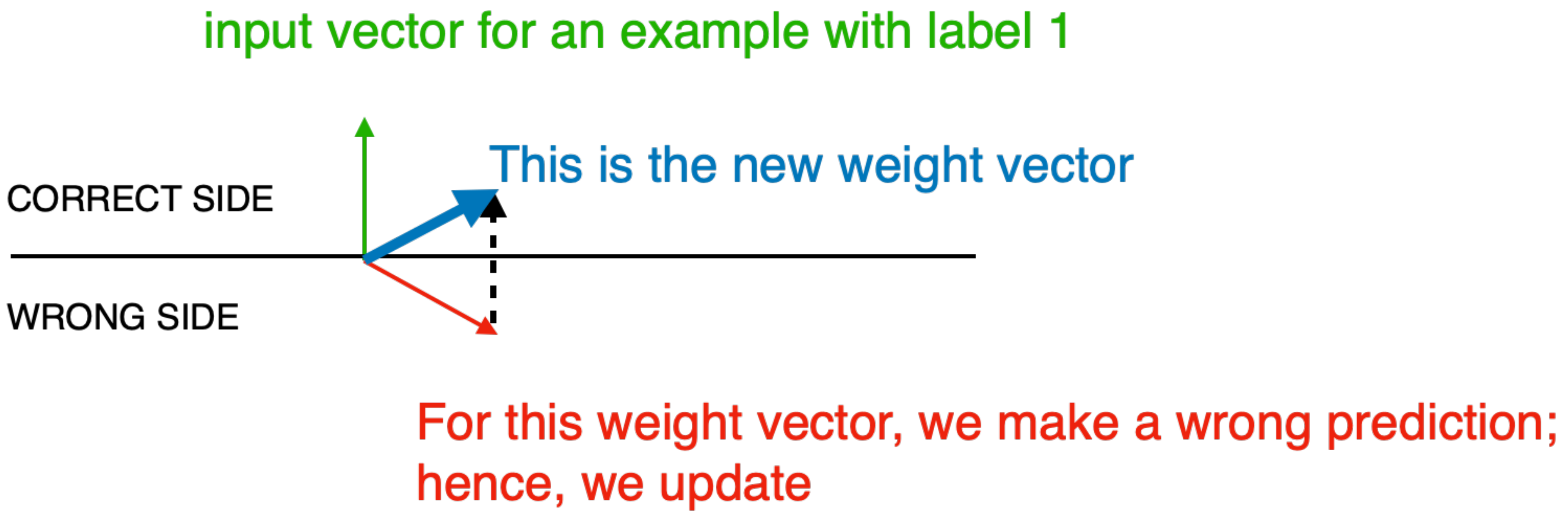


weight vector must be somewhere such that the angle is $\geq$90 degrees to make a correct prediction

The dot product will then $\leq$0, since

$$\mathbf{w}^T\mathbf{x} = ||\mathbf{w}|| \cdot ||\mathbf{x}|| \cdot \cos(\theta)$$

Geometric Intuition: An update



Perceptron Limitations

- The (classic) Perceptron has many problems
  - Linear classifier, no non-linear boundaries
  - Binary classifier, cannot solve XOR problems
  - Does not converge if classes are not linearly separable
  - Many “optimal” solutions in terms of 0/1 loss on the training data
    - Most will not be optimal in terms of generalization performance

Perceptron Fun Fact

Where a perceptron had been trained to distinguish between - this was for military purposes - it was looking at a scene of a forest in which there were camouflaged tanks in one picture and no camouflaged tanks in the other. And the perceptron - after a little training - made a 100% correct distinction between these two different sets of photographs. Then they were embarrassed a few hours later to discover that the two rolls of film had been developed differently. And so these pictures were just a little darker than all of these pictures and the perceptron was just measuring the total amount of light in the scene. But it was very clever of the perceptron to find some way of making the distinction.

-- Marvin Minsky, Famous AI researcher, Author of the famous "Perceptrons" book

Source: https://www.webofstories.com/play/marvin.minsky/122

https://qph.fs.quoracdn.net/main-qimg-305eb8136c4a20f348bb7ab465bc2e10

http://theconversation.com/want-to-beat-climate-change-protect-our-natural-forests-121491

Perceptrons and Distributions

- Is the classic Perceptron learning algorithm a form of MLE/MAP estimation? If so, what's the interpretation? If not, why not?

Perceptrons and Distributions

- Is the classic Perceptron learning algorithm a form of MLE/MAP estimation? If so, what's the interpretation? If not, why not?
  - No
  - Classic Perceptron defines a hard threshold with a deterministic mapping:
    $$\sigma(z) = step(z) = step(\boldsymbol{w}^T \boldsymbol{x})$$
    - No likelihood defined → No MLE/MAP estimation

- What if we use a sigmoid activation $\sigma(z) = \frac{1}{1+e^{-z}}$?

Perceptrons and Distributions

- Is the classic Perceptron learning algorithm a form of MLE/MAP estimation? If so, what's the interpretation? If not, why not?
  - No
  - Classic Perceptron defines a hard threshold with a deterministic mapping:
    $$\sigma(z) = step(z) = step(\boldsymbol{w}^T \boldsymbol{x})$$
    - No likelihood defined → No MLE/MAP estimation

- What if we use a sigmoid activation $\sigma(z) = \frac{1}{1+e^{-z}}$?
  - Yes
  - This is logistic regression!
  - Likelihood function: $L(w) = \prod_i \sigma(w^T x)^{y_i}(1 - \sigma(w^T x))^{1-y_i}$
    can be maximized for MLE → gradient ascent

Parallel Histories

- **1838** – Verhulst: introduces the *logistic function* (population growth).
- **1943** – McCulloch & Pitts: logic neurons (no learning).
- **1944** – Berkson: introduces the logit.
- **1957** – Rosenblatt: perceptron + learning rule (connectionism).
- **1958** – Cox: logistic regression as general regression (statistics).
- **1969** – Minsky & Papert: perceptron limits (can't do XOR).
- **1986** – Rumelhart, Hinton & Williams: backpropagation + sigmoids.
  - Can be seen as "solve the 1969 problem by stacking the 1958 model → MLE by gradient ascent/chain rule"

Perceptrons and DL

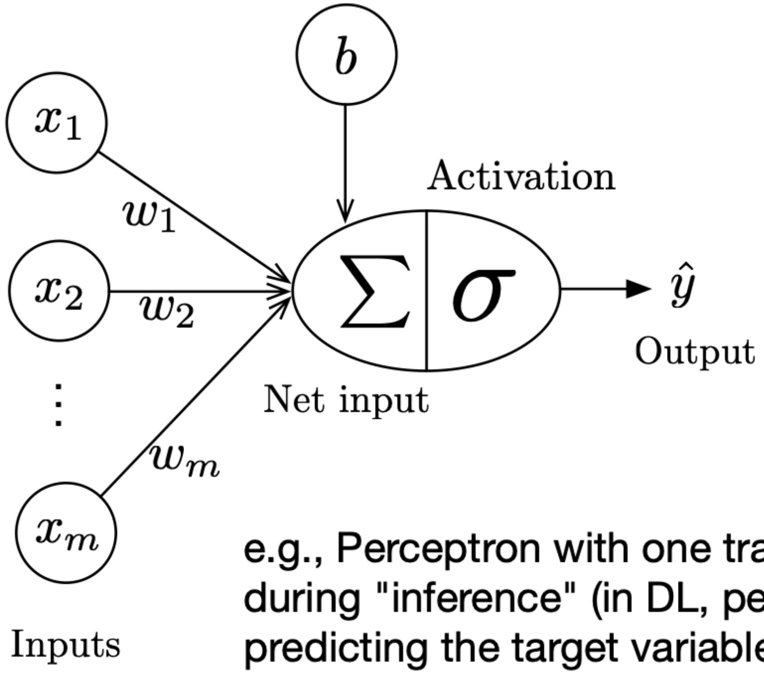
- So why is Rosenblatt's Perceptron considered the basis of DL (instead of logistic regression)?
  - First "Neuron that learns"
    - Rosenblatt framed it explicitly as a biologically-inspired neuron
    - Learning rule + Hardware Prototypes ("Mark 1 Perceptron")
  - Cultural lineage
  - Narrative power
  - Architectural continuity ("logistic" over-specified)
- Maybe if statisticians had won the naming war, we'd be talking about multi-layer logistic models.

Today: Single-layer neural networks

1. Perceptrons
2. Geometric Intuition
3. **Notational Conventions for Neural Networks**
4. A Fully Connected (Linear) Layer in PyTorch

So far...

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

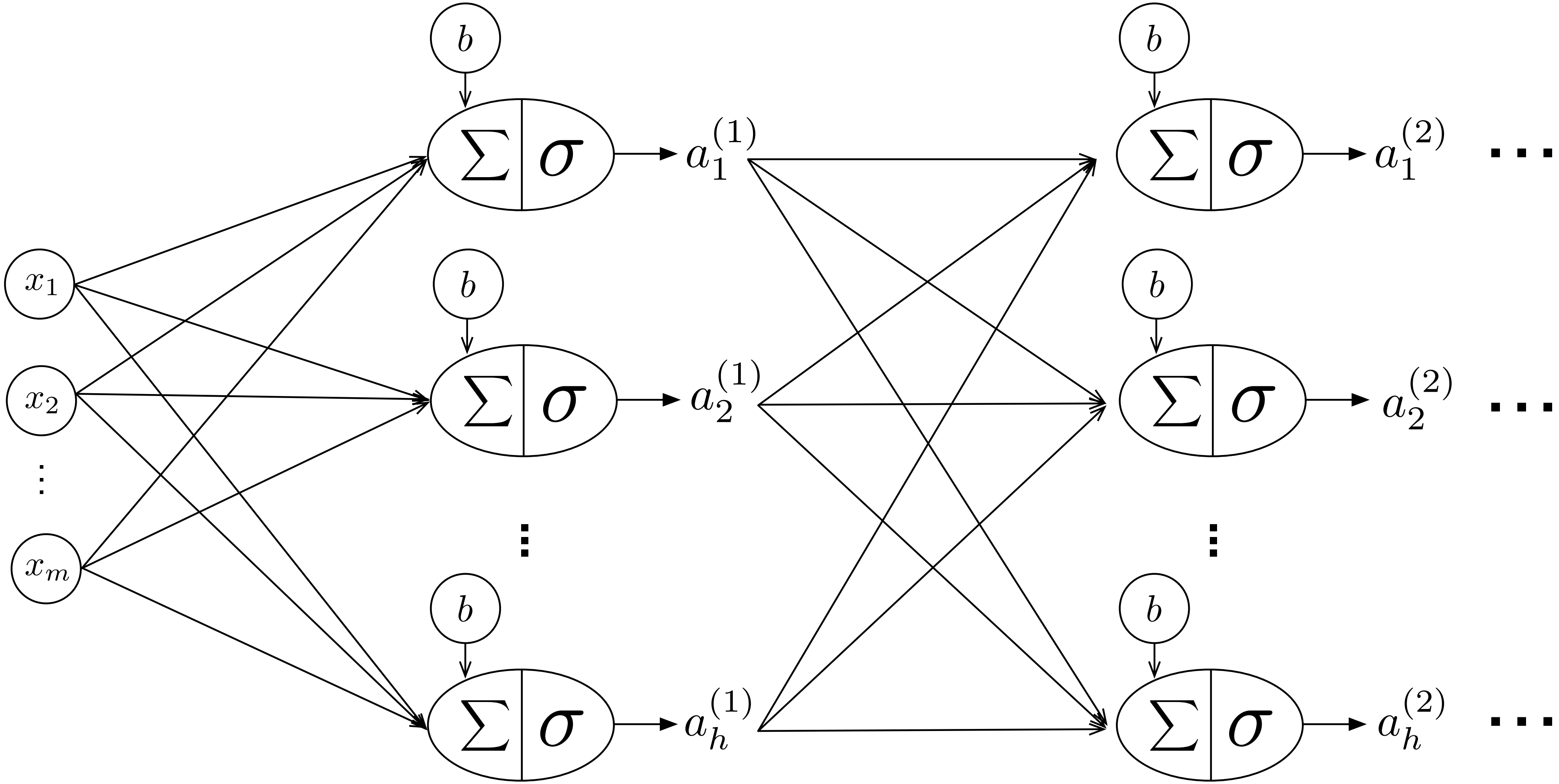


e.g., Perceptron with one training example as input during "inference" (in DL, people now often refer to predicting the target variable as "inference")

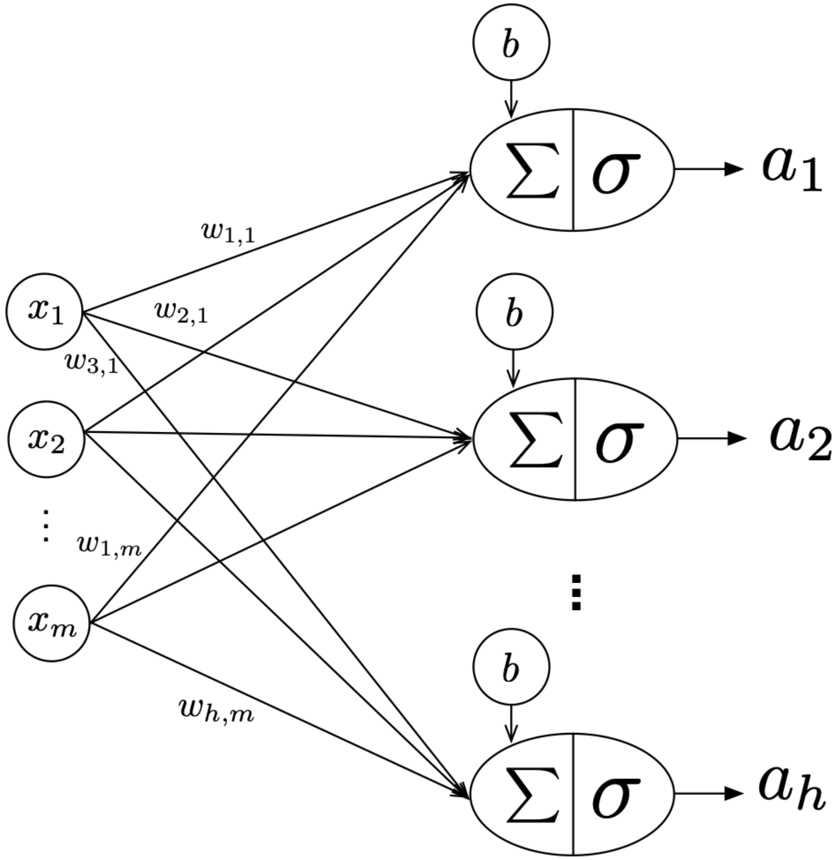
If we have *n* training examples, $\mathbf{X} \in \mathbb{R}^{n \times m}, \quad \mathbf{z} \in \mathbb{R}^{n \times 1}$

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

Soon...



A Fully-Connected Layer



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

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

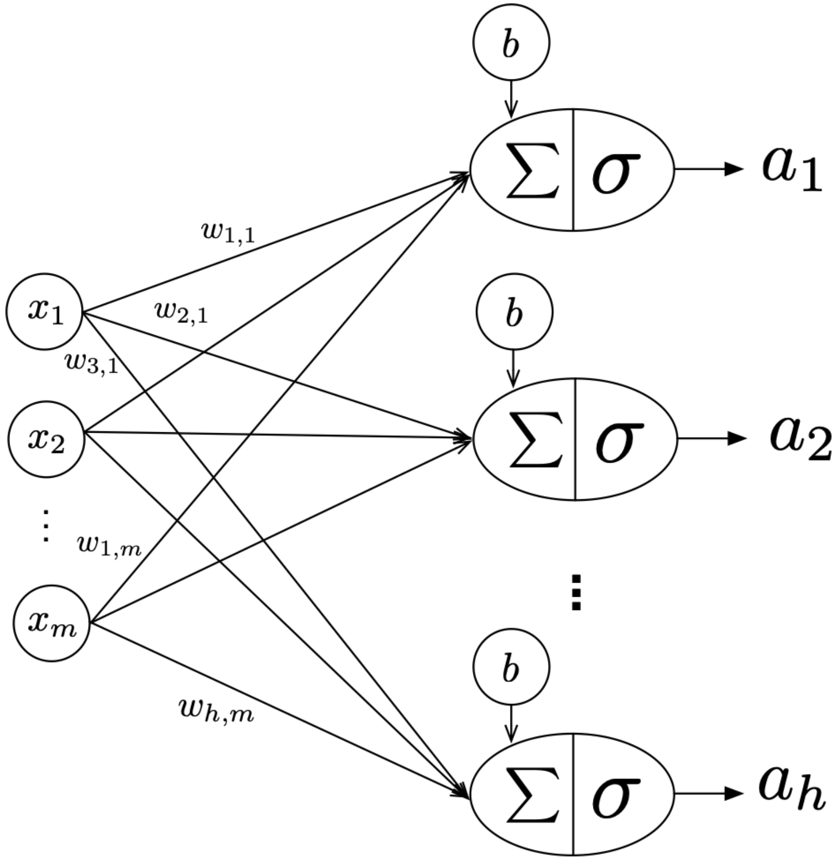
Layer activations for 1 training example

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

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

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

A Fully-Connected Layer



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

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

Layer activations for *n* training examples

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

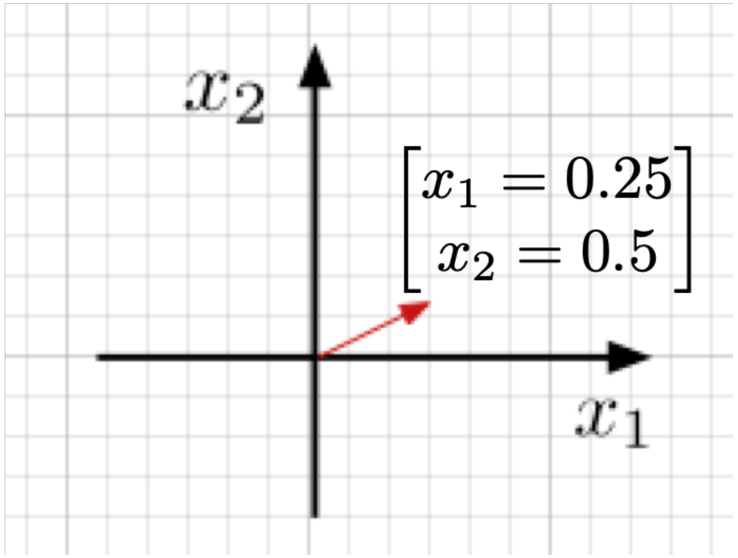
$$\mathbf{A} \in \mathbb{R}^{n \times h}$$

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

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



Why is the Wx notation intuitive?

scales the x coordinate

moves y into x direction

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

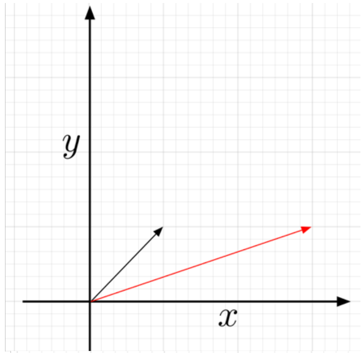
moves x in y direction

scales the y coordinate

Why is the Wx notation intuitive?

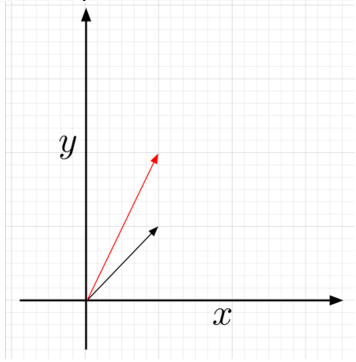
Stretching x-axis by factor of 3:

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



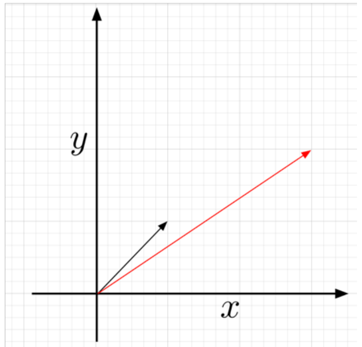
Stretching y-axis by factor of 2:

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



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

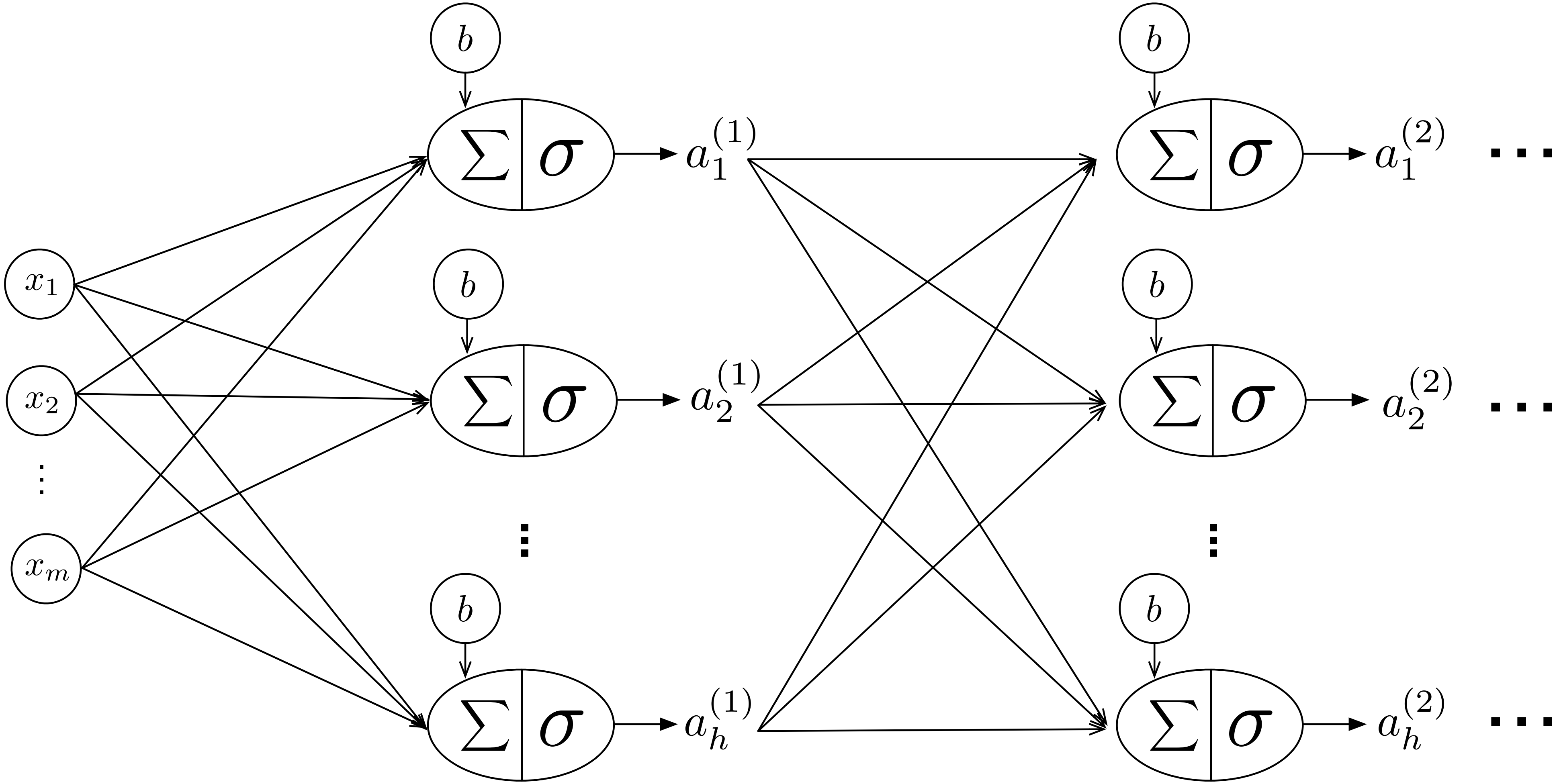
$$\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ 2y \end{bmatrix}$$



Today: Single-layer neural networks

1. Perceptrons
2. Geometric Intuition
3. Notational Conventions for Neural Networks
4. **A Fully Connected (Linear) Layer in PyTorch**

A Fully-Connected Layer



A 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)
```

```
[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])
```

About notation

- ML culture is mix of implementation and model development
  - These do not always suggest the same notations
- 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:

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

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

Be cautious of this when debugging…very prone to mistakes with shapes…

Next time

- A better learning algorithm for neural networks

Questions?

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON