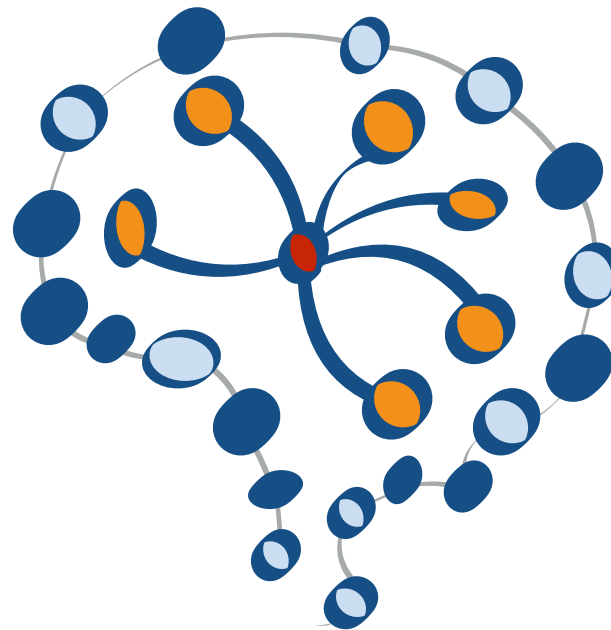


# STAT 453: Introduction to Deep Learning and Generative Models

Sebastian Raschka

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



## Lecture 03

# The Perceptron

## An Introduction to Single Layer Neural Networks

# Announcements

- Project groups (by next Thu), 3 members per group -- TA will set up a document where you can add your team member preferences
- Project topics (brainstorm with group members)
- HW1 (related to the Perceptron; more about that later)
- Piazza for questions, encouraged to help each other (but don't share your HW solutions)

After this lecture, you will be able to  
implement your first neuron model for  
making predictions!

# Lecture Overview

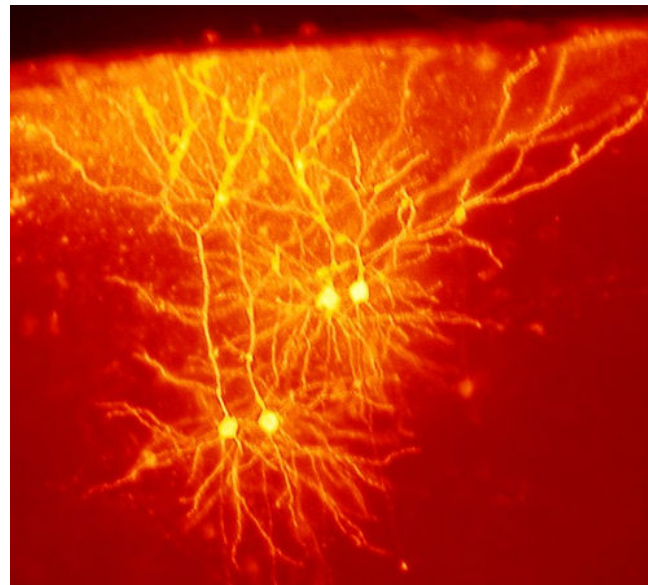
- 1. Brains and neuron models**
2. The perceptron learning rule
3. Interlude: "vectorization" in Python
4. Implementing a perceptron in Python using NumPy and PyTorch
5. Optional: The perceptron convergence theorem
6. Geometric intuition



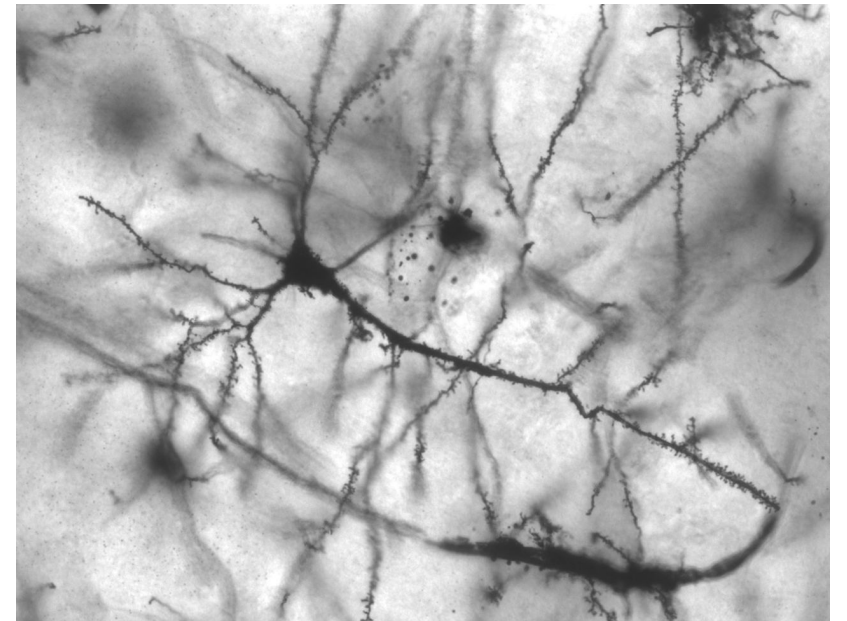
# Inspired by Biological Brains and Neurons



<https://publicdomainpictures.net/en/view-image.php?image=130359&picture=human-brain>



[https://commons.wikimedia.org/wiki/Neuron#/media/File:Mouse\\_cingulate\\_cortex\\_neurons.jpg](https://commons.wikimedia.org/wiki/Neuron#/media/File:Mouse_cingulate_cortex_neurons.jpg)



[https://commons.wikimedia.org/wiki/Neuron#/media/File:Pyramidal\\_hippocampal\\_neuron\\_40x.jpg](https://commons.wikimedia.org/wiki/Neuron#/media/File:Pyramidal_hippocampal_neuron_40x.jpg)

## Do our brains use deep learning?



source: [https://media.mnn.com/assets/images/2016/10/plane-birds.jpg.1000x0\\_q80\\_crop-smart.jpg](https://media.mnn.com/assets/images/2016/10/plane-birds.jpg.1000x0_q80_crop-smart.jpg)

# Number of neurons in brains ...



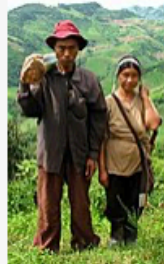




Article [Talk](#)

## List of animals by number of neurons

From Wikipedia, the free encyclopedia

sixteen billion three hundred forty million

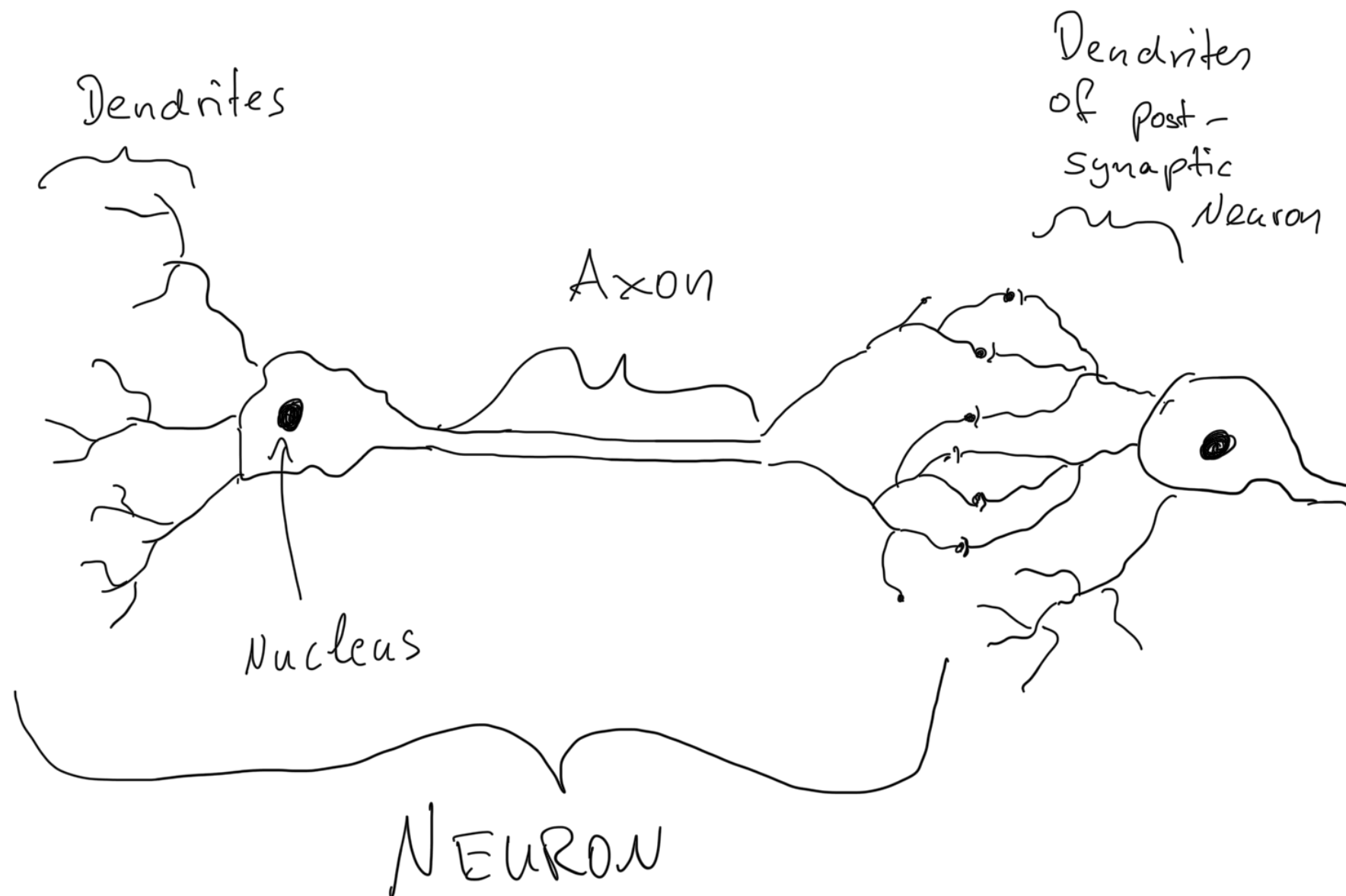
On a sidenote:

<b>Human</b>	16,340,000,000 21,000,000,000*	± 2,170,000,000 <sup>[38]</sup>	Isotropic fractionator Optical fractionator	Pallium (cortex)	<i>Homo sapiens</i>	
<b>Risso's dolphin</b>	18,750,000,000^		Estimated	Pallium (cortex)	<i>Grampus griseus</i>	
<b>Short-finned pilot whale</b>	35,000,000,000^		Estimated	Pallium (cortex)	<i>Globicephala macrorhynchus</i>	
<b>Long-finned pilot whale</b>	37,200,000,000*		Optical fractionator	Pallium (cortex)	<i>Globicephala melas</i>	
<b>Killer whale</b>	43,100,000,000*		Optical fractionator	Pallium (cortex)	<i>Orcinus orca</i>	

Name	Short scale (US, Eastern Europe, English Canadian, and modern British)	Long scale (Western, Central Europe, older British, and French Canadian)
Million	10 <sup>6</sup>	10 <sup>6</sup>
Milliard		10 <sup>9</sup>
Billion	10 <sup>9</sup>	10 <sup>12</sup>
Billiard		10 <sup>15</sup>
Trillion	10 <sup>12</sup>	10 <sup>18</sup>

Source: [https://en.wikipedia.org/wiki/List\\_of\\_animals\\_by\\_number\\_of\\_neurons](https://en.wikipedia.org/wiki/List_of_animals_by_number_of_neurons)

# A Biological Neuron

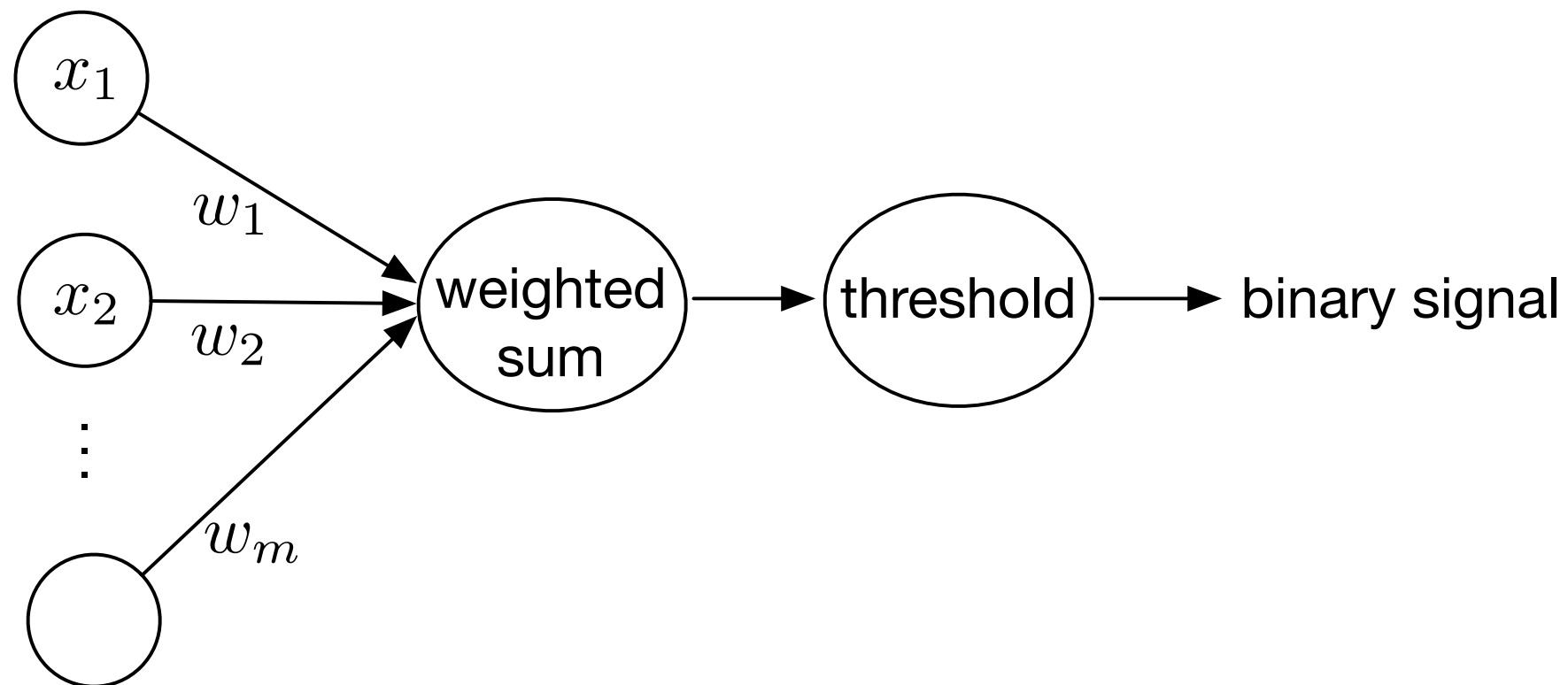




# McCulloch & Pitts Neuron Model

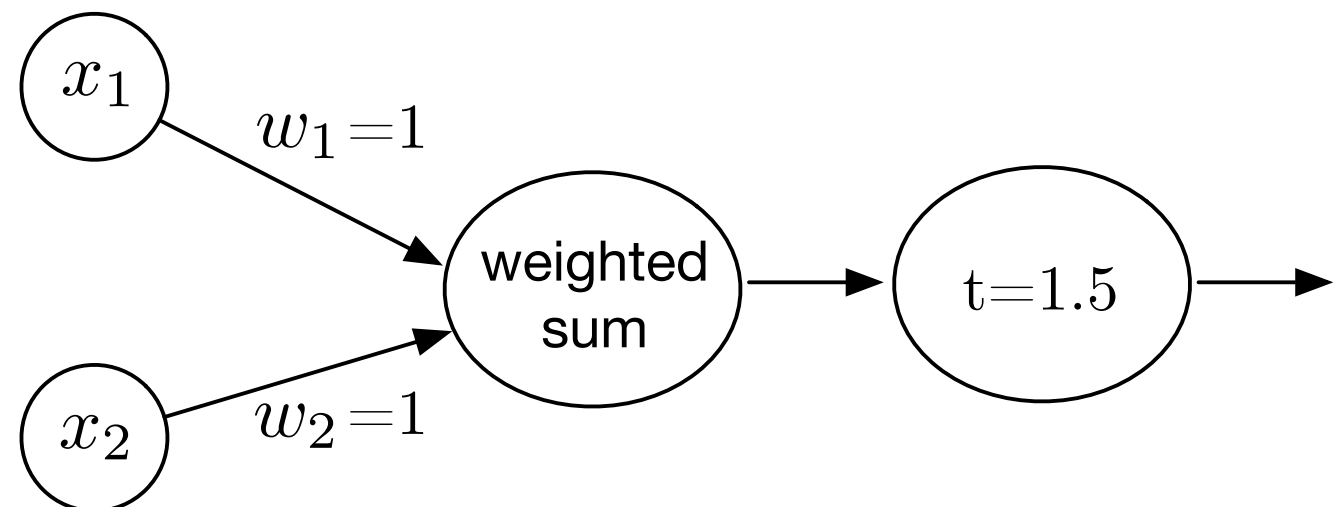
## A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY

WARREN S. MCCULLOCH and WALTER H. PITTS 1943



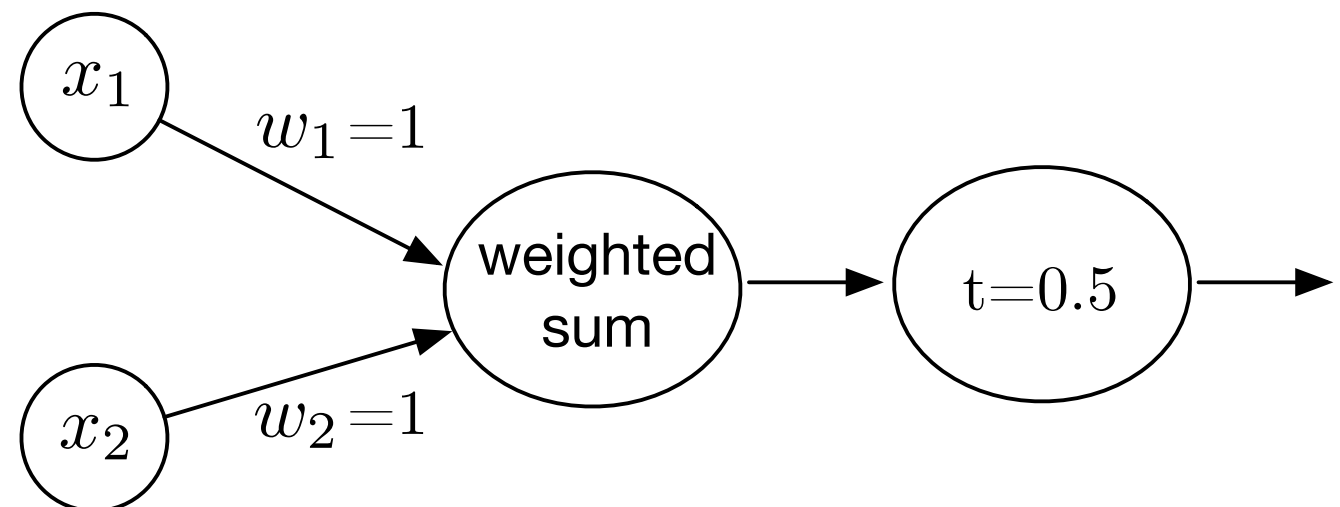
# Logical AND Gate

$x_1$	$x_2$	$Out$
0	0	0
0	1	0
1	0	0
1	1	1



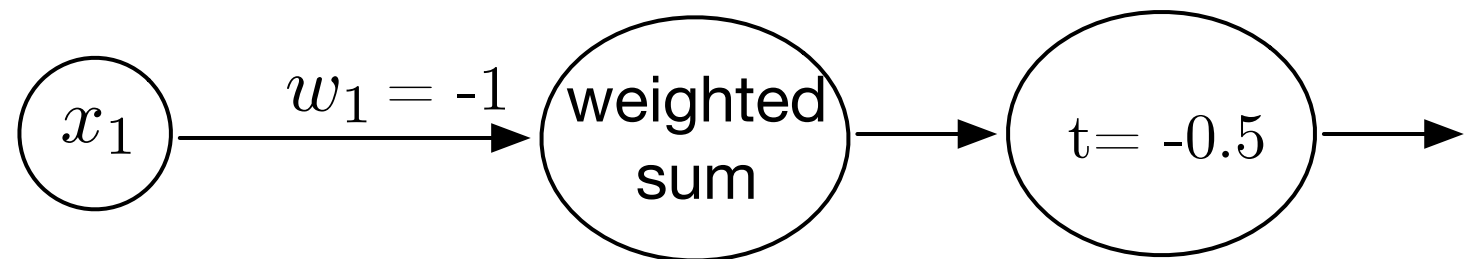
# Logical OR Gate

$x_1$	$x_2$	$Out$
0	0	0
0	1	1
1	0	1
1	1	1



# Logical NOT Gate

$x_1$	$Out$
0	1
1	0

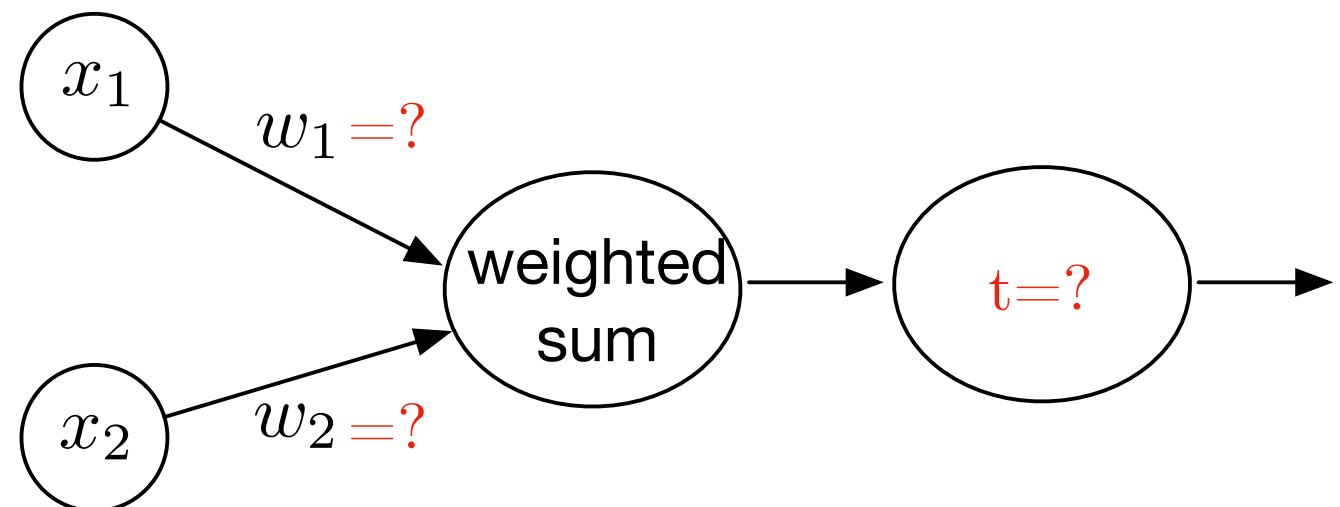




# Logical XOR Gate

(Take-home exercise)

$x_1$	$x_2$	$Out$
0	0	0
0	1	1
1	0	1
1	1	0



# Training Single-Layer Neural Networks

1. Brains and neuron models
- 2. The perceptron learning rule**
3. Interlude: "vectorization" in Python
4. Implementing a perceptron in Python using NumPy and PyTorch
5. Optional: The perceptron convergence theorem
6. Geometric intuition

# 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.



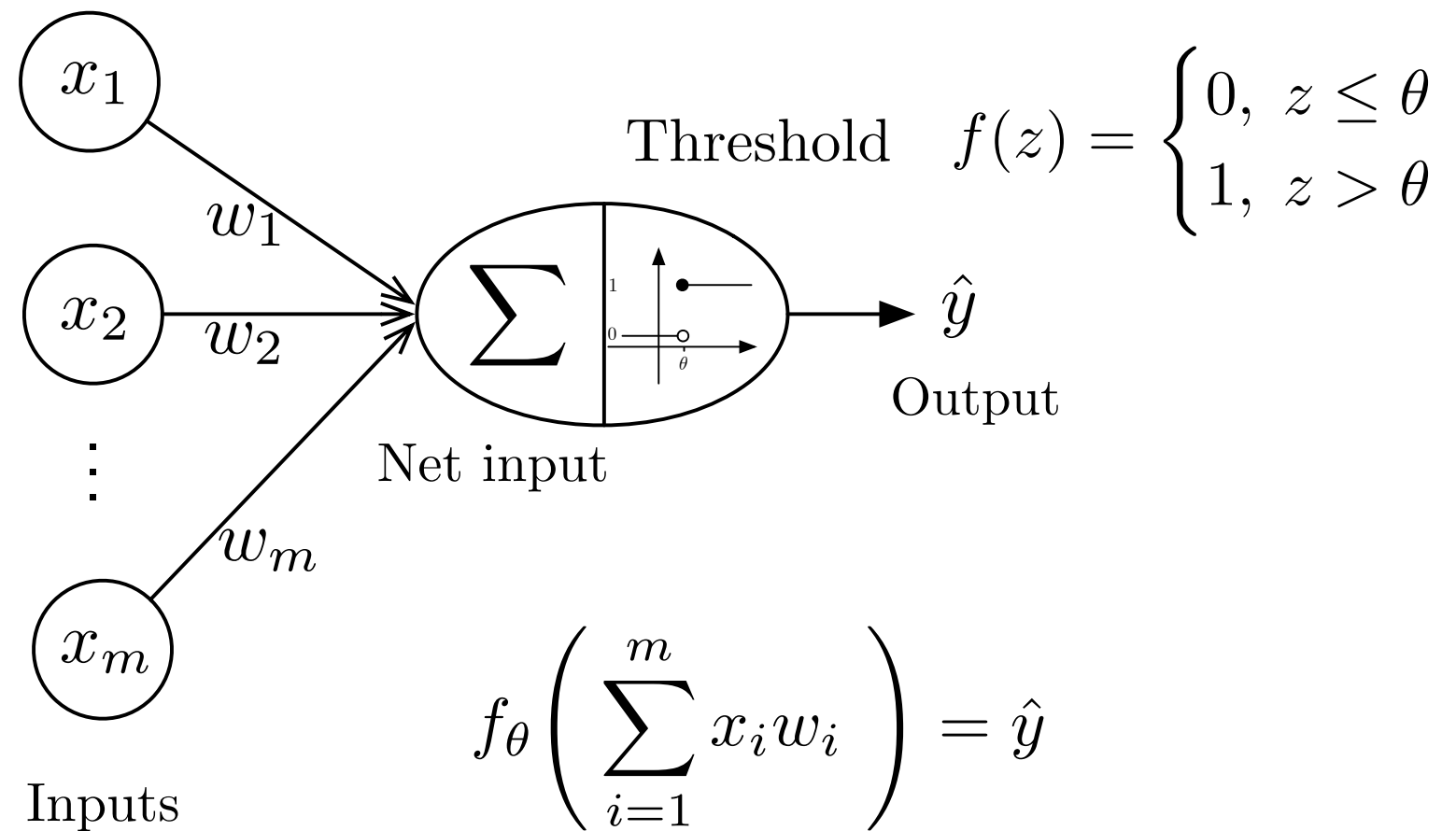
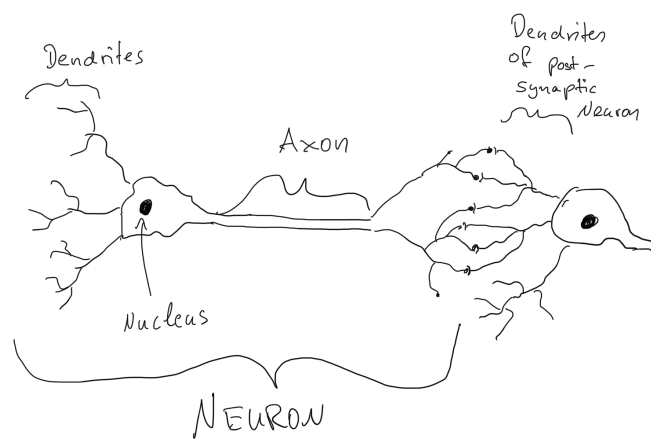
Source: <http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/Members/wilex4/Rosen-2.jpg>

# Perceptron Variants

Note that Rosenblatt (and later others) proposed many variants of the Perceptron model and learning rule. We discuss a "basic" version; let's say,

"Perceptron" := "a classic Rosenblatt Perceptron"

# A Computational Model of a Biological Neuron



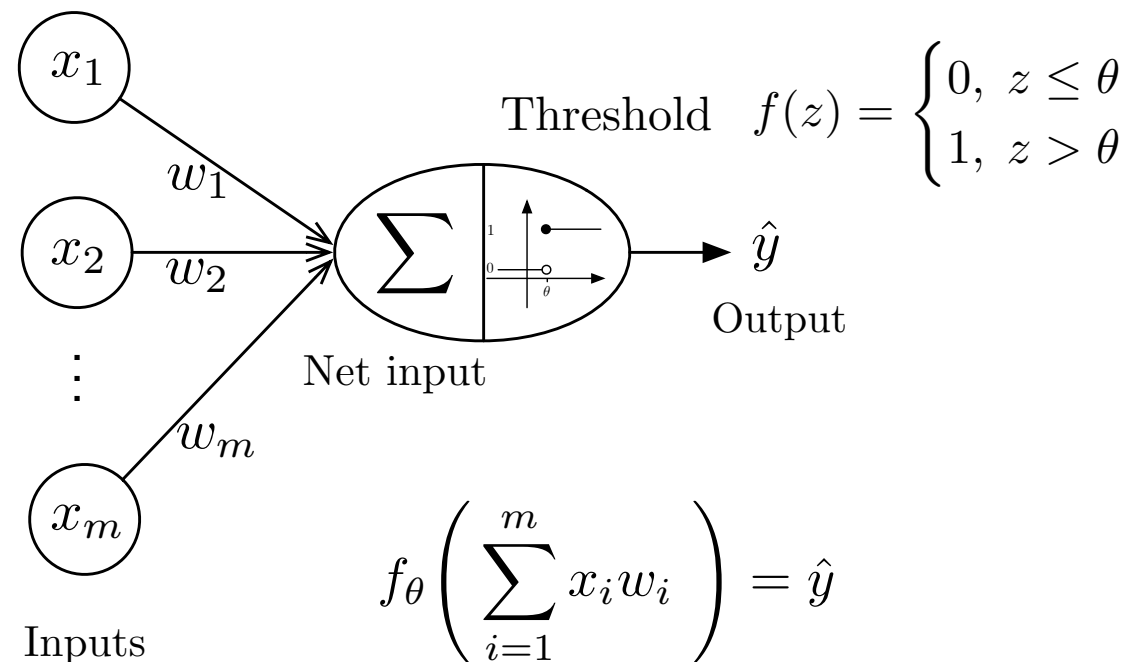
# Terminology

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

- Net input = weighted inputs,  $z$
- 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 = net input = output



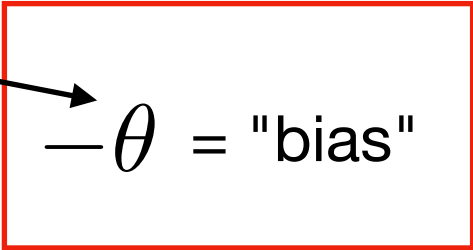
# Perceptron Output

$$\hat{y} = \begin{cases} 0, & z \leq \theta \\ 1, & z > \theta \end{cases}$$

More convenient to re-arrange:

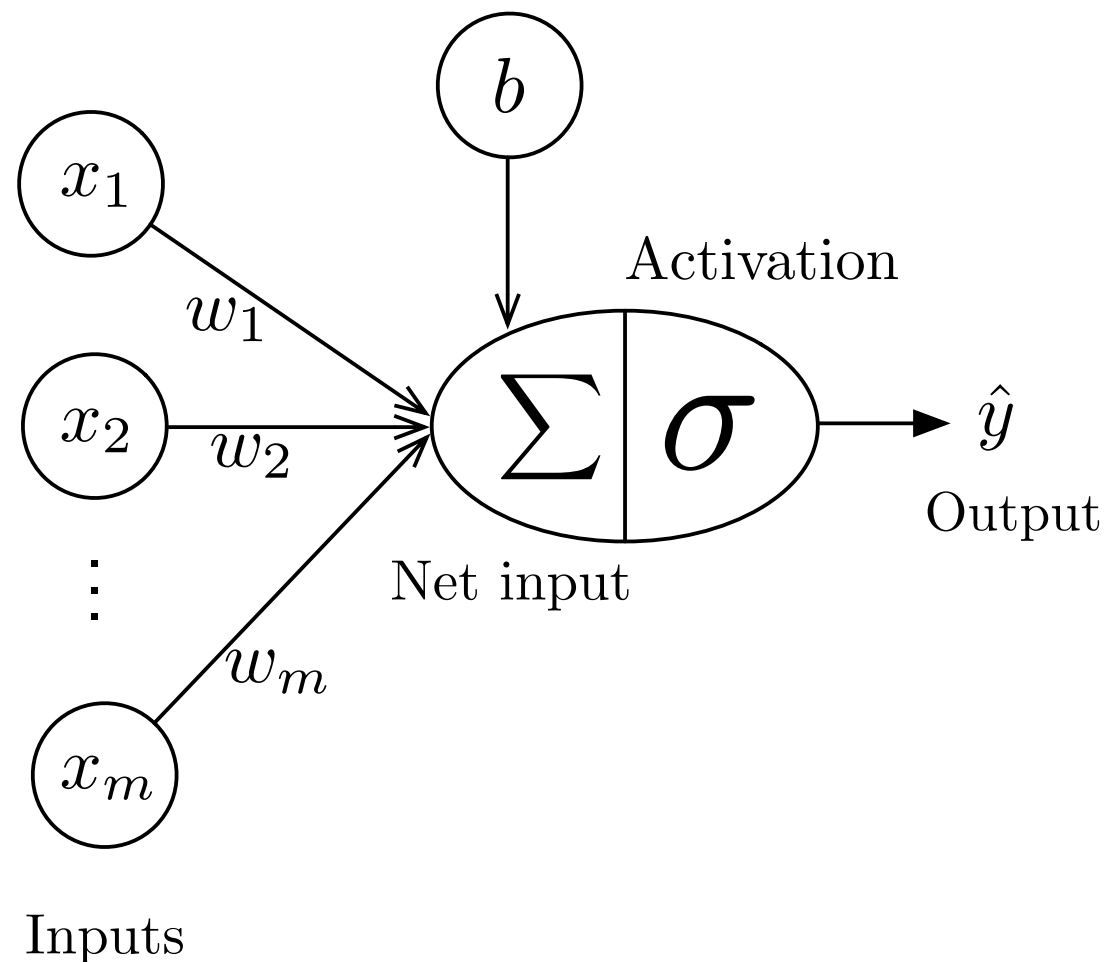
$$\hat{y} = \begin{cases} 0, & z - \theta \leq 0 \\ 1, & z - \theta > 0 \end{cases}$$

negative threshold


$$-\theta = \text{"bias"}$$

# General Notation for Single-Layer Neural Networks

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



"separate" bias unit

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

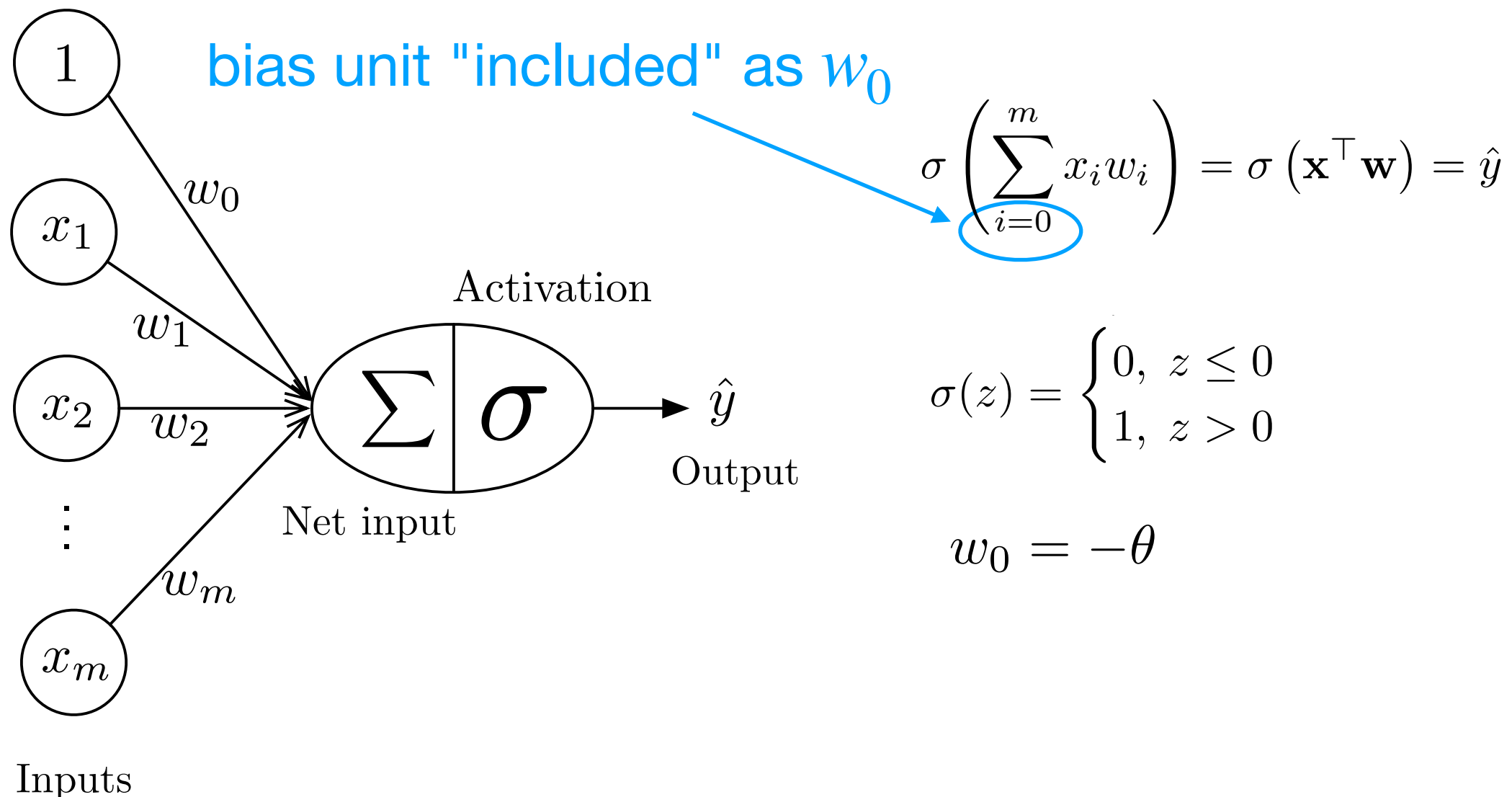
$$\sigma(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$$

$$b = -\theta$$

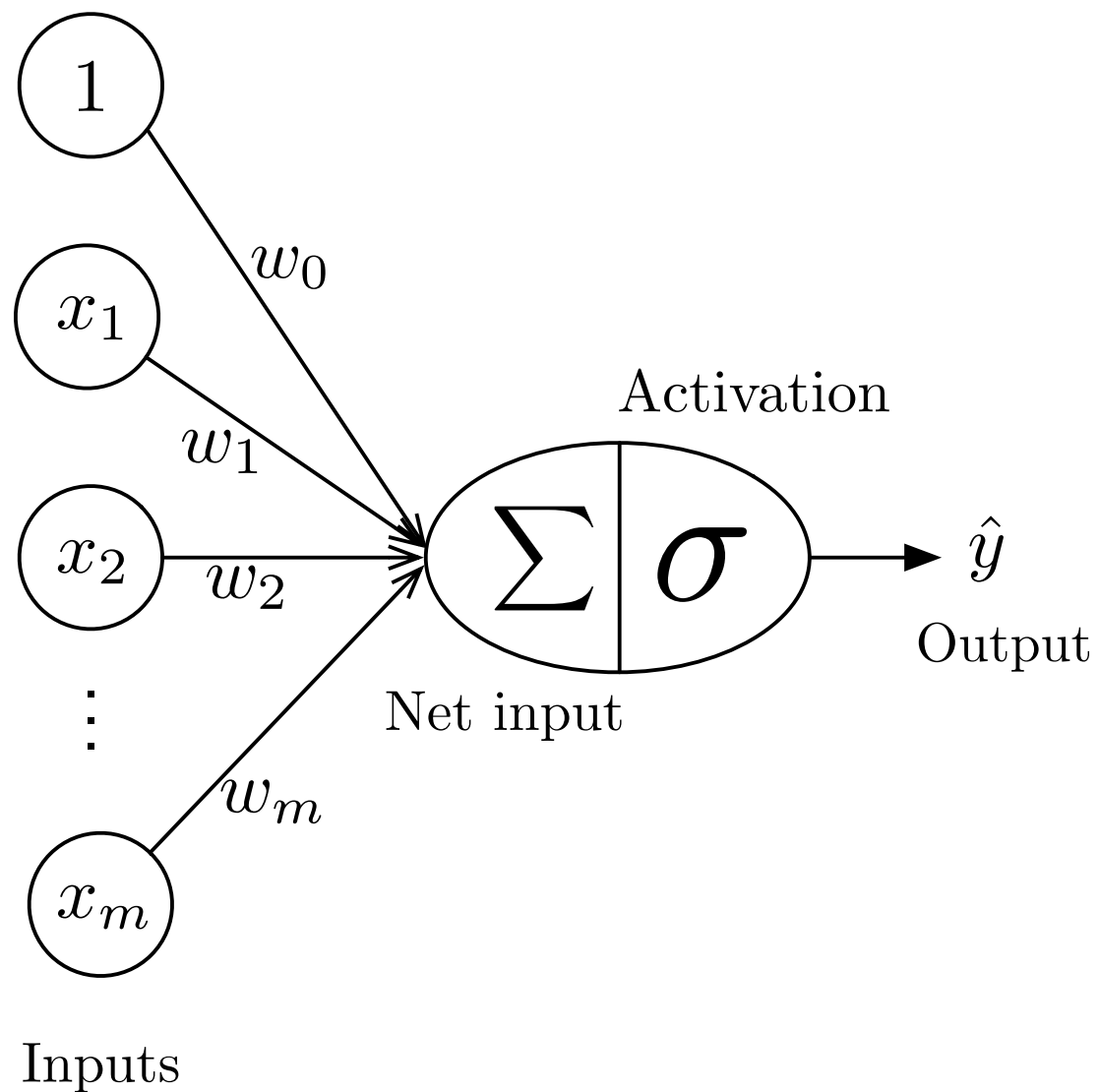


# General Notation for Single-Layer Neural Networks

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



# General Notation for Single-Layer Neural Networks



Vector dot product

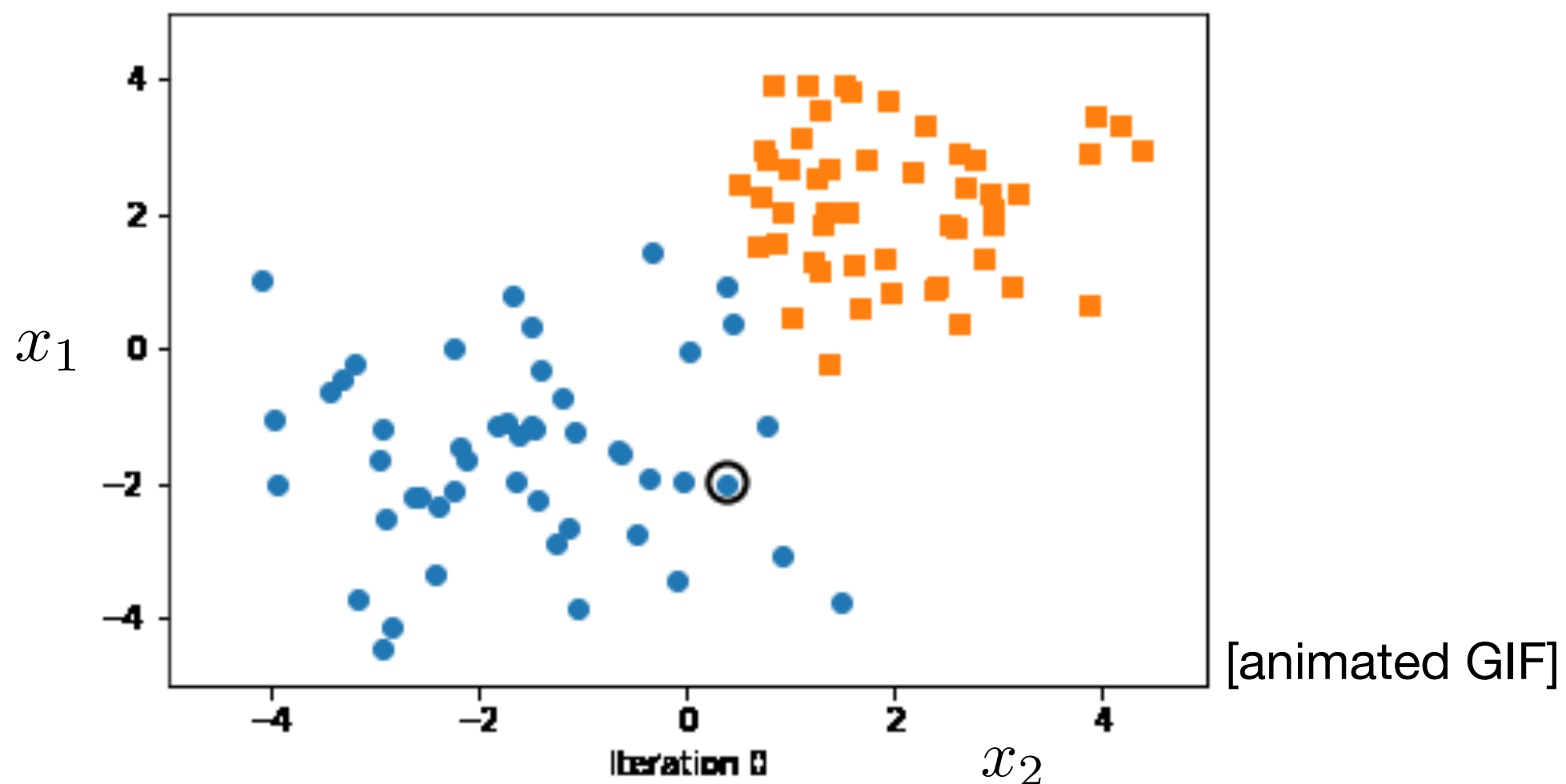
$$\sigma \left( \sum_{i=0}^m x_i w_i \right) = \sigma(\mathbf{x}^T \mathbf{w}) = \hat{y}$$

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

$$w_0 = -\theta$$

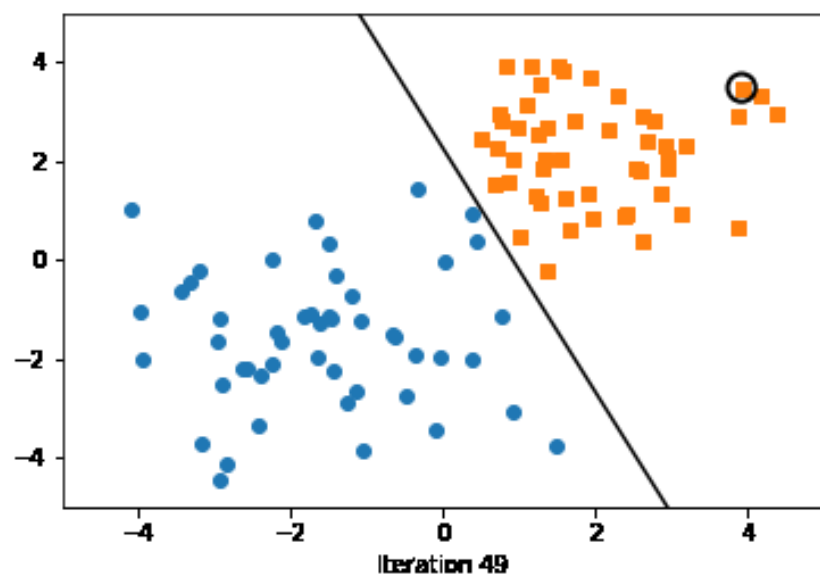
# Perceptron Learning Rule

Assume binary classification task, Perceptron finds decision boundary if classes are separable



# The Perceptron Learning Algorithm

- If correct: Do nothing if the prediction if output is equal to the target
- If incorrect, scenario **a)**:  
If output is 0 and target is 1, add input vector to weight vector
- If incorrect, scenario **b)**:  
If output is 1 and target is 0, subtract input vector from weight vector



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

# The Perceptron Learning Algorithm

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  $\mathbf{w} := \mathbf{0}^m$  (assume notation where weight incl. bias)
2. For every training epoch:
  - A. For every  $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
    - (a)  $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]\top} \mathbf{w})$
    - (b)  $err := (y^{[i]} - \hat{y}^{[i]})$
    - (c)  $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$

# Efficient Scientific Computing: Vectorization in Python

1. Brains and neuron models
2. The perceptron learning rule
- 3. Interlude: "vectorization" in Python**
4. Implementing a perceptron in Python using NumPy and PyTorch
5. Optional: The perceptron convergence theorem
6. Geometric intuition

# Running Computations is a Big Part of Deep Learning!

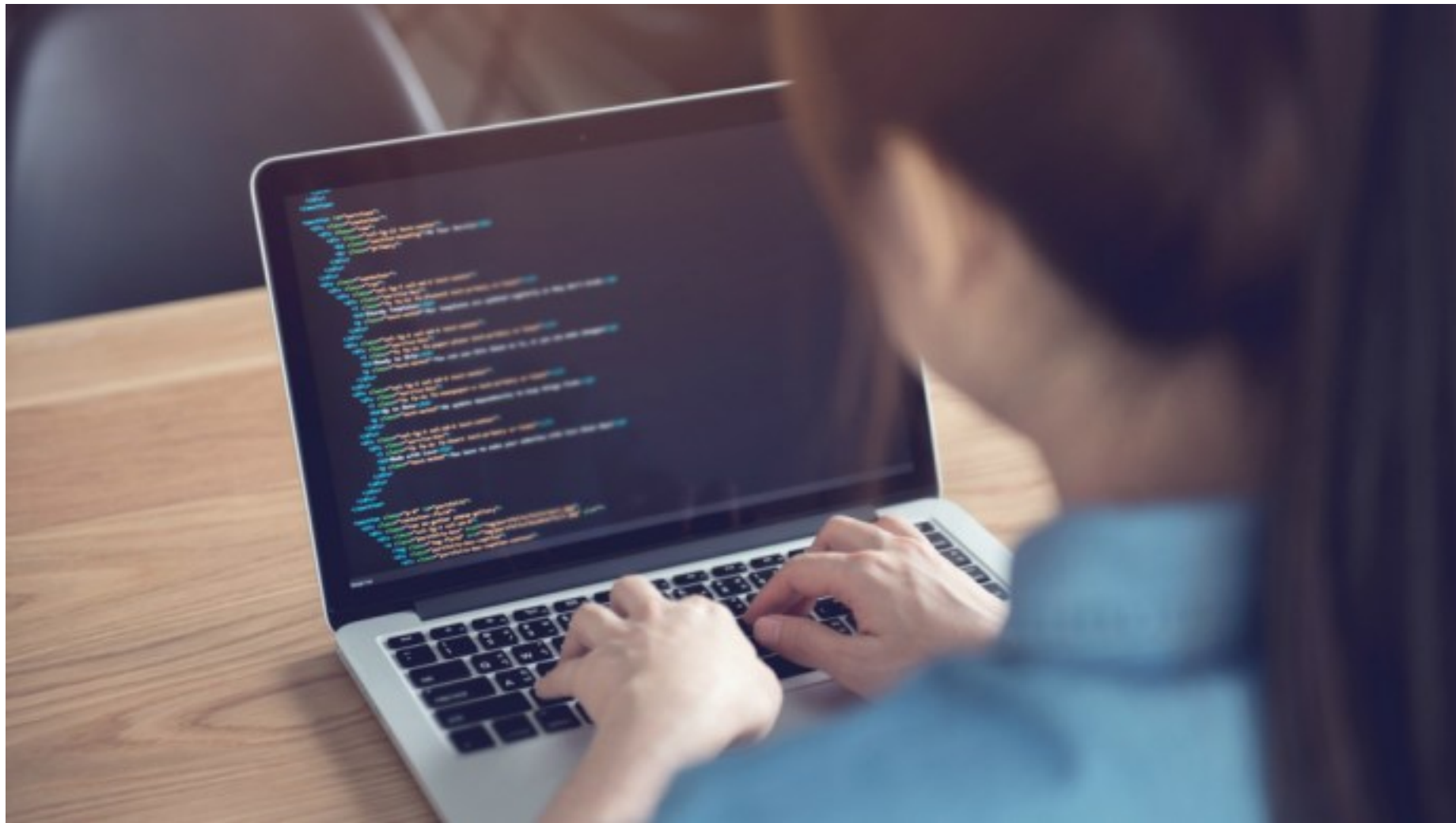


Image source: <https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcRYIMXrK6UIC4IBvLKW4qfHgMUCbQLQ-vnvqA&usqp=C>

# Interlude: "Vectorization" in Python

Question for you: What are we computing here?

In [1]:

```
x0, x1, x2 = 1., 2., 3.  
bias, w1, w2 = 0.1, 0.3, 0.5  
  
x = [x0, x1, x2]  
w = [bias, w1, w2]
```

A simple for-loop:

In [2]:

```
z = 0.  
for i in range(len(x)):  
    z += x[i] * w[i]  
  
print(z)
```

2.2



# Interlude: "Vectorization" in Python

A simple for-loop:

In [2]:

```
z = 0.  
for i in range(len(x)):  
    z += x[i] * w[i]  
  
print(z)
```

2.2

A little bit better, list comprehensions:

In [3]:

```
z = sum(x_i*w_i for x_i, w_i in zip(x, w))  
print(z)
```

2.2

# Interlude: "Vectorization" in Python

list comprehensions (still sequential):

In [3]:

```
z = sum(x_i*w_i for x_i, w_i in zip(x, w))  
print(z)
```

2.2

A vectorized implementation using **NumPy**:

In [4]:

```
import numpy as np  
  
x_vec, w_vec = np.array(x), np.array(w)  
  
z = (x_vec.transpose()).dot(w_vec)  
print(z)  
  
z = x_vec.dot(w_vec)  
print(z)
```

2.2

2.2

# Interlude: "Vectorization" in Python

a)

```
def forloop(x, w):  
    z = 0.  
    for i in range(len(x)):  
        z += x[i] * w[i]  
    return z
```

b)

```
def listcomprehension(x, w):  
    return sum(x_i*w_i for x_i, w_i in zip(x, w))
```

c)

```
def vectorized(x, w):  
    return x_vec.dot(w_vec)
```

```
x, w = np.random.rand(100000), np.random.rand(100000)
```

Questions for you:

Which one is the fastest?

How much faster is the fastest one compared to the slowest one?

# Interlude: "Vectorization" in Python

In [6]: `%timeit -r 100 -n 10 forloop(x, w)`

38.9 ms  $\pm$  1.32 ms per loop (mean  $\pm$  std. dev. of 100 runs, 10 loops each)

In [7]: `%timeit -r 100 -n 10 listcomprehension(x, w)`

29.7 ms  $\pm$  842  $\mu$ s per loop (mean  $\pm$  std. dev. of 100 runs, 10 loops each)

In [8]: `%timeit -r 100 -n 10 vectorized(x_vec, w_vec)`

46.8  $\mu$ s  $\pm$  8.07  $\mu$ s per loop (mean  $\pm$  std. dev. of 100 runs, 10 loops each)

# Interlude: Connections and Parallel Computation

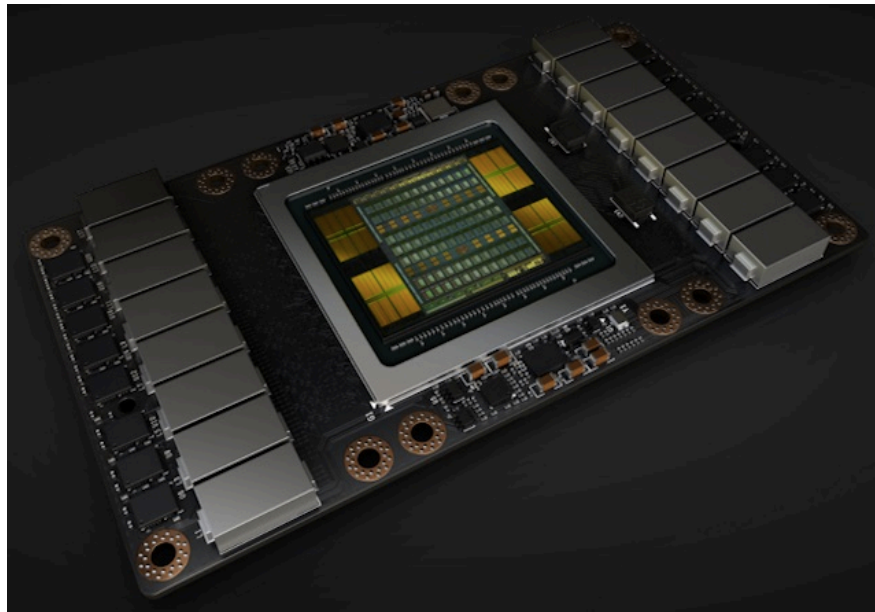


Image Source: <https://fossbytes.com/wp-content/uploads/2017/05/nvidia-volta-v100-gpu.jpg>

NVIDIA Volta with approx.  $2.1 \times 10^{10}$  transistors  
approx. only 10 connections per transistor



Image Source: <https://timedotcom.files.wordpress.com/2014/05/brain.jpg?w=1100&quality=85>

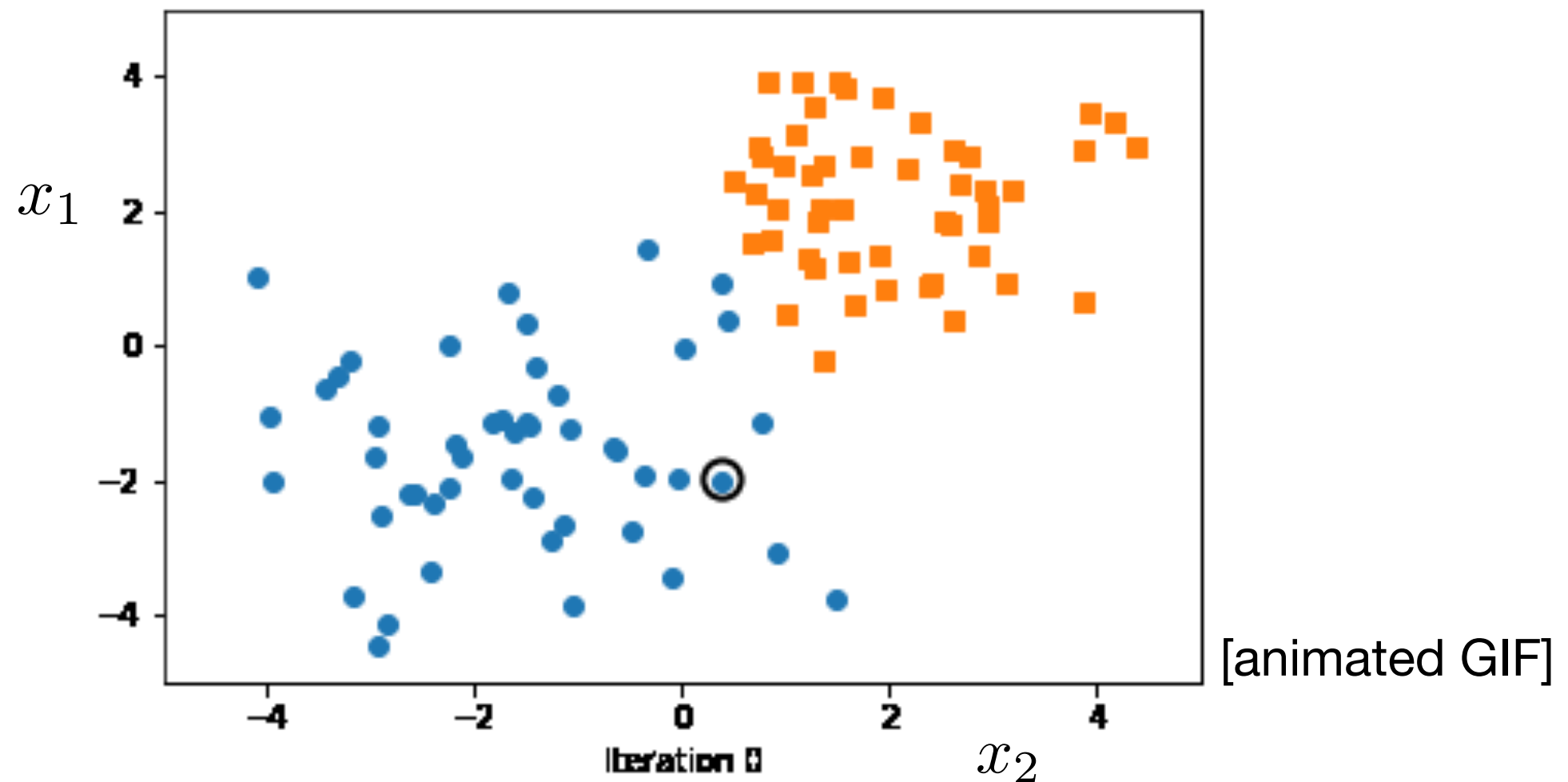
Brain with  $1.6 \times 10^{10}$  neurons  
 $10^4$  -  $10^5$  connections per neuron  
approx.  $10^{15}$  connections in total

# Implementing a perceptron in Python using NumPy and PyTorch

1. Brains and neuron models
2. The perceptron learning rule
3. Interlude: "vectorization" in Python
- 4. Implementing a perceptron in Python using NumPy and PyTorch**
5. Optional: The perceptron convergence theorem
6. Geometric intuition

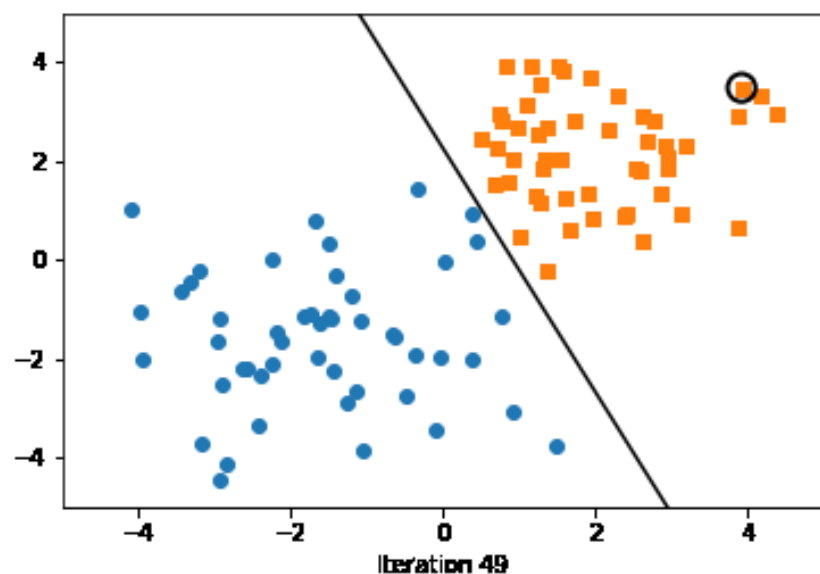
# Perceptron Learning Rule

Assume binary classification task, Perceptron finds decision boundary if classes are separable



# The Perceptron Learning Algorithm

- If correct: Do nothing if the prediction if output is equal to the target
- If incorrect, scenario **a)**:  
If output is 0 and target is 1, add input vector to weight vector
- If incorrect, scenario **b)**:  
If output is 1 and target is 0, subtract input vector from weight vector



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



# The Perceptron Learning Algorithm

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  $\mathbf{w} := \mathbf{0}^m$  (assume notation where weight incl. bias)
2. For every training epoch:
  - A. For every  $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
    - (a)  $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]\top} \mathbf{w})$
    - (b)  $err := (y^{[i]} - \hat{y}^{[i]})$
    - (c)  $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$

# Perceptron Code Examples

<https://github.com/rasbt/stat453-deep-learning-ss21/blob/master/L03>

# Why Do I "Make" You Understand NumPy?

```

3
4 class Perceptron():
5     def __init__(self, num_features):
6         self.num_features = num_features
7         self.weights = np.zeros((num_features, 1), dtype=np.float)
8         self.bias = np.zeros(1, dtype=np.float)
9
10
11
12
13
14
15
16
17
18     def forward(self, x):
19         linear = np.dot(x, self.weights) + self.bias
20         predictions = np.where(linear > 0., 1, 0)
21         return predictions
22
23     def backward(self, x, y):
24         predictions = self.forward(x)
25         errors = y - predictions
26         return errors
27
28     def train(self, x, y, epochs):
29         for e in range(epochs):
30
31             for i in range(y.shape[0]):
32
33                 errors = self.backward(x[i].reshape(1, self.num_features),
34 y[i]).reshape(-1)
35                 self.weights += (errors * x[i]).reshape(self.num_features,
36 1)
37                 self.bias += errors
38
39     def evaluate(self, x, y):
40         predictions = self.forward(x).reshape(-1)
41         accuracy = np.sum(predictions == y) / y.shape[0]
42         return accuracy

```

```

3
4 class Perceptron():
5     def __init__(self, num_features):
6         self.num_features = num_features
7
8
9         self.weights = torch.zeros(num_features, 1,
10                                     dtype=torch.float32, device=device)
11         self.bias = torch.zeros(1, dtype=torch.float32, device=device)
12
13         # placeholder vectors so they don't
14         # need to be recreated each time
15         self.ones = torch.ones(1)
16         self.zeros = torch.zeros(1)
17
18     def forward(self, x):
19         linear = torch.add(torch.mm(x, self.weights), self.bias)
20         predictions = torch.where(linear > 0., self.ones, self.zeros)
21         return predictions
22
23     def backward(self, x, y):
24         predictions = self.forward(x)
25         errors = y - predictions
26         return errors
27
28     def train(self, x, y, epochs):
29         for e in range(epochs):
30
31             for i in range(y.shape[0]):
32
33                 # use view because backward expects a matrix (i.e., 2D tensor)
34                 errors = self.backward(x[i].reshape(1, self.num_features),
35 y[i]).reshape(-1)
36                 self.weights += (errors * x[i]).reshape(self.num_features,
37 1)
38                 self.bias += errors
39
40     def evaluate(self, x, y):
41         predictions = self.forward(x).reshape(-1)
42         accuracy = torch.sum(predictions == y).float() / y.shape[0]
43         return accuracy

```

# The perceptron convergence theorem

1. Brains and neuron models
2. The perceptron learning rule
3. Interlude: "vectorization" in Python
4. Implementing a perceptron in Python using NumPy and PyTorch
- 5. Optional: The perceptron convergence theorem**
6. Geometric intuition

# Perceptron Convergence Theorem

Let

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

$$\begin{aligned} \forall y^{[i]} \in \mathcal{D}_1 : y^{[i]} &= 1 \\ \forall y^{[i]} \in \mathcal{D}_2 : y^{[i]} &= 0 \end{aligned} \quad \text{and} \quad \mathcal{D}_1 \cup \mathcal{D}_2 = \mathcal{D}$$

Assume the input vectors come from two linearly separable classes such that a feasible weight vector  $\mathbf{w}^*$  exists.

The perceptron learning algorithm is guaranteed to converge to a weight vector in the feasible region in a finite number of iterations such that

$$\forall \mathbf{x}^{[i]} \in \mathcal{D}_1 : \mathbf{w}^\top \mathbf{x}^{[i]} > 0$$

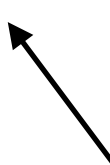
$$\forall \mathbf{x}^{[i]} \in \mathcal{D}_2 : \mathbf{w}^\top \mathbf{x}^{[i]} \leq 0$$

# Perceptron Convergence Theorem -- Proof

Let us slightly rewrite the update rule (upon misclassification) for convenience when we construct the proof:

$$\mathbf{w}^{[i+1]} = \mathbf{w}^{[i]} + \mathbf{x}^{[i]} \quad \text{if } (\mathbf{w}^{[i]})^\top \mathbf{x}^{[i]} \leq 0, \mathbf{x}^{[i]} \in \mathcal{D}_1$$

$$\mathbf{w}^{[i+1]} = \mathbf{w}^{[i]} - \mathbf{x}^{[i]} \quad \text{if } (\mathbf{w}^{[i]})^\top \mathbf{x}^{[i]} > 0, \mathbf{x}^{[i]} \in \mathcal{D}_2$$




Here  $[i + 1]$  refers to the weight vector of the next training example (that is, the weight after updating)

# Perceptron Convergence Theorem -- Proof

From the previous slide:

$$\mathbf{w}^{[i+1]} = \mathbf{w}^{[i]} + \mathbf{x}^{[i]} \quad \text{if } (\mathbf{w}^{[i]})^\top \mathbf{x}^{[i]} \leq 0, \mathbf{x}^{[i]} \in \mathcal{D}_1$$

We can rewrite this as follows:

$$\mathbf{w}^{[i+1]} = \mathbf{w}^{[0]} + \mathbf{x}^{[1]} + \dots + \mathbf{x}^{[i]}$$


Also, we can drop this term if we initialize the weight vector as  $\mathbf{0}^m$

$$\mathbf{w}^{[i+1]} = \mathbf{x}^{[1]} + \dots + \mathbf{x}^{[i]}$$



# Perceptron Convergence Theorem -- Proof

From the previous slide, the update rule:

$$\mathbf{w}^{[i+1]} = \mathbf{x}^{[1]} + \dots + \mathbf{x}^{[i]}$$

Let's multiply both sides by  $w^*$  :

$$(\mathbf{w}^*)^T \mathbf{w}^{[i+1]} = (\mathbf{w}^*)^T \mathbf{x}^{[1]} + \dots + (\mathbf{w}^*)^T \mathbf{x}^{[i]}$$

All these terms are  $> 0$ , because remember that we have

$$\mathbf{w}^{[i+1]} = \mathbf{w}^{[i]} + \mathbf{x}^{[i]} \quad \text{if} \quad (\mathbf{w}^{[i]})^T \mathbf{x}^{[i]} \leq 0, \mathbf{x}^{[i]} \in \mathcal{D}_1$$

so the updates are all to make the net inputs more positive

Now, let  $\alpha = \min_{x^{[j]}} (\mathbf{w}^*)^T \mathbf{x}^{[j]}, j = 1, \dots, i$

then  $(\mathbf{w}^*)^T \mathbf{w}^{[i+1]} \geq \alpha i$

# Perceptron Convergence Theorem -- Proof

From the previous slide, we had the inequality:

$$\underbrace{(\mathbf{w}^*)^T \mathbf{w}^{[i+1]}} \geq \alpha i$$

Using the Cauchy-Schwarz inequality, we can then say

$$||\mathbf{w}^*||^2 \cdot ||\mathbf{w}^{[i+1]}||^2 \geq \left( (\mathbf{w}^*)^T \mathbf{w}^{[i+1]} \right)^2$$

as well as

$$||\mathbf{w}^*||^2 \cdot ||\mathbf{w}^{[i+1]}||^2 \geq (\alpha i)^2$$

So, we can finally define the lower bound of the size of the weights

$$||\mathbf{w}^{[i+1]}||^2 \geq \frac{\alpha^2 i^2}{||\mathbf{w}^*||^2}$$

# Perceptron Convergence Theorem -- Proof

Now that we defined the lower bound of the size of the weights, let us get the upper bound.

For that, let's go back to the update rule

$$\mathbf{w}^{[i+1]} = \mathbf{w}^{[i]} + \mathbf{x}^{[i]} \quad \text{if } (\mathbf{w}^{[i]})^T \mathbf{x}^{[i]} \leq 0, \mathbf{x}^{[i]} \in \mathcal{D}_1$$

and apply the squared L2 norm on both sides

$$\begin{aligned} ||\mathbf{w}^{[i+1]}||^2 &= ||\mathbf{w}^{[i]} + \mathbf{x}^{[i]}||^2 \\ &= ||\mathbf{w}^{[i]}||^2 + 2(\mathbf{x}^{[i]})^T \mathbf{w}^{[i]} + ||\mathbf{x}^{[i]}||^2 \end{aligned}$$

# Perceptron Convergence Theorem -- Proof

Now that we defined the lower bound of the size of the weights, let us get the upper bound.

For that, let's go back to the update rule

$$\mathbf{w}^{[i+1]} = \mathbf{w}^{[i]} + \mathbf{x}^{[i]} \quad \text{if } (\mathbf{w}^{[i]})^T \mathbf{x}^{[i]} \leq 0, \mathbf{x}^{[i]} \in \mathcal{D}_1$$

and apply the squared L2 norm on both sides

$$\begin{aligned} ||\mathbf{w}^{[i+1]}||^2 &= ||\mathbf{w}^{[i]} + \mathbf{x}^{[i]}||^2 \\ &= ||\mathbf{w}^{[i]}||^2 + 2(\mathbf{x}^{[i]})^T \mathbf{w}^{[i]} + ||\mathbf{x}^{[i]}||^2 \end{aligned}$$

Leads to

$$||\mathbf{w}^{[i+1]}||^2 \leq ||\mathbf{w}^{[i]}||^2 + ||\mathbf{x}^{[i]}||^2$$

# Perceptron Convergence Theorem -- Proof

Now that we defined the lower bound of the size of the weights, let us get the upper bound.

For that, let's go back to the update rule

$$\mathbf{w}^{[i+1]} = \mathbf{w}^{[i]} + \mathbf{x}^{[i]} \quad \text{if } \underbrace{(\mathbf{w}^{[i]})^T \mathbf{x}^{[i]} \leq 0}_{\text{implies}} , \mathbf{x}^{[i]} \in \mathcal{D}_1$$

and apply the squared L2 norm on both sides

$$\begin{aligned} ||\mathbf{w}^{[i+1]}||^2 &= ||\mathbf{w}^{[i]} + \mathbf{x}^{[i]}||^2 \\ &= ||\mathbf{w}^{[i]}||^2 + \underbrace{2(\mathbf{x}^{[i]})^T \mathbf{w}^{[i]}}_{\leq 0} + ||\mathbf{x}^{[i]}||^2 \end{aligned}$$

Thus

$$||\mathbf{w}^{[i+1]}||^2 \leq ||\mathbf{w}^{[i]}||^2 + ||\mathbf{x}^{[i]}||^2$$

# Perceptron Convergence Theorem -- Proof

Now, we simply expand:

$$||\mathbf{w}^{[i+1]}||^2 \leq ||\mathbf{w}^{[i]}||^2 + ||\mathbf{x}^{[i]}||^2$$

$$||\mathbf{w}^{[i+1]}||^2 \leq ||\mathbf{w}^{[i-1]}||^2 + ||\mathbf{x}^{[i-1]}||^2 + ||\mathbf{x}^{[i]}||^2$$

$$||\mathbf{w}^{[i+1]}||^2 \leq ||\mathbf{w}^{[i-2]}||^2 + ||\mathbf{x}^{[i-2]}||^2 + ||\mathbf{x}^{[i-1]}||^2 + ||\mathbf{x}^{[i]}||^2$$

...

$$||\mathbf{w}^{[i+1]}||^2 \leq ||\mathbf{w}^{[1]}||^2 + \sum_{j=1}^i ||\mathbf{x}^{[j]}||^2$$

$$||\mathbf{w}^{[i+1]}||^2 \leq \sum_{j=1}^i ||\mathbf{x}^{[j]}||^2$$

# Perceptron Convergence Theorem -- Proof

From  $||\mathbf{w}^{[i+1]}||^2 \leq \sum_{j=1}^i ||\mathbf{x}^{[j]}||^2$  we can finally get the [upper bound](#).

Let  $\beta = \max ||\mathbf{x}^{[j]}||^2$

then  $||\mathbf{w}^{[i+1]}||^2 \leq \beta i$

# Perceptron Convergence Theorem -- Proof

lower bound

$$||\mathbf{w}^{[i+1]}||^2 \geq \frac{\alpha^2 i^2}{||\mathbf{w}^*||^2}$$

upper bound

$$||\mathbf{w}^{[i+1]}||^2 \leq \beta i$$

combined

$$\beta i \geq ||\mathbf{w}^{[i+1]}||^2 \geq \frac{\alpha^2 i^2}{||\mathbf{w}^*||^2}$$

$$i \leq \frac{\beta ||\mathbf{w}^*||^2}{\alpha^2}$$

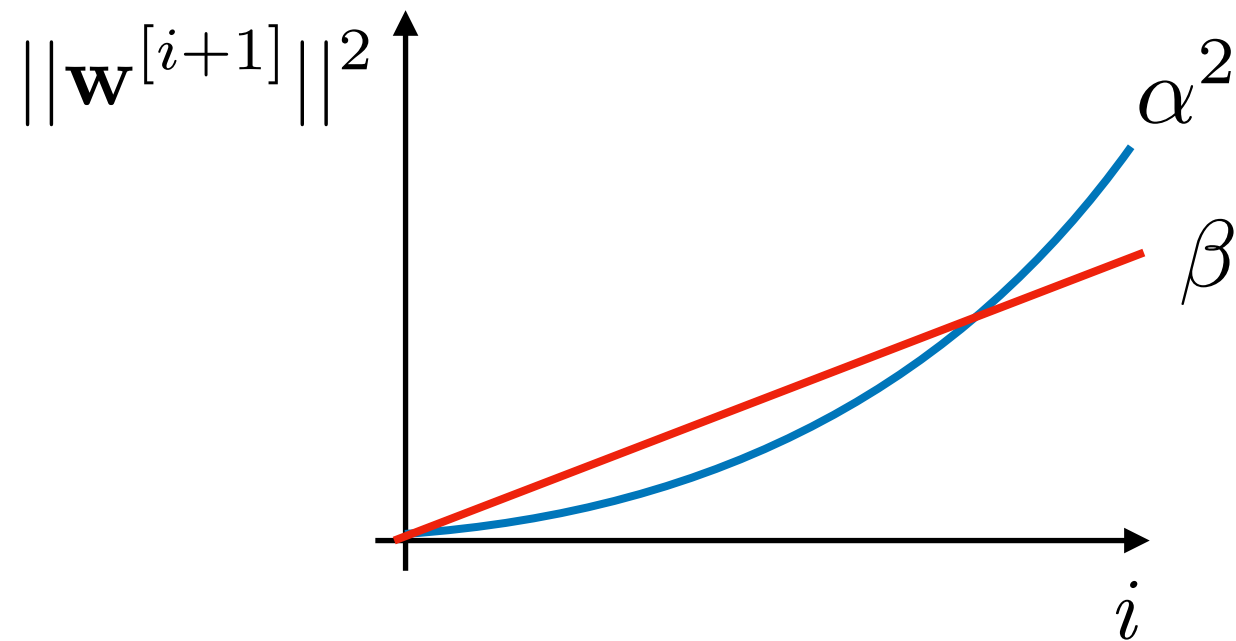
Since the number of iterations  $i$  has an upper bound,  
we can conclude that the weights only change a finite number of times and will converge if the classes are linearly separable.





# Perceptron Convergence Theorem -- Proof

$$\beta i \geq ||\mathbf{w}^{[i+1]}||^2 \geq \alpha^2 i^2$$



# Perceptron Convergence Theorem -- Proof

In the convergence theorem, we can assume that  $||\mathbf{w}^*|| = 1$   
(so you may remove it from all equations)

$$\beta_i \geq ||\mathbf{w}^{[i+1]}||^2 \geq \frac{\alpha^2 i^2}{||\mathbf{w}^*||^2} \quad \Leftrightarrow \quad \beta_i \geq ||\mathbf{w}^{[i+1]}||^2 \geq \alpha^2 i^2$$

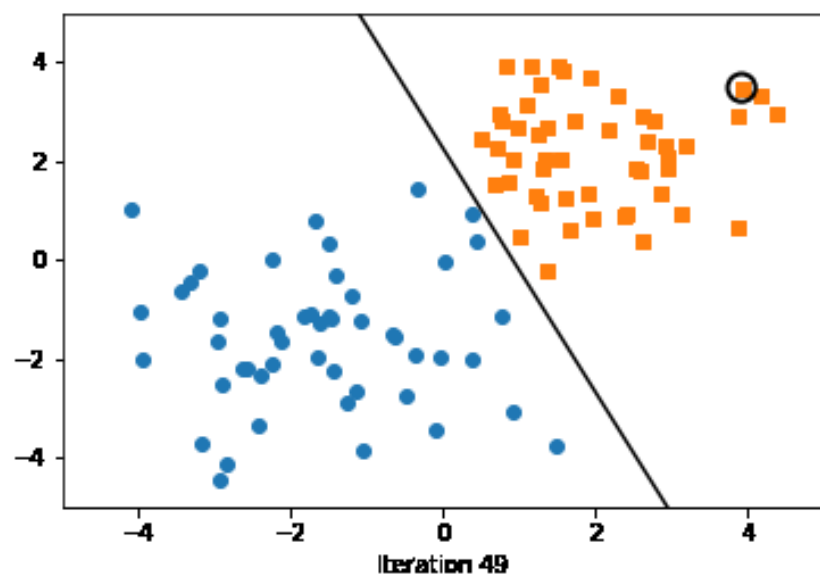
$$i \leq \frac{\beta ||\mathbf{w}^*||^2}{\alpha^2} \quad \Leftrightarrow \quad i \leq \frac{\beta}{\alpha^2}$$

# Geometric Intuition

1. Brains and neuron models
2. The perceptron learning rule
3. Interlude: "vectorization" in Python
4. Implementing a perceptron in Python using NumPy and PyTorch
5. Optional: The perceptron convergence theorem
- 6. Geometric intuition**

# The Perceptron Learning Algorithm

- If correct: Do nothing if the prediction if output is equal to the target
- If incorrect, scenario **a)**:  
If output is 0 and target is 1, add input vector to weight vector
- If incorrect, scenario **b)**:  
If output is 1 and target is 0, subtract input vector from weight vector



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

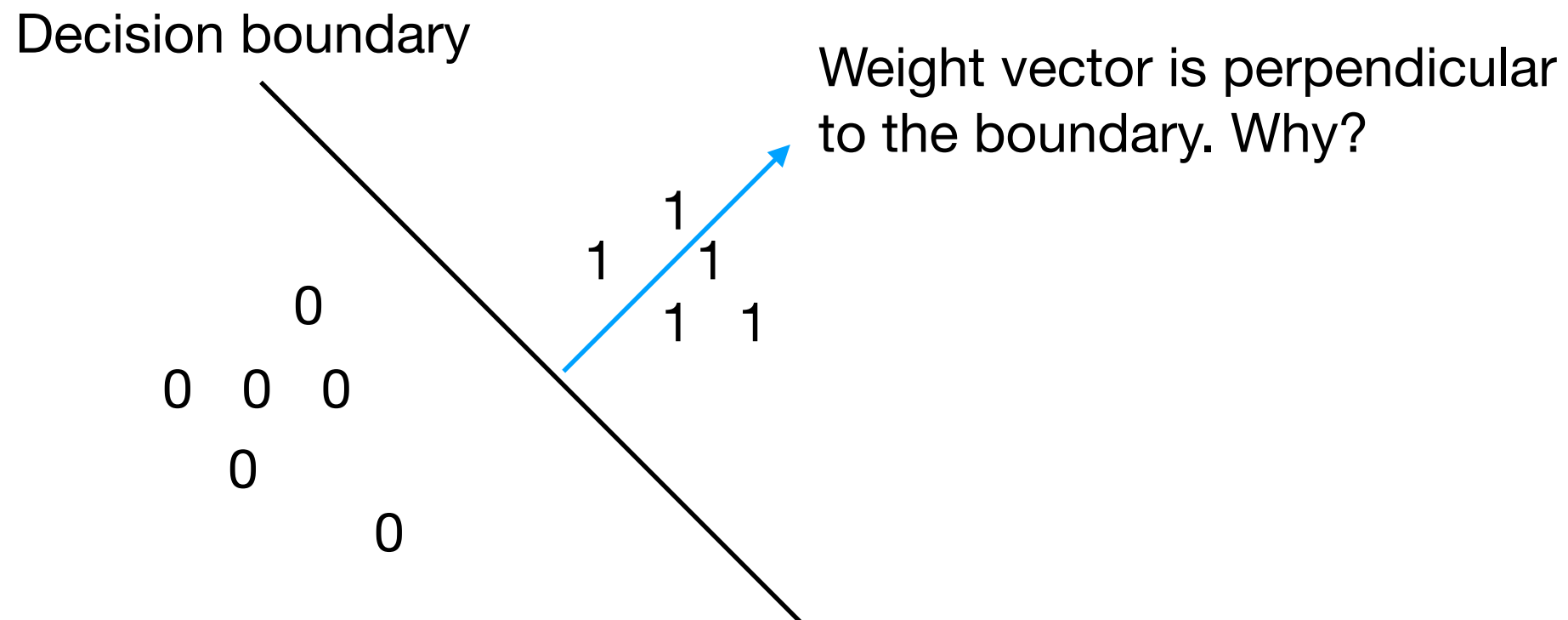
# The Perceptron Learning Algorithm

Let

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

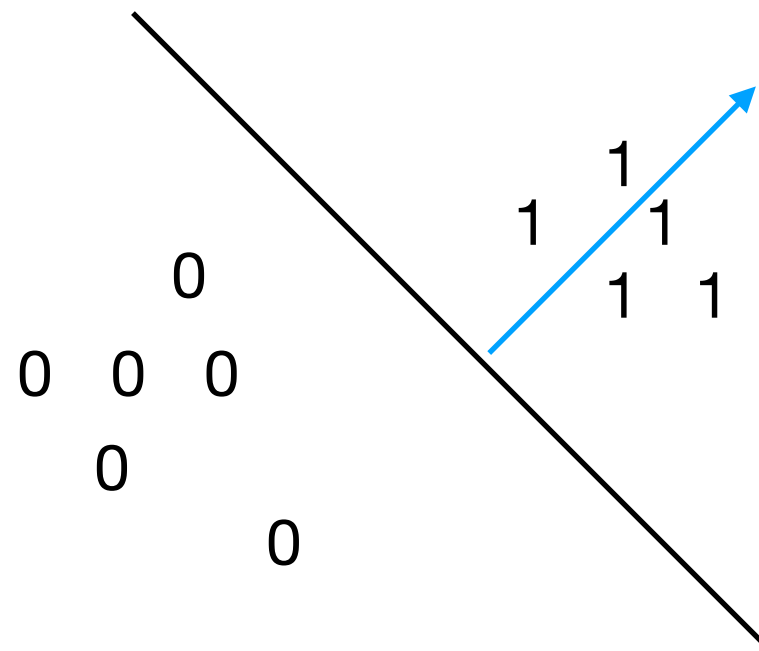
1. Initialize  $\mathbf{w} := \mathbf{0}^m$  (assume notation where weight incl. bias)
2. For every training epoch:
  - A. For every  $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
    - (a)  $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]\top} \mathbf{w})$
    - (b)  $err := (y^{[i]} - \hat{y}^{[i]})$
    - (c)  $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$

# Geometric Intuition



# Geometric Intuition

Decision boundary



Weight vector is perpendicular to the boundary. Why?

Remember,

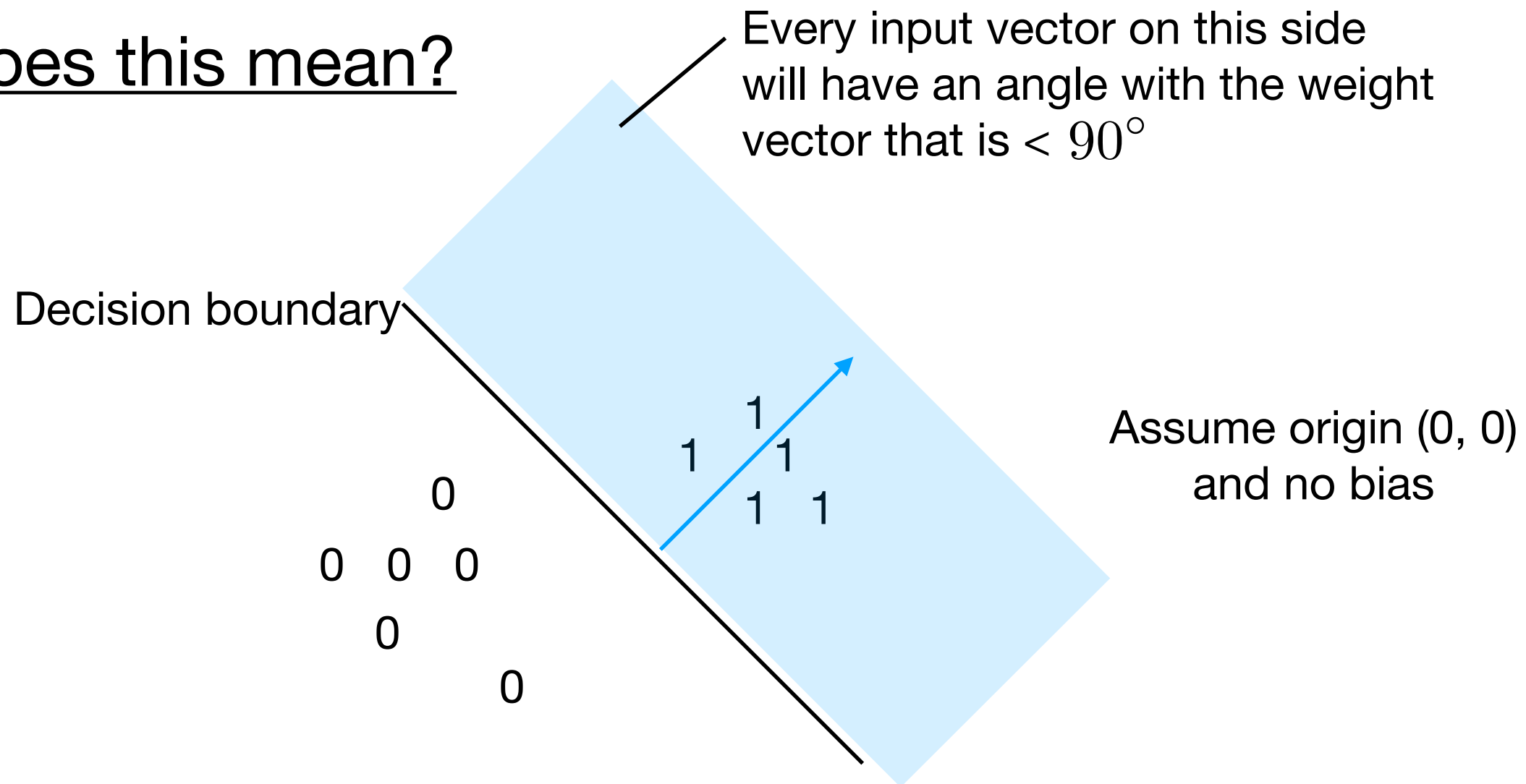
$$\hat{y} = \begin{cases} 0, & \mathbf{w}^T \mathbf{x} \leq 0 \\ 1, & \mathbf{w}^T \mathbf{x} > 0 \end{cases}$$

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

So this needs to be 0 at the boundary,  
and it is zero at  $90^\circ$

# Geometric Intuition

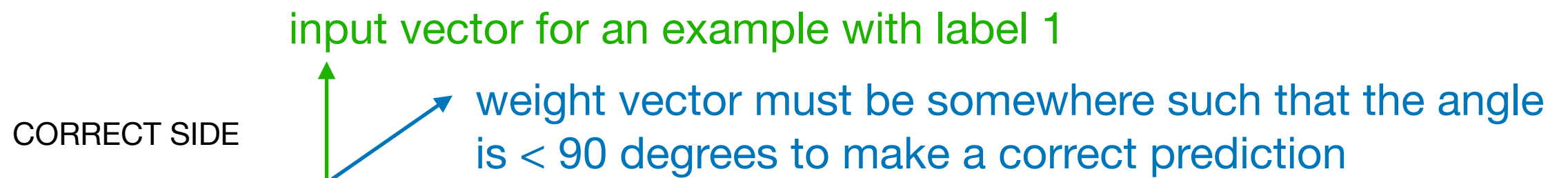
What else does this mean?



So, we could scale the weights and/or inputs by an arbitrary factor and still get the same classification results  
(but large inputs will take much longer to converge if you check the bounds we defined previously ...)



# Geometric Intuition

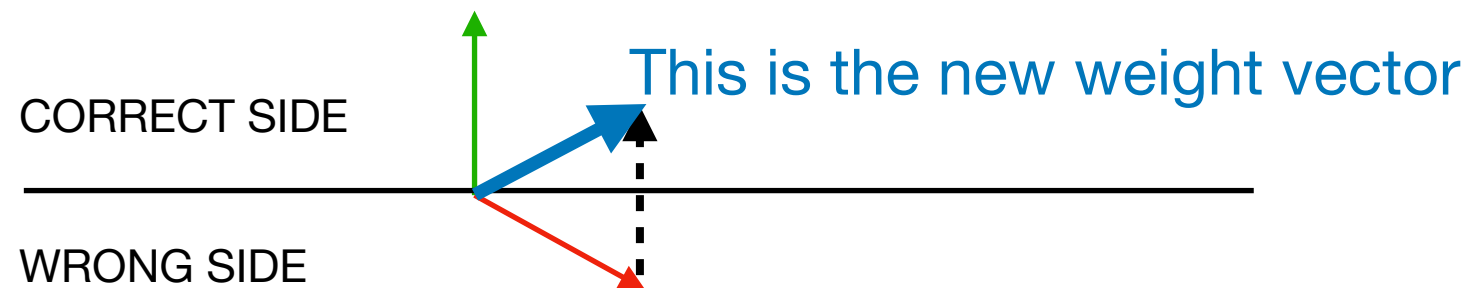


The dot product will then be positive, i.e.,  $> 0$ , since

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

# Geometric Intuition

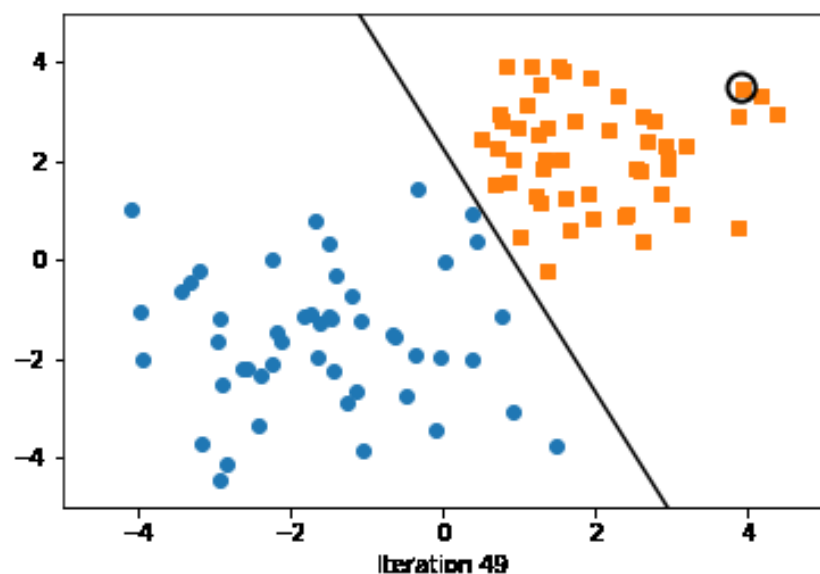
input vector for an example with label 1



For this weight vector, we make a wrong prediction;  
hence, we update

# The Perceptron Learning Algorithm

- If correct: Do nothing if the prediction if output is equal to the target
- If incorrect, scenario **a)**:  
If output is 0 and target is 1, add input vector to weight vector
- If incorrect, scenario **b)**:  
If output is 1 and target is 0, subtract input vector from weight vector



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

# Perceptron Conclusions

The (classic) Perceptron has many problems  
(as discussed in the previous lecture)

- Linear classifier, no non-linear boundaries possible
- Binary classifier
- 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





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



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

# 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, AI researcher & author of the "Perceptrons" book

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