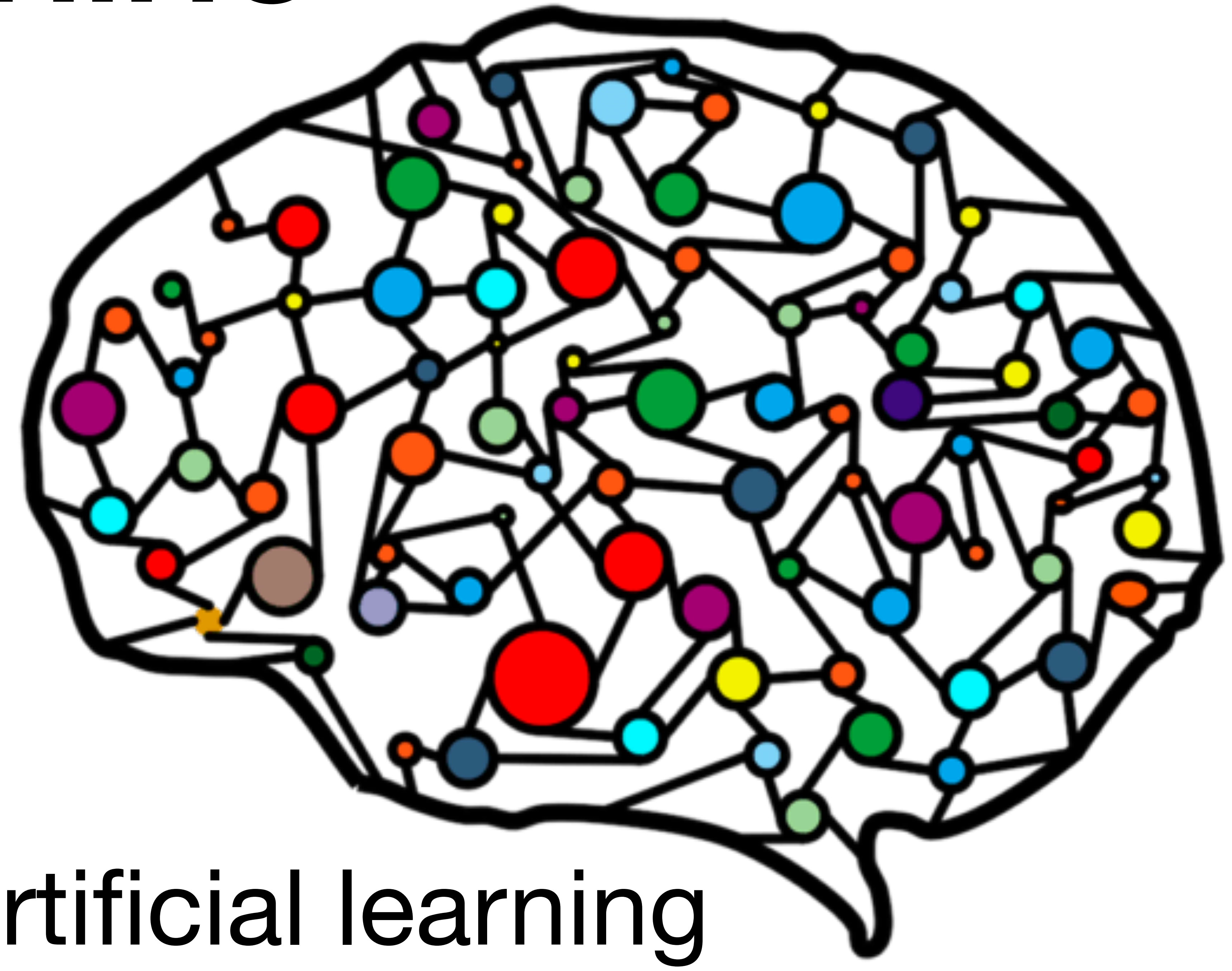
General Principles of Human and Machine Learning



Lecture 2: Origins of biological and artificial learning

Dr. Charley Wu

https://hmc-lab.com/GPHML.html

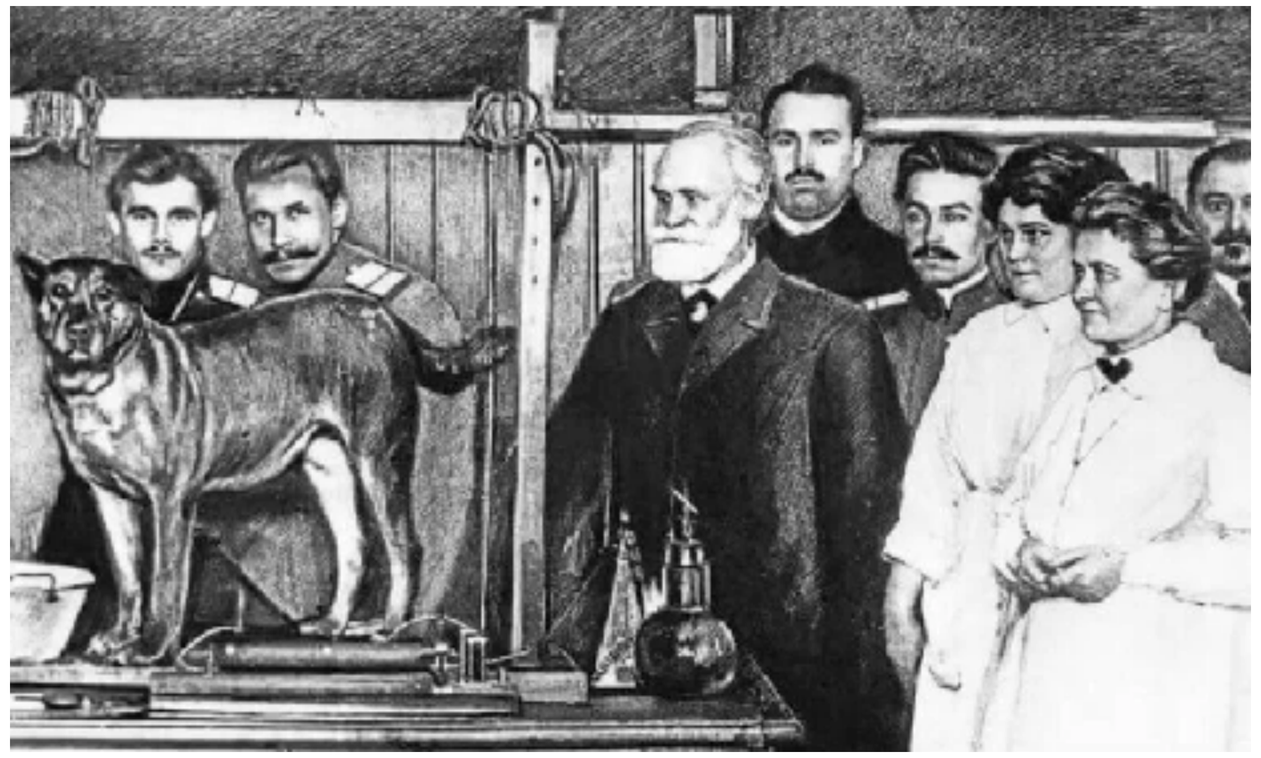
Organization

- To allow time for people to travel between classes
  - Lectures: 12:15 - 13:45 on Tuesdays
  - Tutorials: 16:15 - 17:30 on Wednesday
- Anyone not yet registered?
  - Send me an email today with your student number charley.wu@uni-tuebingen.de

Lesson plan

1. Behavioralism

- Understanding intelligence through behavior



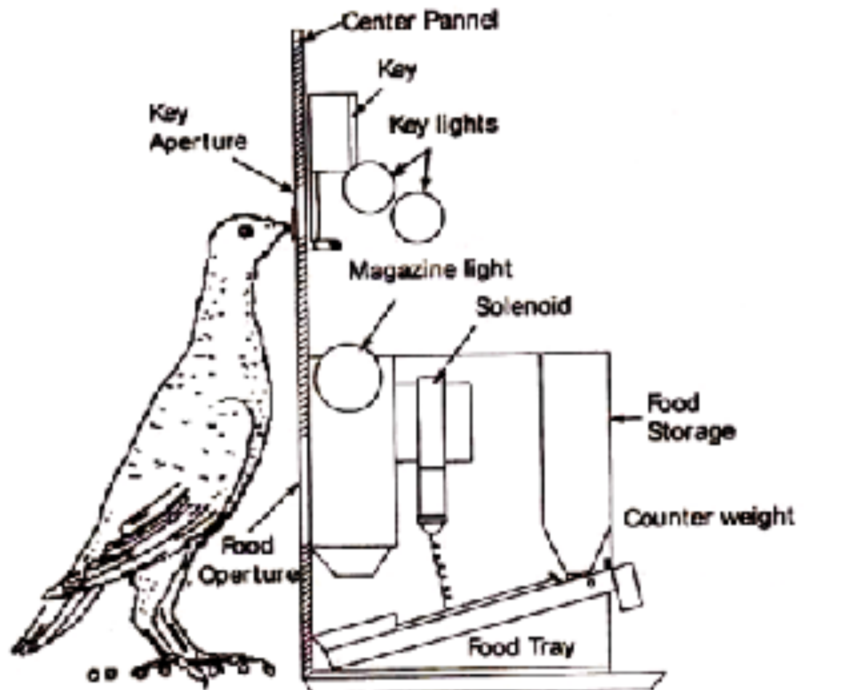
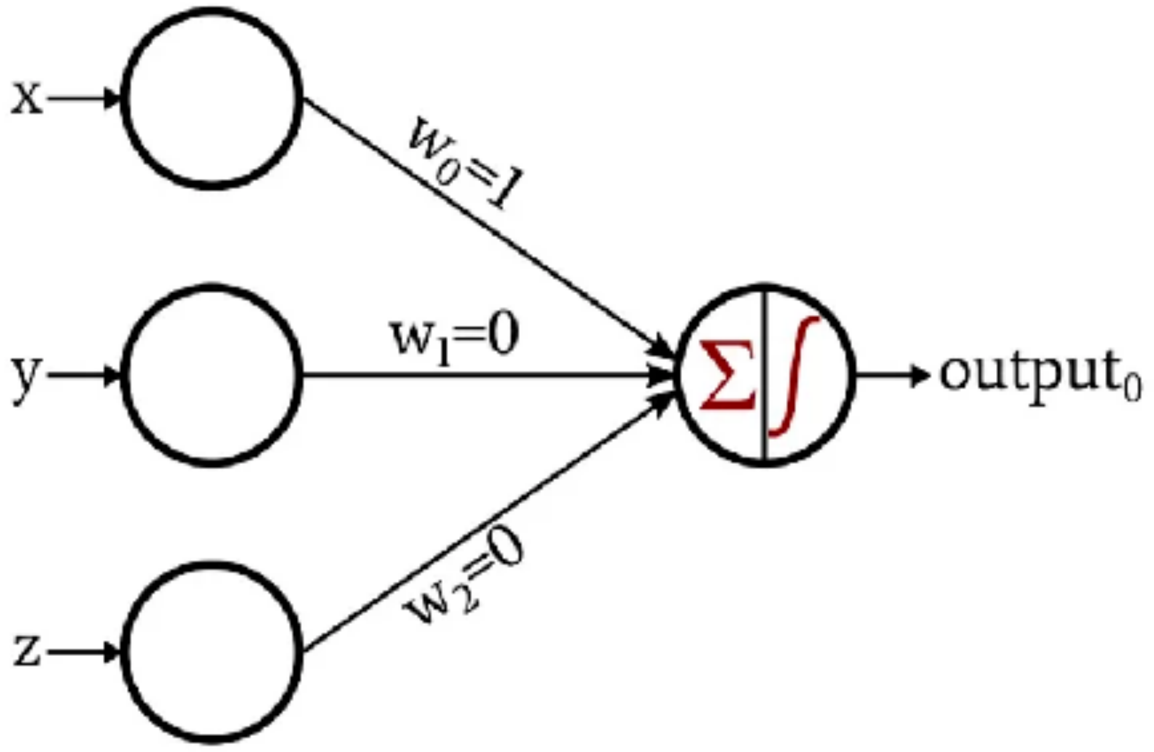


Illustration. Skinner box as adapted for the pigeon.

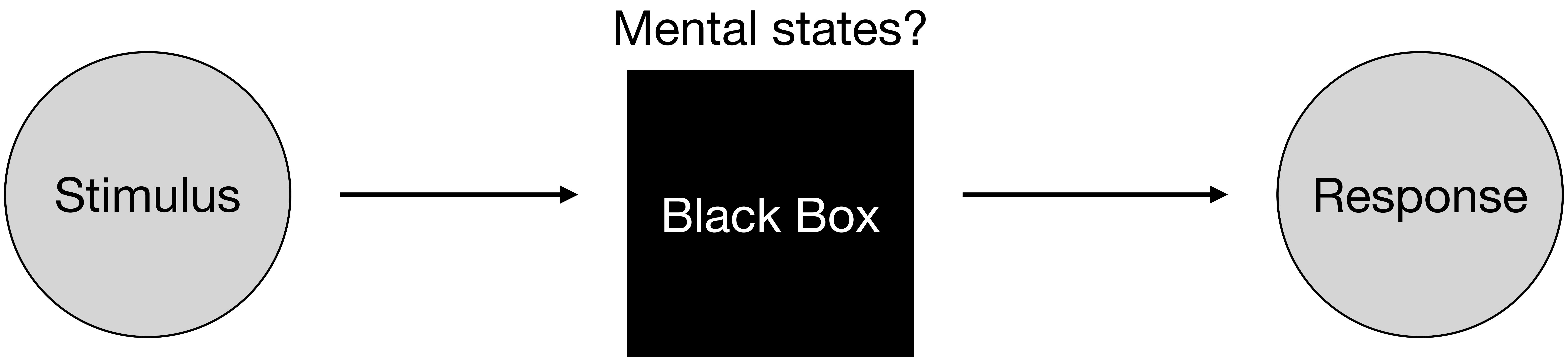
2. Connectionism

- Understanding intelligence through artificial neural networks



Behaviorism

- [*noun* Psychology.] An approach to understanding the behavior of humans and animals that emerged in the early 1900s
  - Generally tries to focus on outward observable behavior rather than hidden inner mental states
  - One of the earliest programs to empirically study biological intelligence and learning



Varieties of Behaviorism

John B. Watson

B.F. Skinner

Methodological Behaviorism ⟷ **Radical Behaviorism**

- Thoughts and feelings exist, but cannot be the target of scientific study
- Only public events can be objectively observed and studied scientifically

- Internal processes are also the target of scientific study
- But they are fully controlled by environmental variables just as environmental variables control behavior

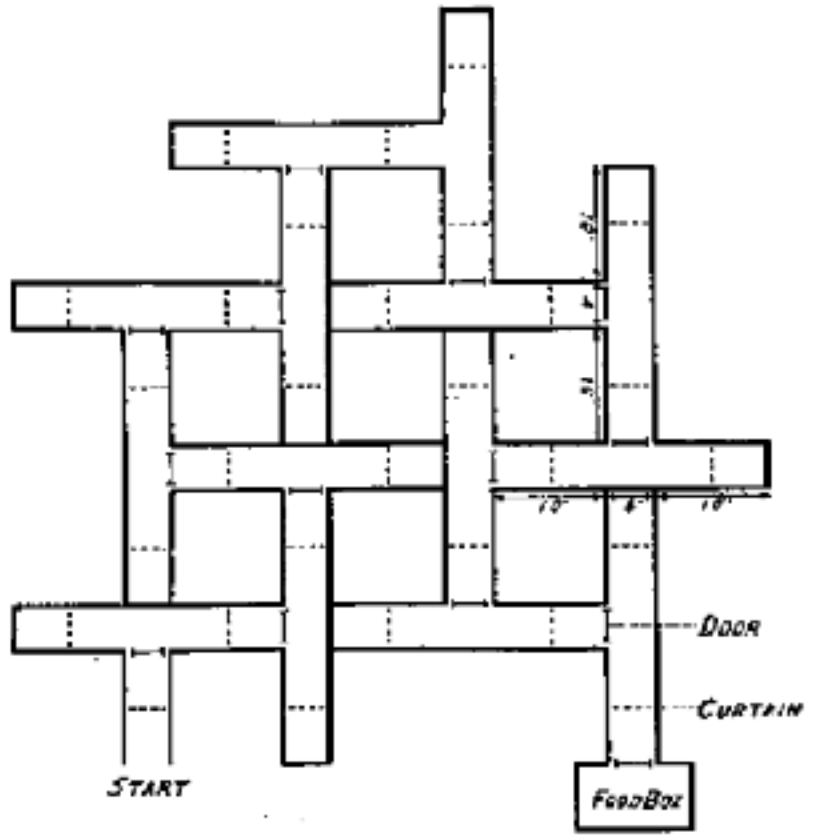
A brief timeline of early research on learning

Thorndike (1911)

Pavlov (1927)

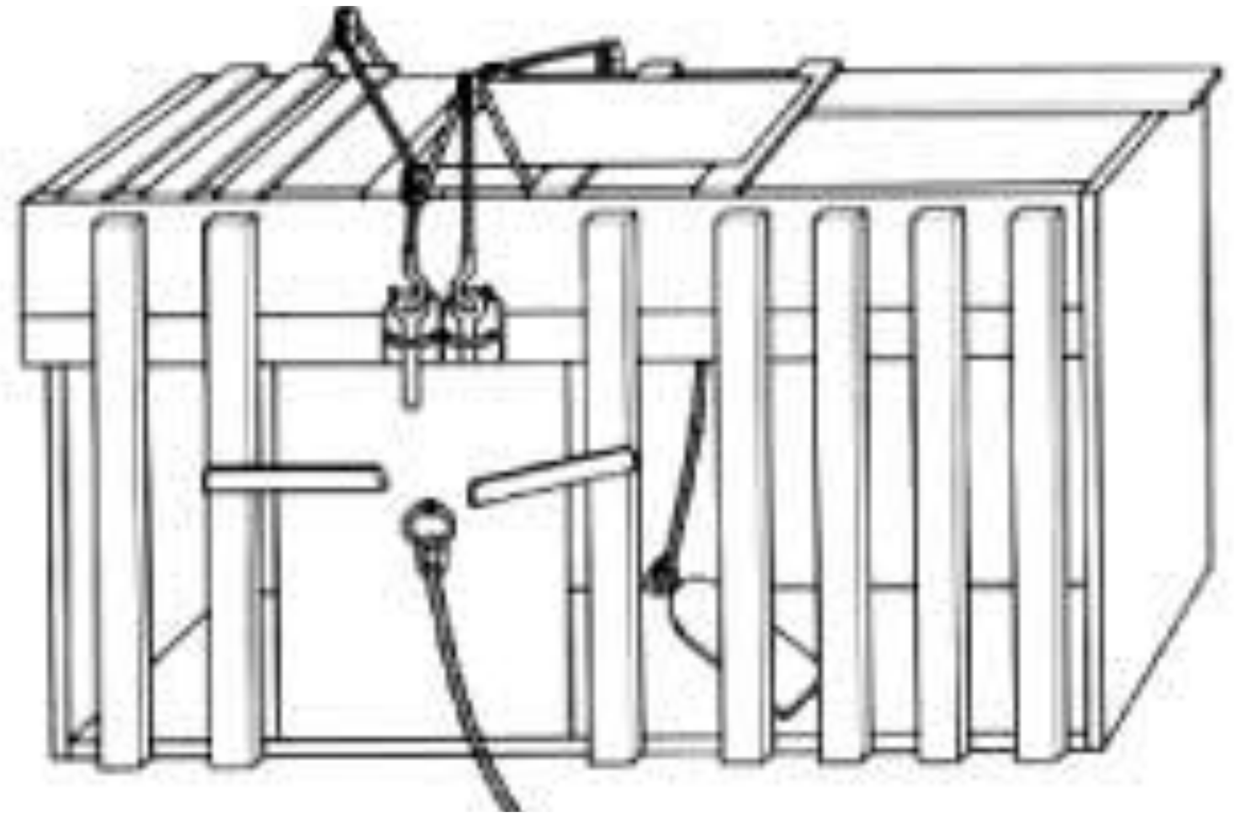
Skinner (1938)

Tolman (1948)



Plan of maze
14-Unit T-Alley Maze
FIG. 1
(From M. H. Elliott, The effect of change of reward on the maze performance of rats. *Univ. Calif. Publ. Psychol.*, 1928, 4, p. 20.)

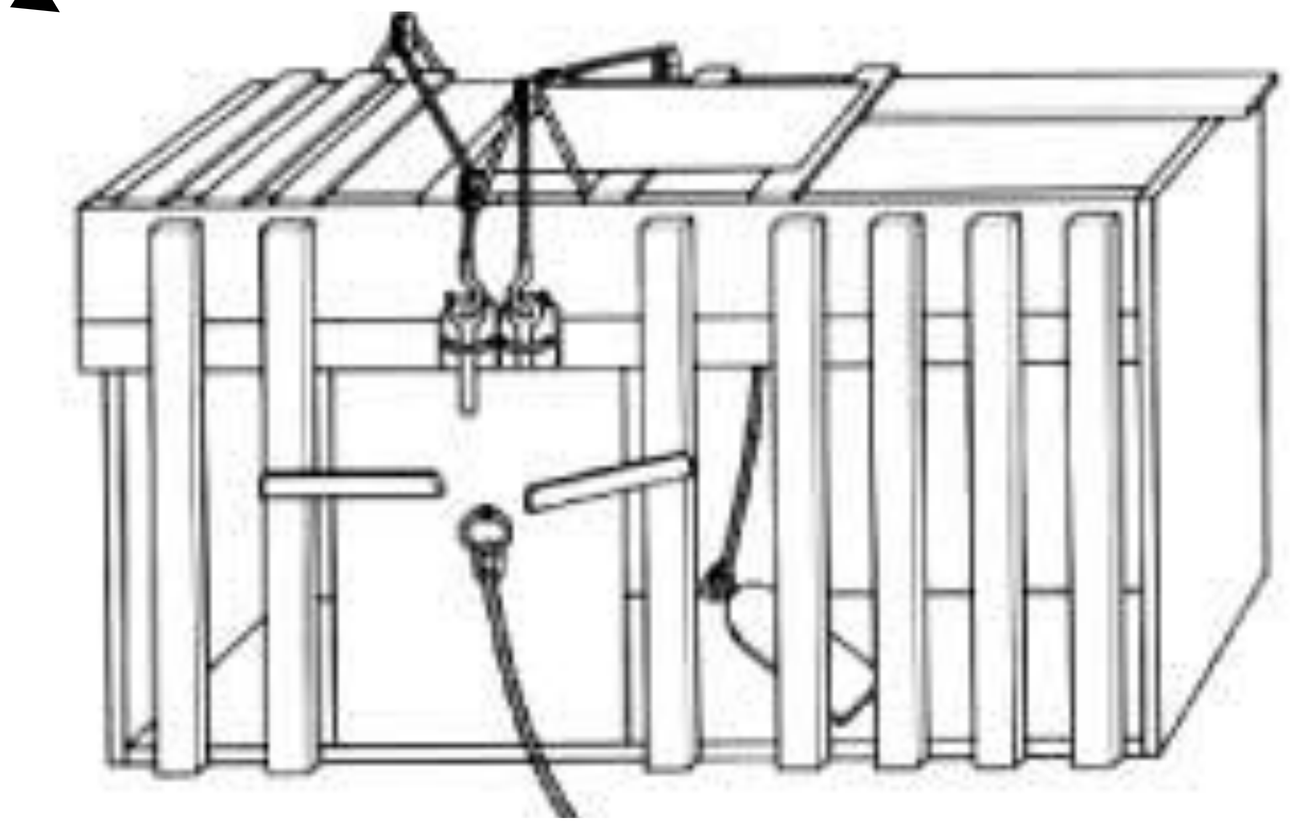
Thorndike's (1911) Law of Effect



Puzzle Box

Thorndike's (1911) Law of Effect

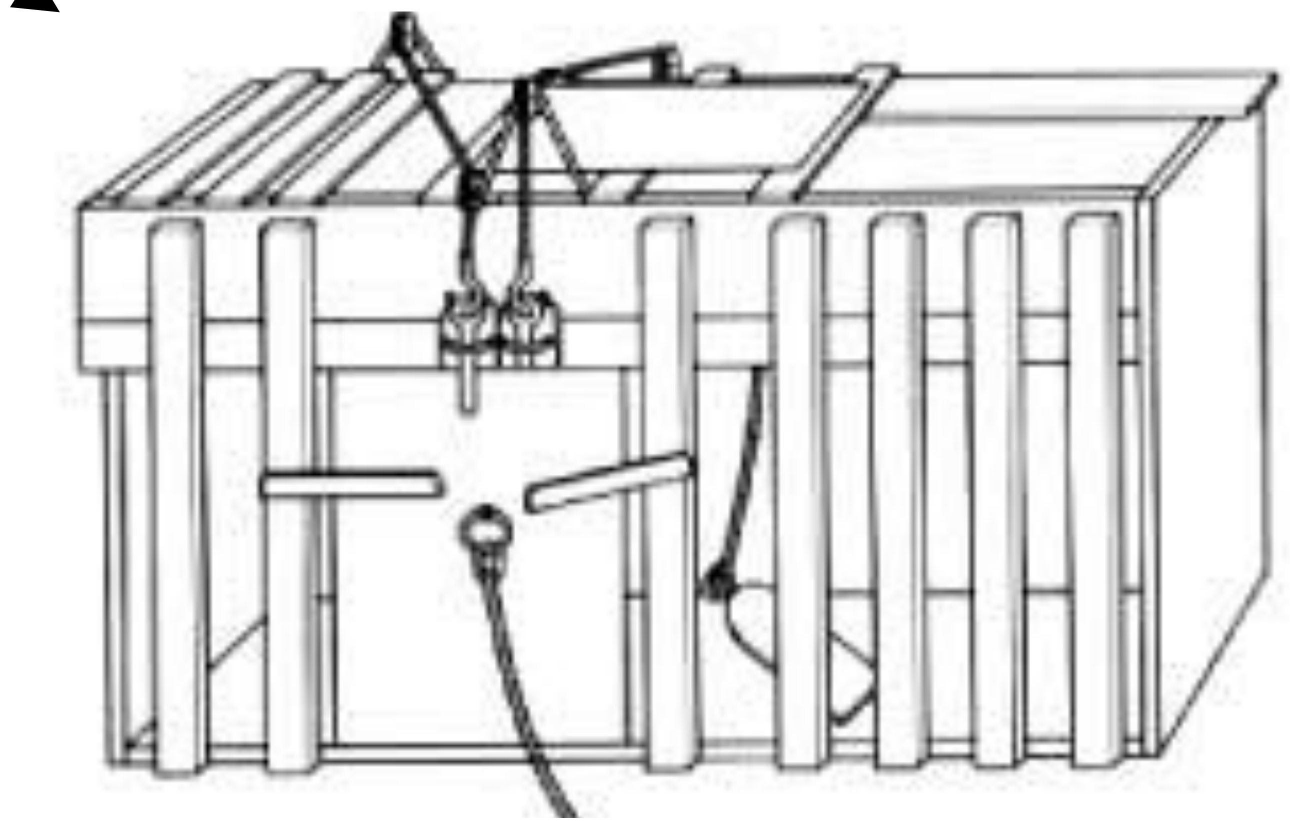
Cat



Puzzle Box

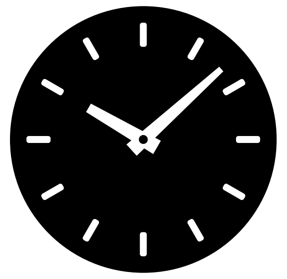
Thorndike’s (1911) Law of Effect

Cat



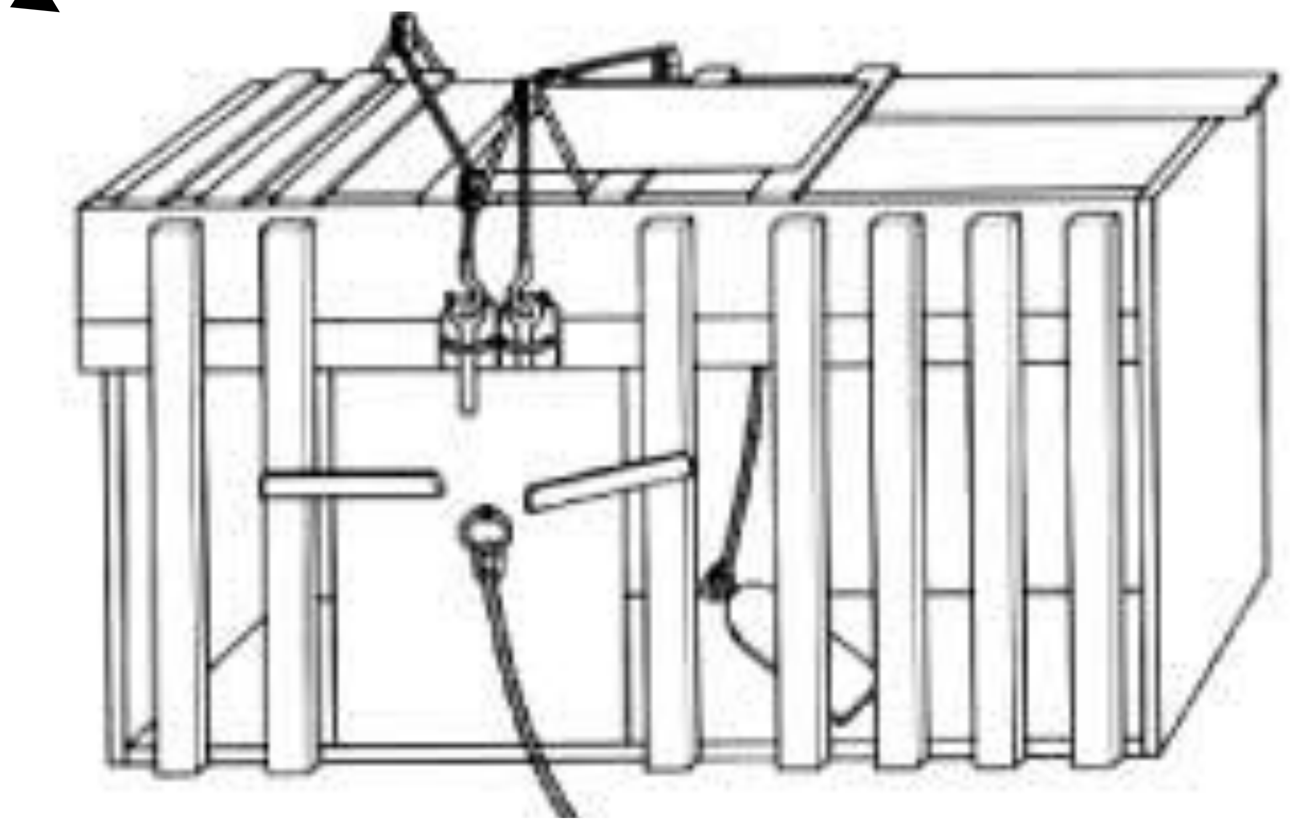
Puzzle Box

Time to escape



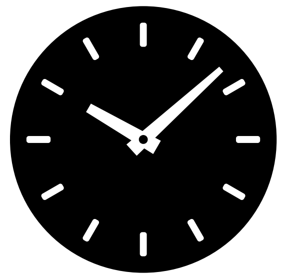
Thorndike's (1911) Law of Effect

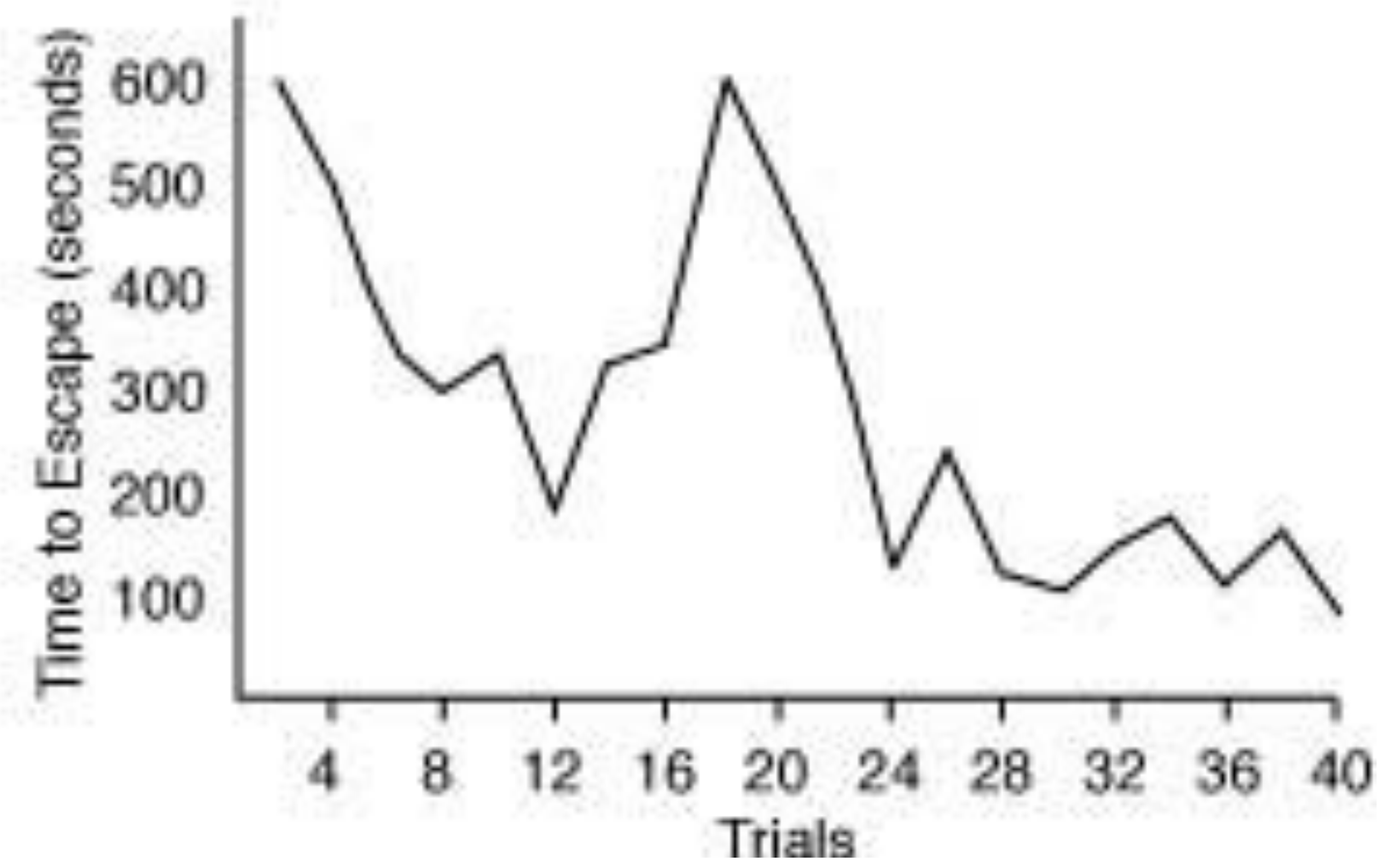
Cat



Puzzle Box

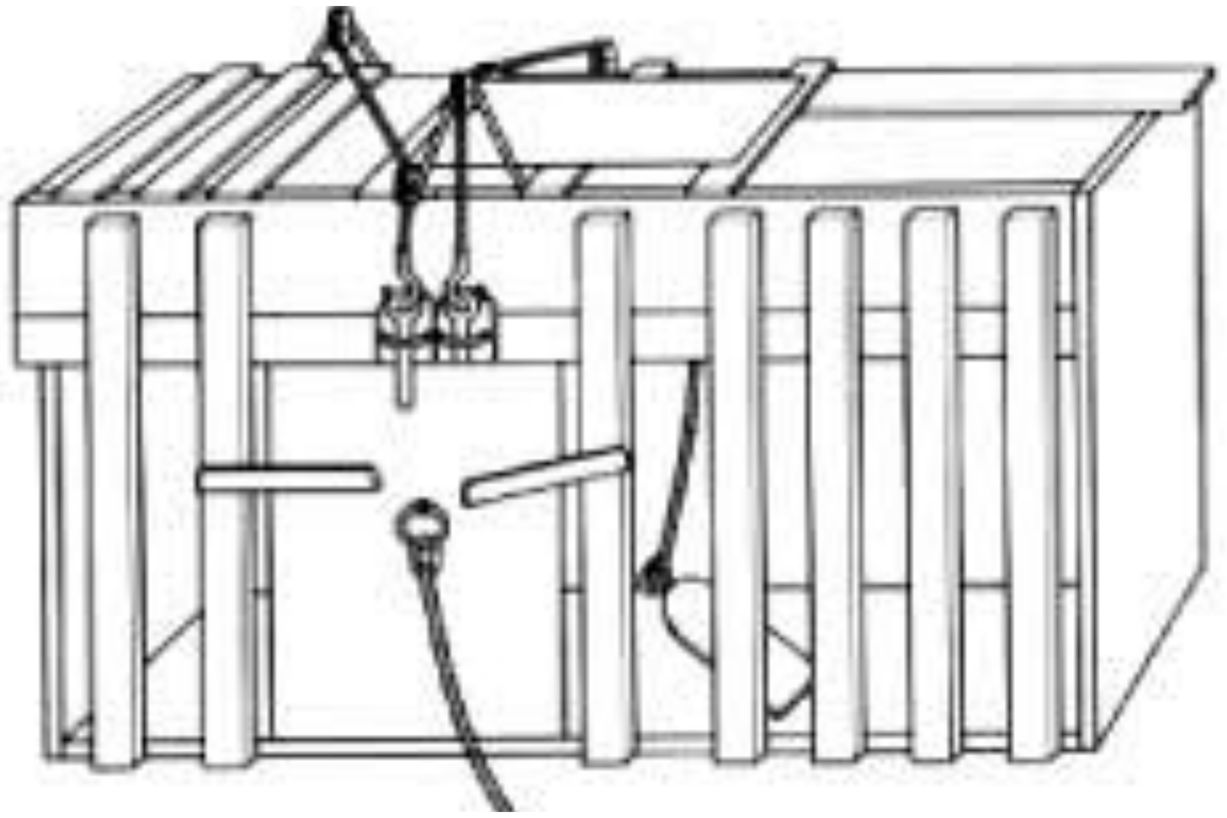
Time to escape





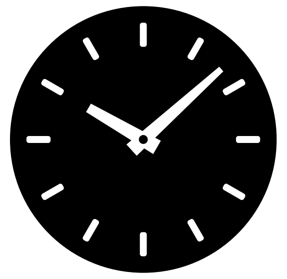
Thorndike's (1911) Law of Effect

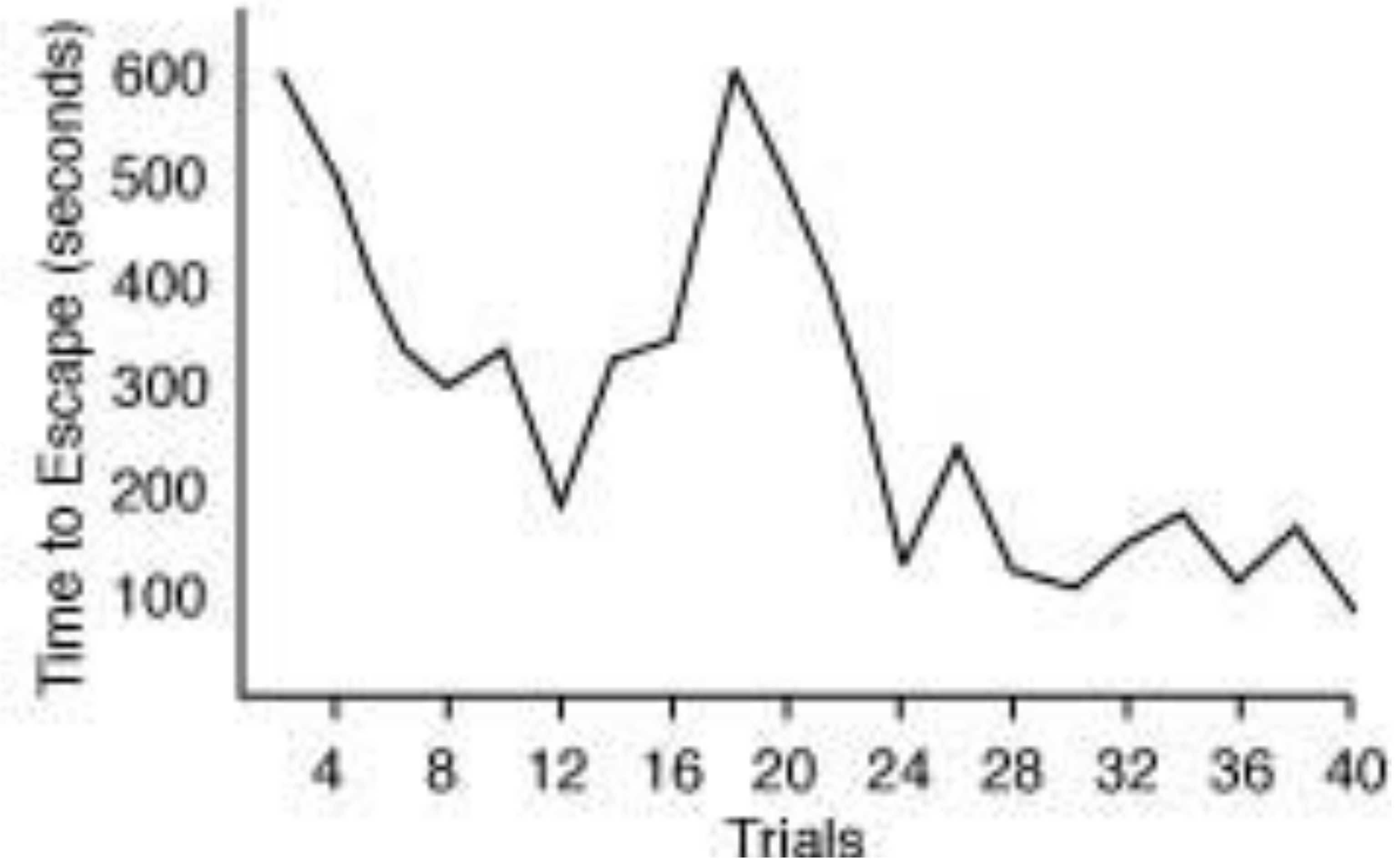
Cat



Puzzle Box

Time to escape



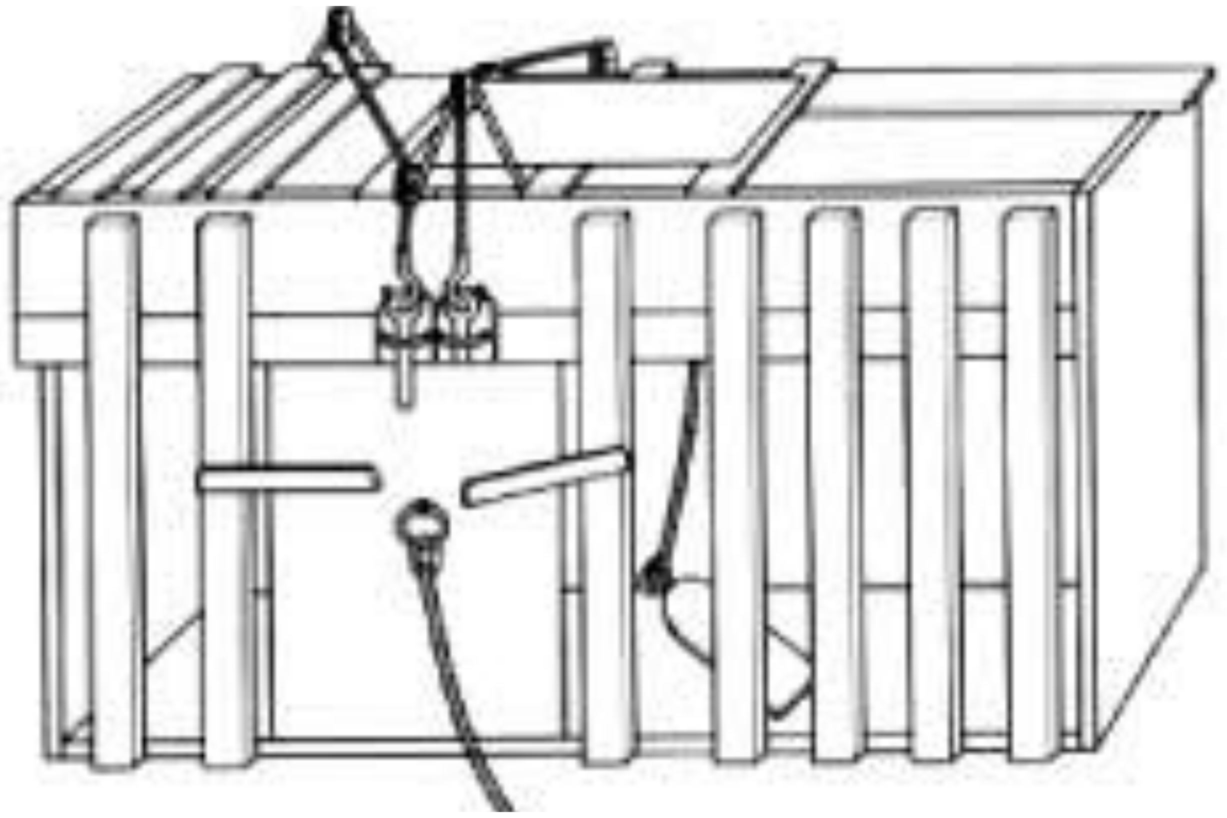


Law of Effect

"Actions associated with satisfaction are strengthened, while those associated with discomfort become weakened"

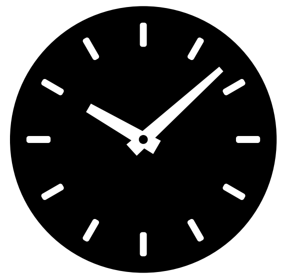
Thorndike's (1911) Law of Effect

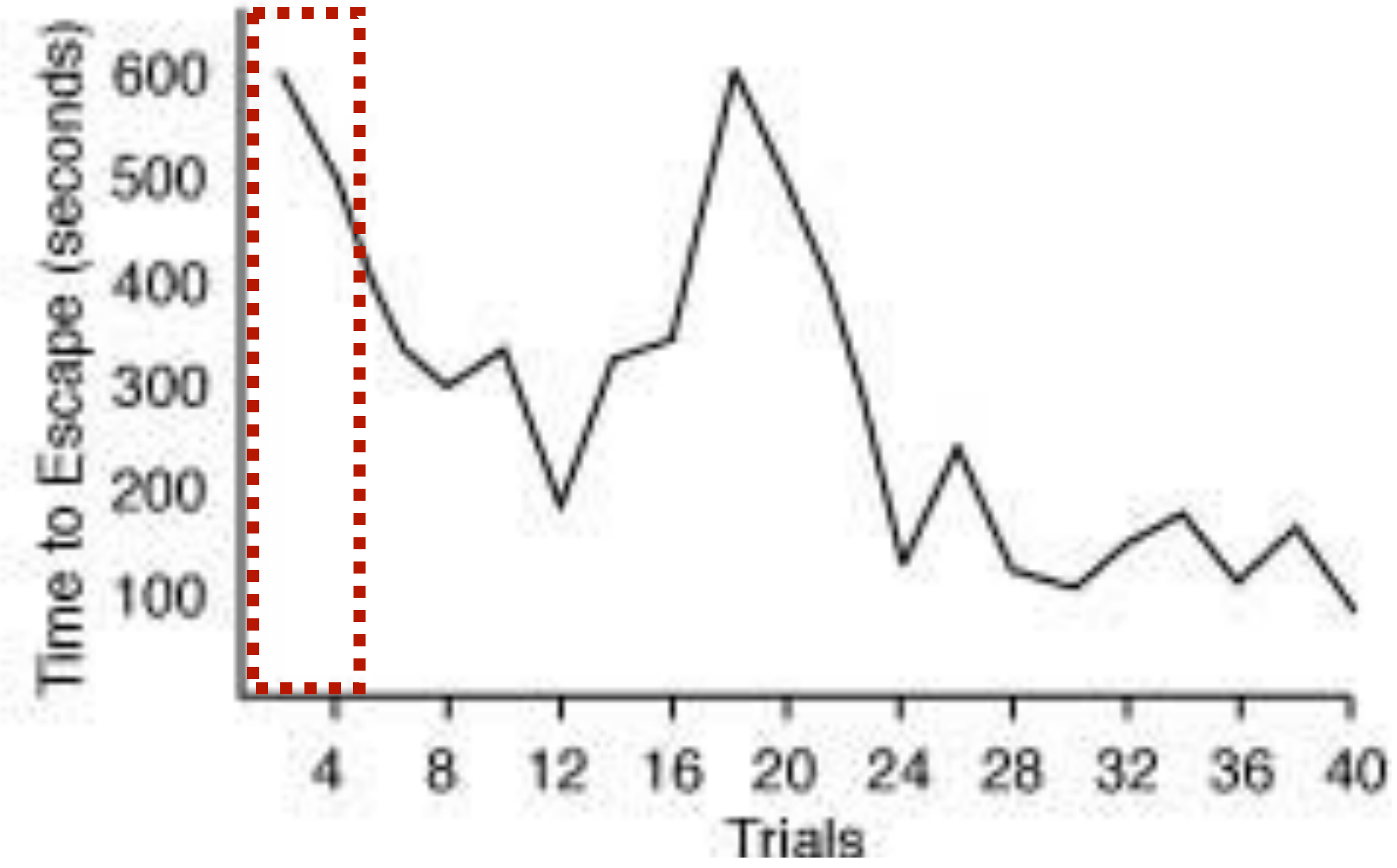
Cat



Puzzle Box

Time to escape



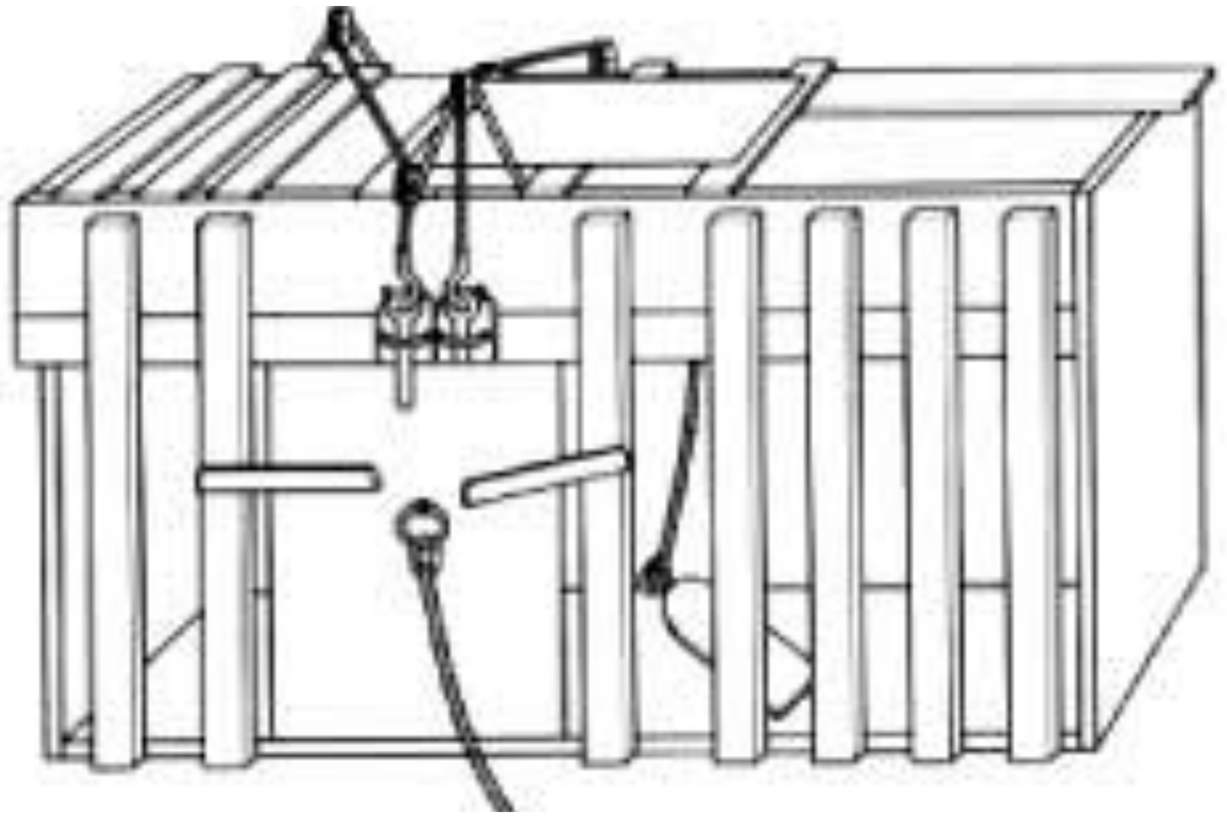


Law of Effect

"Actions associated with satisfaction are strengthened, while those associated with discomfort become weakened"

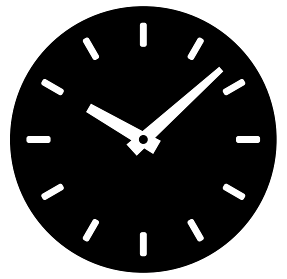
Thorndike's (1911) Law of Effect

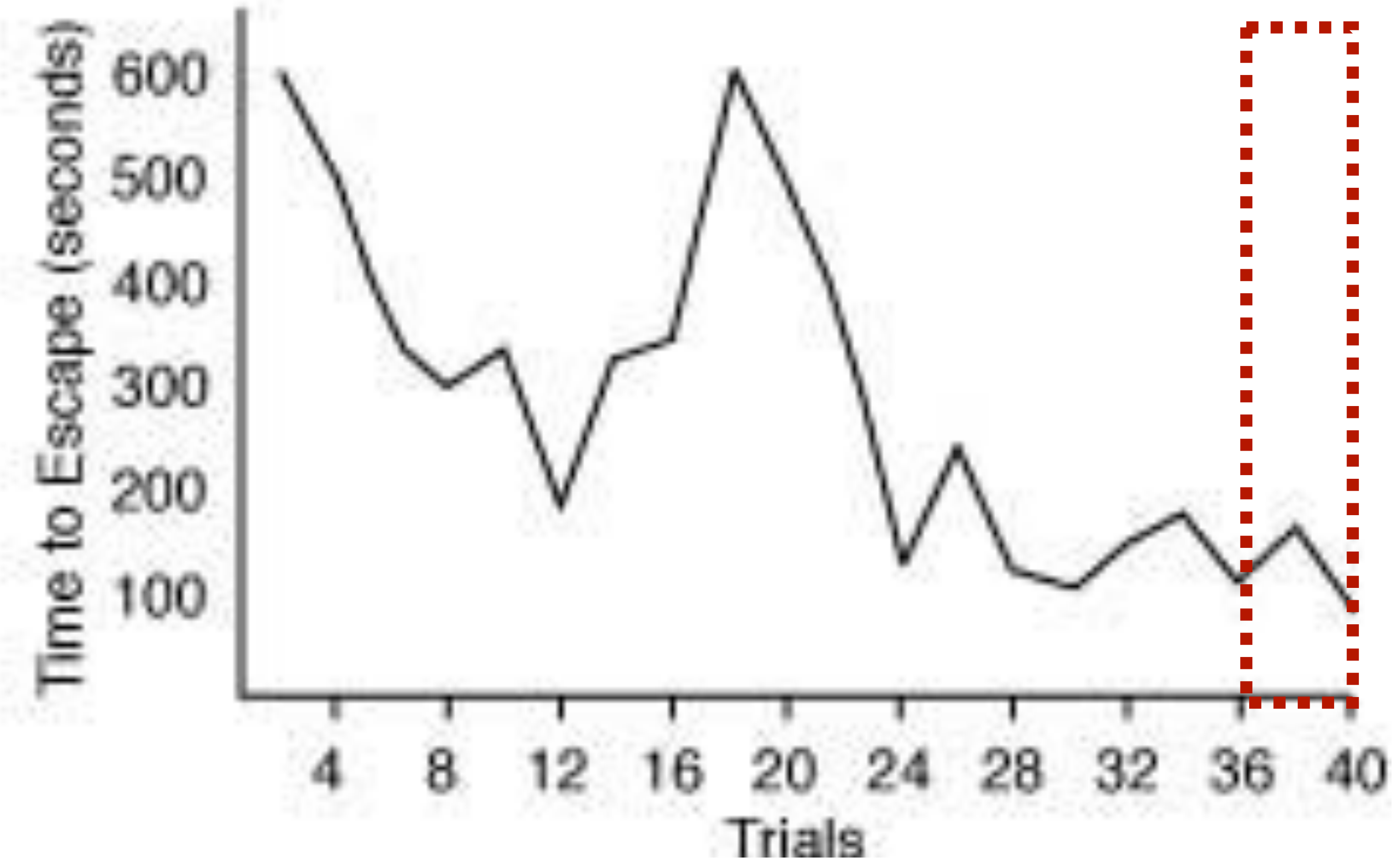
Cat

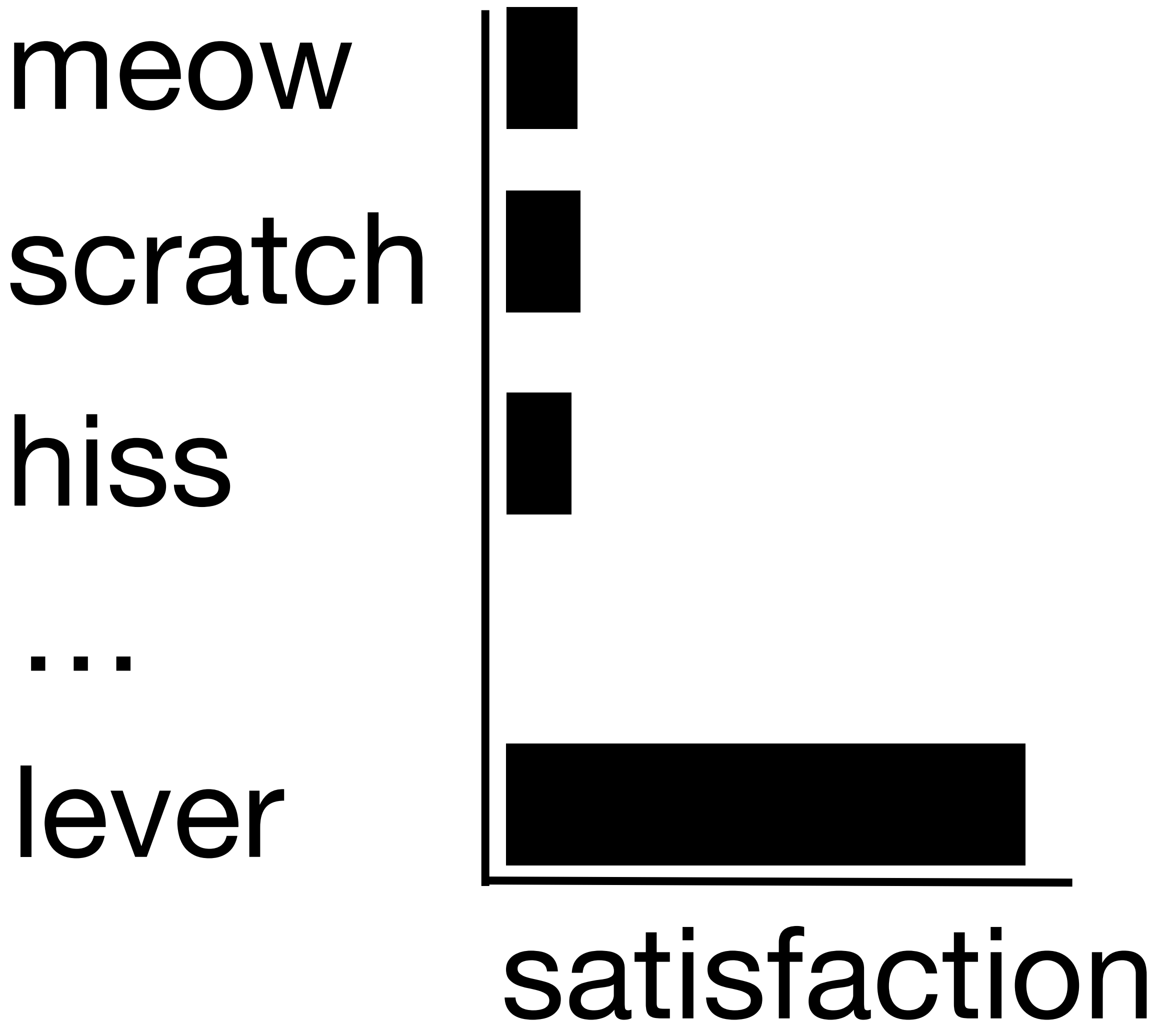


Puzzle Box

Time to escape







Law of Effect

"Actions associated with satisfaction are strengthened, while those associated with discomfort become weakened"

Learning as Trial and Error

What are the *benefits*? What are the *limitations*?

Learning as Trial and Error

What are the *benefits*? What are the *limitations*?

Benefits:

- Errors decrease over time
- Openess to trying new solutions
- Basis for all modern reinforcement learning (RL)

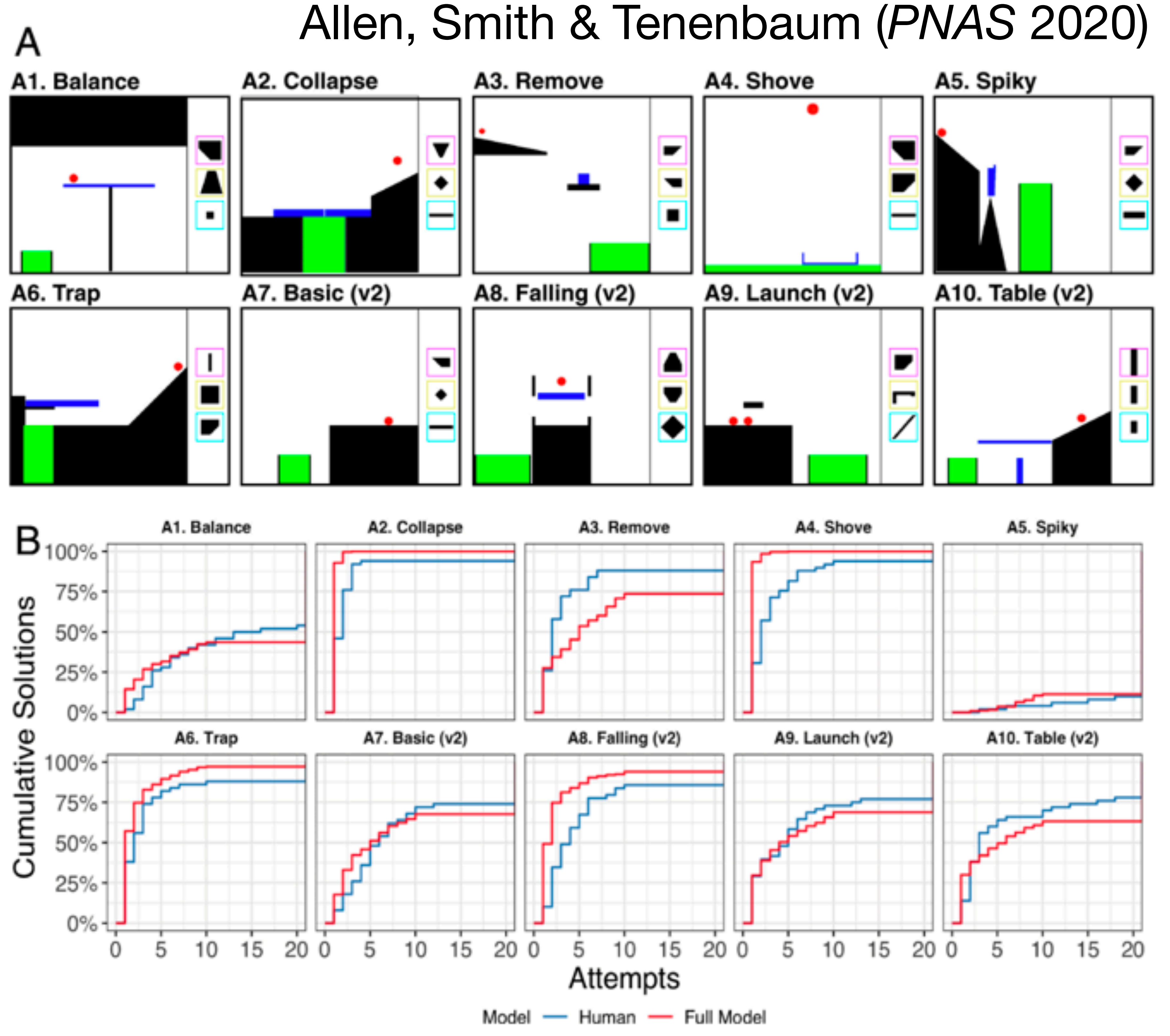
Learning as Trial and Error

What are the *benefits*? What are the *limitations*?

Benefits:

- Errors decrease over time
- Openess to trying new solutions
- Basis for all modern reinforcement learning (RL)

Allen, Smith & Tenenbaum (*PNAS* 2020)



Learning as Trial and Error

What are the *benefits*? What are the *limitations*?

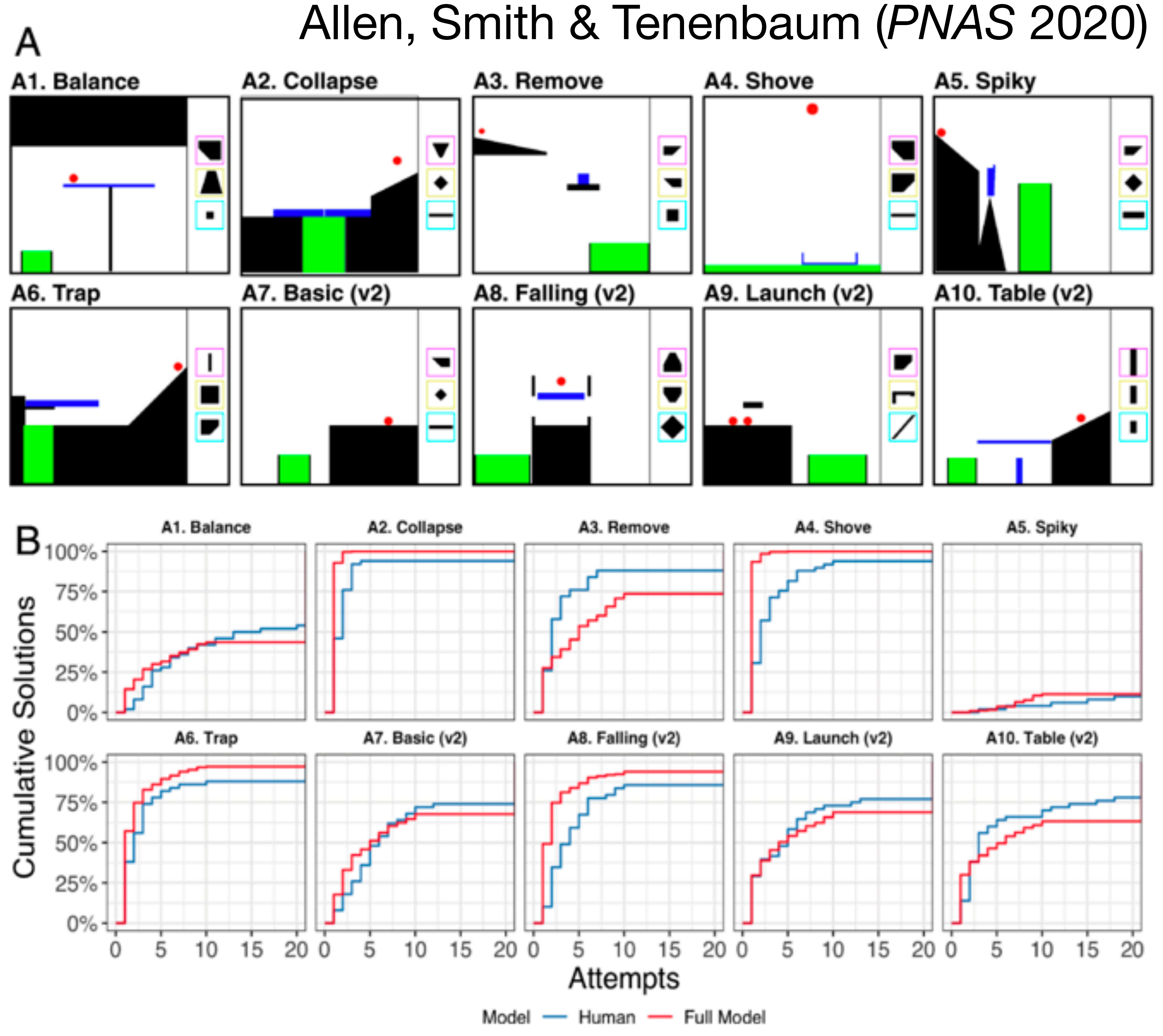
Benefits:

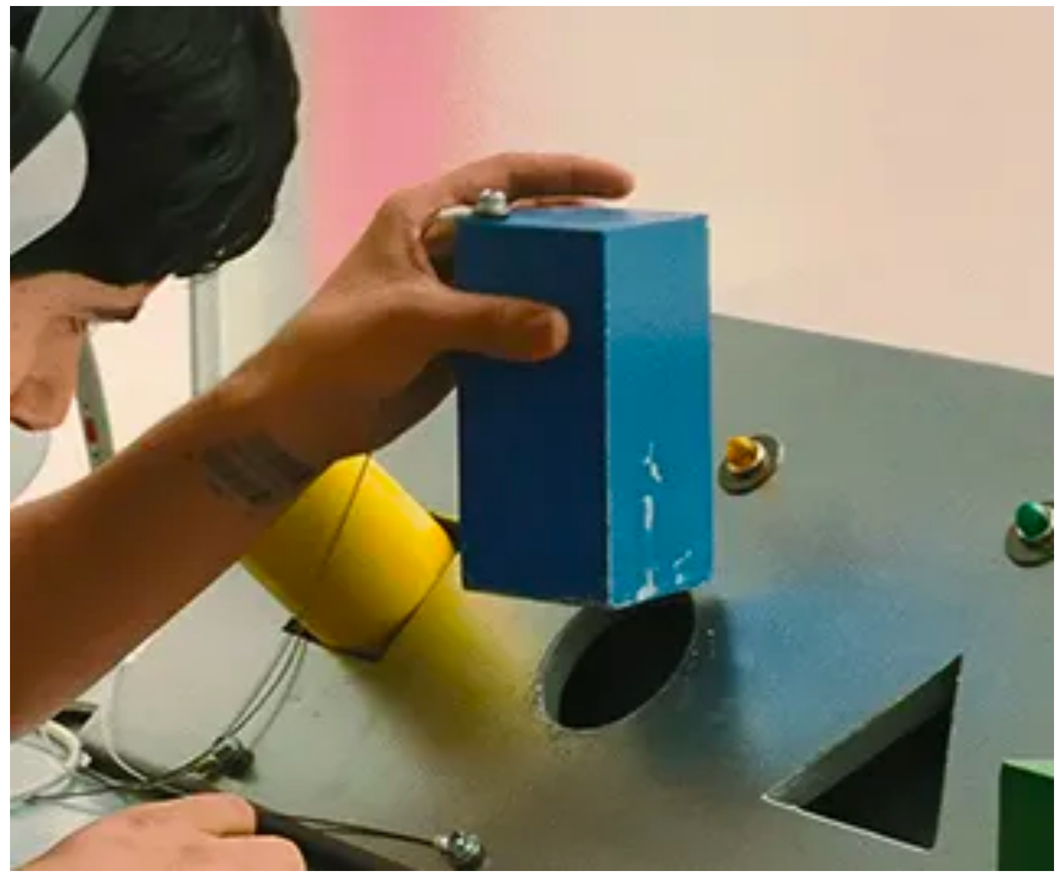
- Errors decrease over time
- Openess to trying new solutions
- Basis for all modern reinforcement learning (RL)

Limitations:

- Dangerous when some errors are fatal
- Lacks creativity and generalizastion of past solutions
- No formalism between behavior and outcome….

Allen, Smith & Tenenbaum (*PNAS* 2020)





Thorndike’s (1911) Law of Exercise

- In addition to the repeating successful actions, we also *repeat actions that we performed in the past*
- Learning as habit formation
    - e.g., morning routine, commute to university, studying/exercise routine, etc…
- Behavior is reinforced through frequent connections of stimulus and response

Thorndike's (1911) Law of Exercise

- In addition to the repeating successful actions, we also *repeat actions that we performed in the past*
- Learning as habit formation
  - e.g., morning routine, commute to university, studying/exercise routine, etc…
- Behavior is reinforced through frequent connections of stimulus and response

Law of Exercise

Any response to a stimulus will be strengthened proportional to how often it has been associated in the past

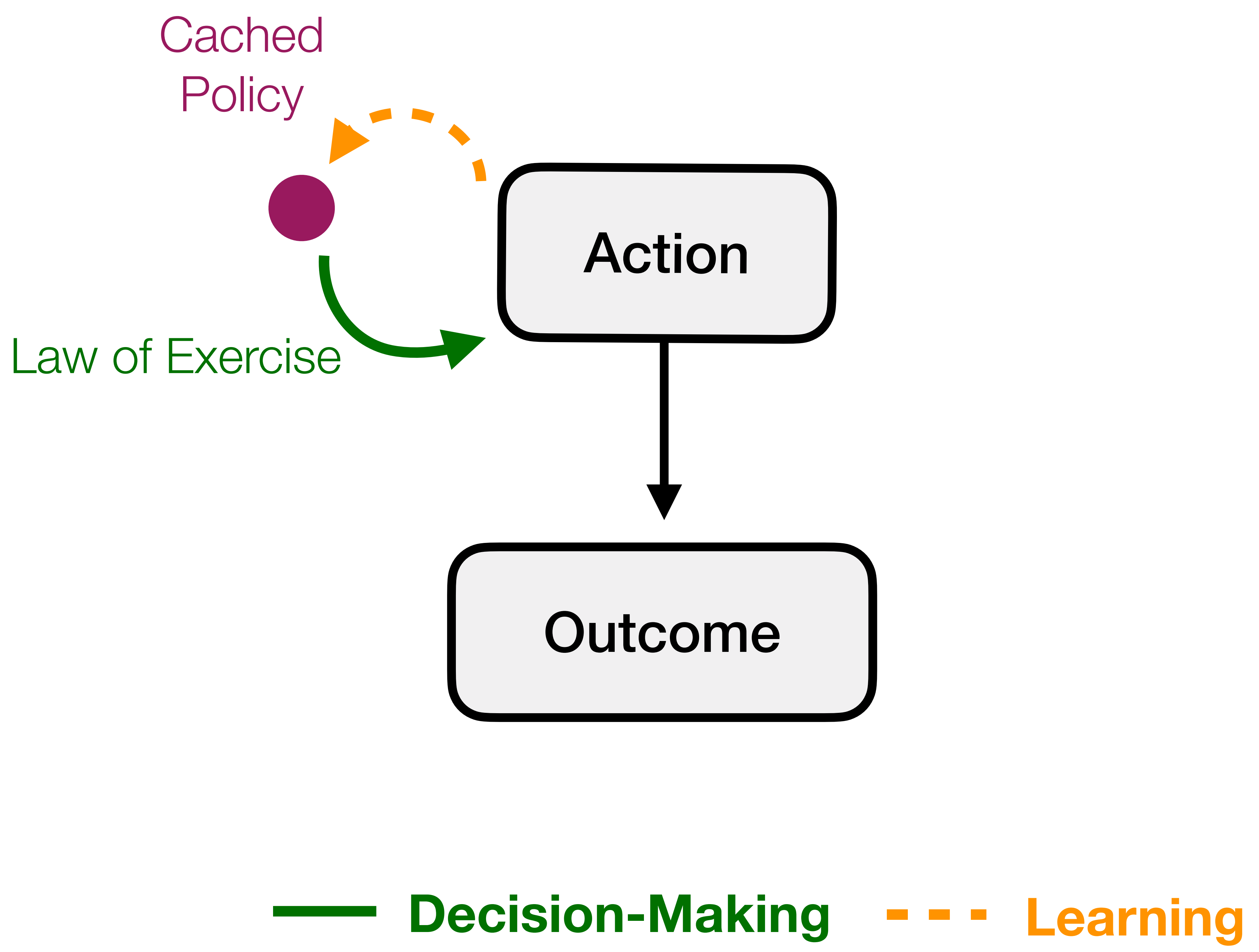
Key ideas: Two Pathways for Learning

Law of Effect & Law of Exercise

Key ideas: Two Pathways for Learning

Law of Effect & Law of Exercise

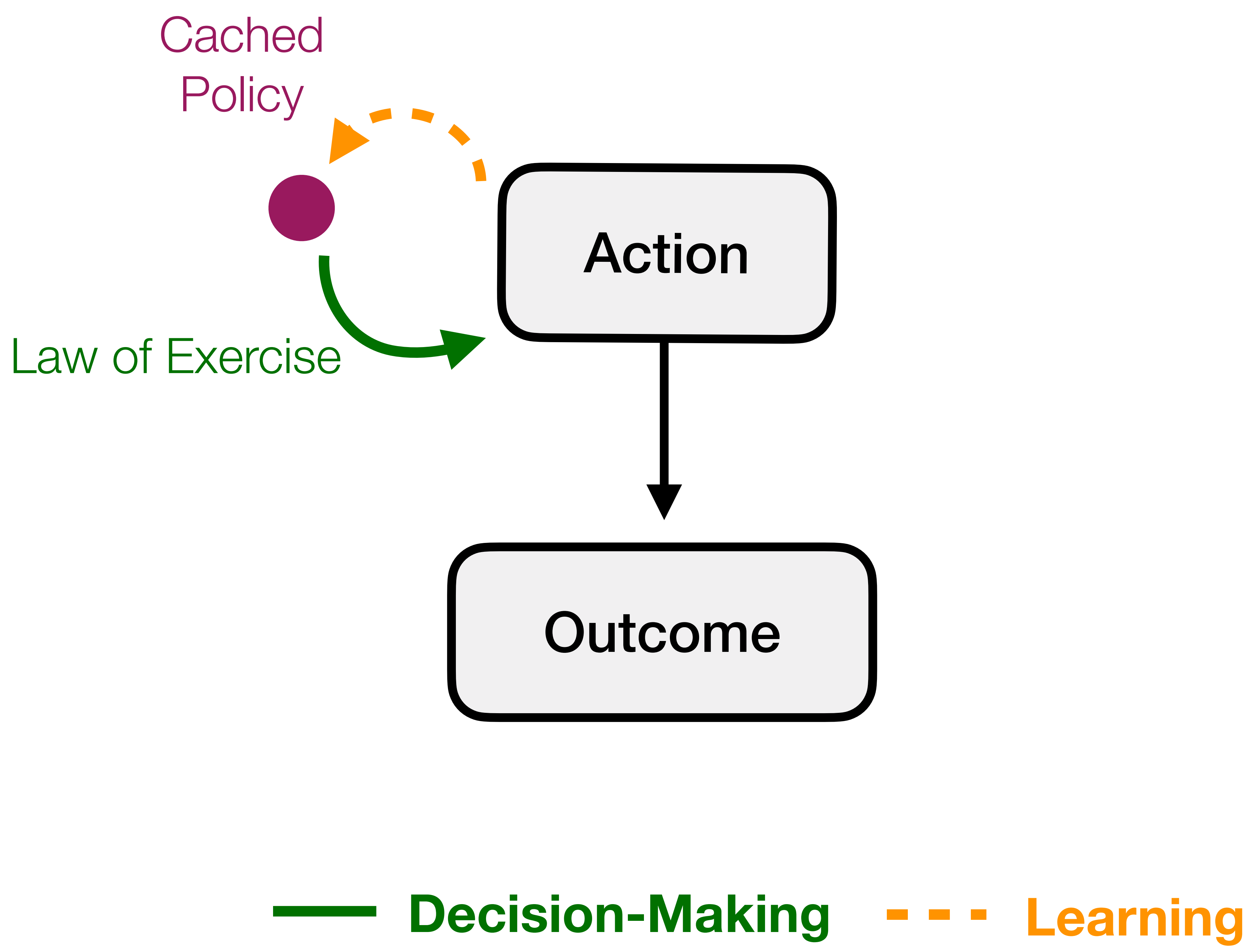
- **Law of Exercise**: Repeat actions performed in the past (regardless of outcome)



Key ideas: Two Pathways for Learning

Law of Effect & Law of Exercise

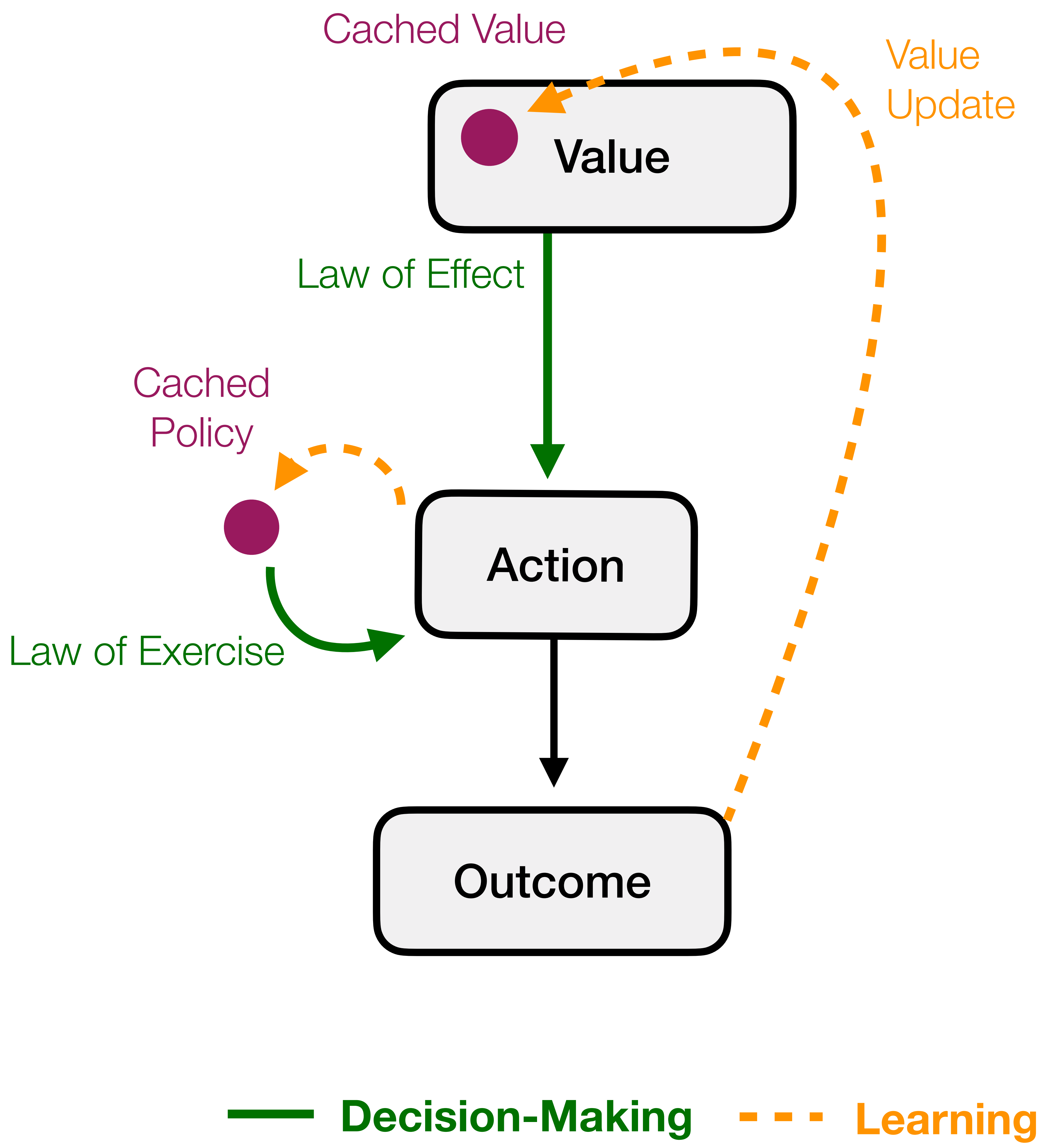
- **Law of Exercise**: Repeat actions performed in the past (regardless of outcome)
  - Learn a "**cached policy**"
    (Cushman & Morris, 2015; Daw et al., 2005; Gershman, 2020)



Key ideas: Two Pathways for Learning

Law of Effect & Law of Exercise

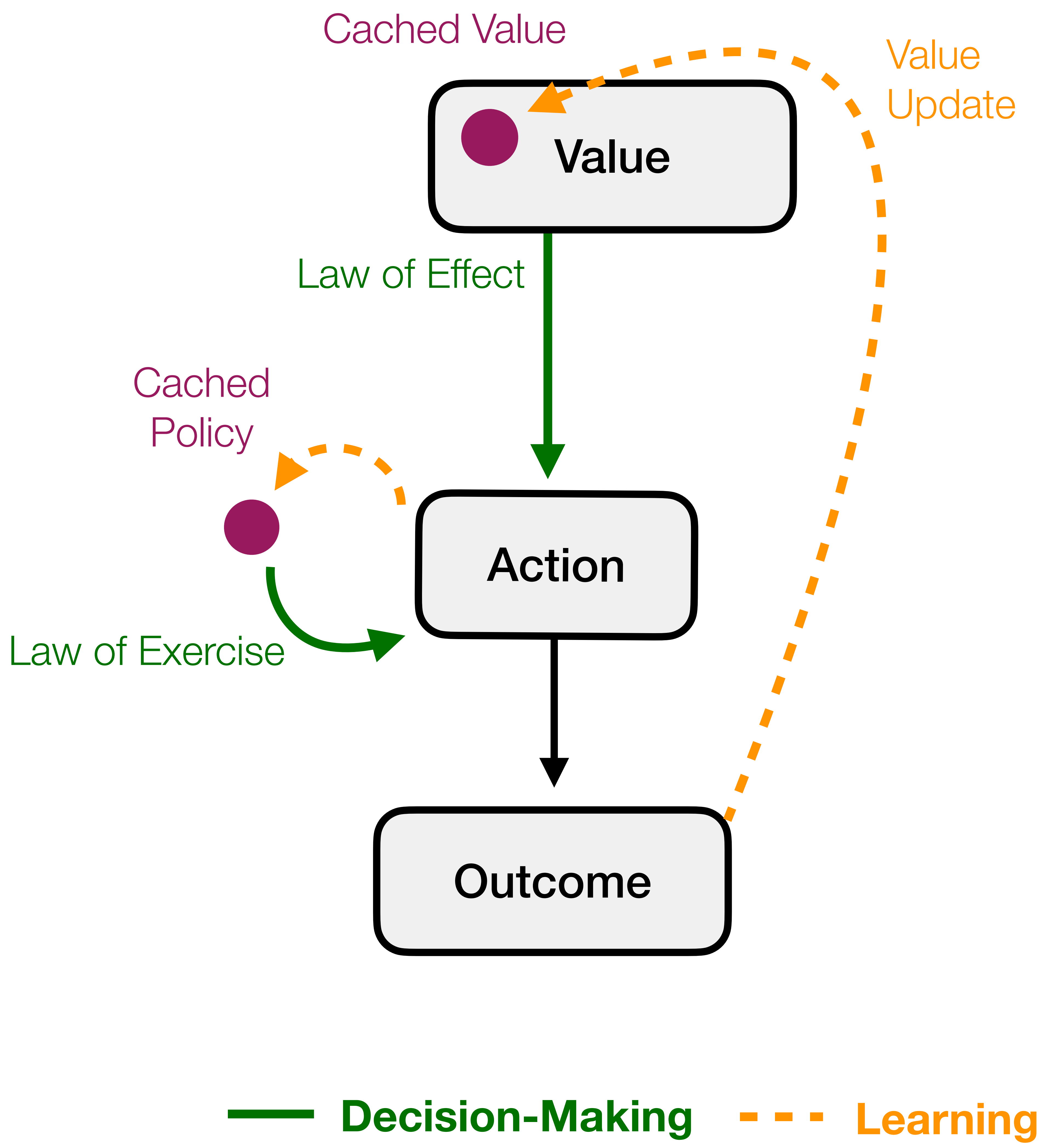
- **Law of Exercise**: Repeat actions performed in the past (regardless of outcome)
    - Learn a “**cached policy**” (Cushman & Morris, 2015; Daw et al., 2005; Gershman, 2020)
- **Law of Effect**: Choose actions on the basis of what has worked in the past



Key ideas: Two Pathways for Learning

Law of Effect & Law of Exercise

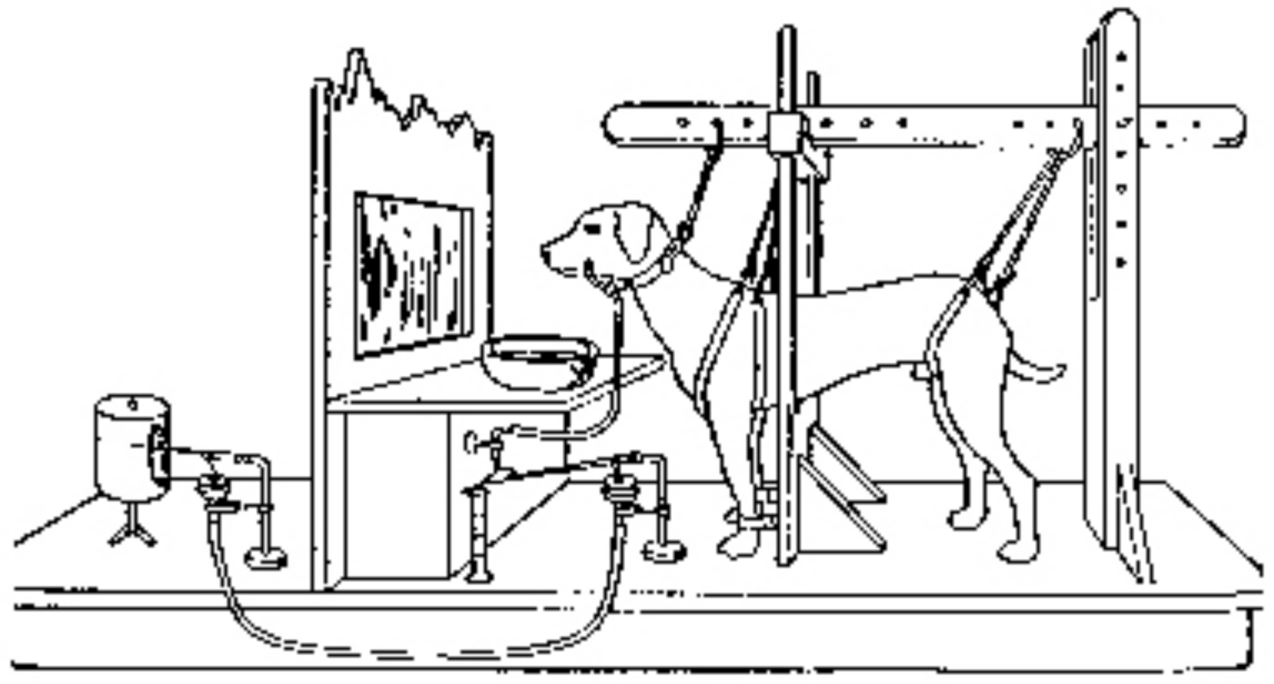
- **Law of Exercise**: Repeat actions performed in the past (regardless of outcome)
  - Learn a "**cached policy**"
    (Cushman & Morris, 2015; Daw et al., 2005; Gershman, 2020)
- **Law of Effect**: Choose actions on the basis of what has worked in the past
  - Learn a "**cached value**" that can be used to select actions
    (Botvinick & Weinstein, 2014; Keramati et al., 2016; Maisto et al., 2019)

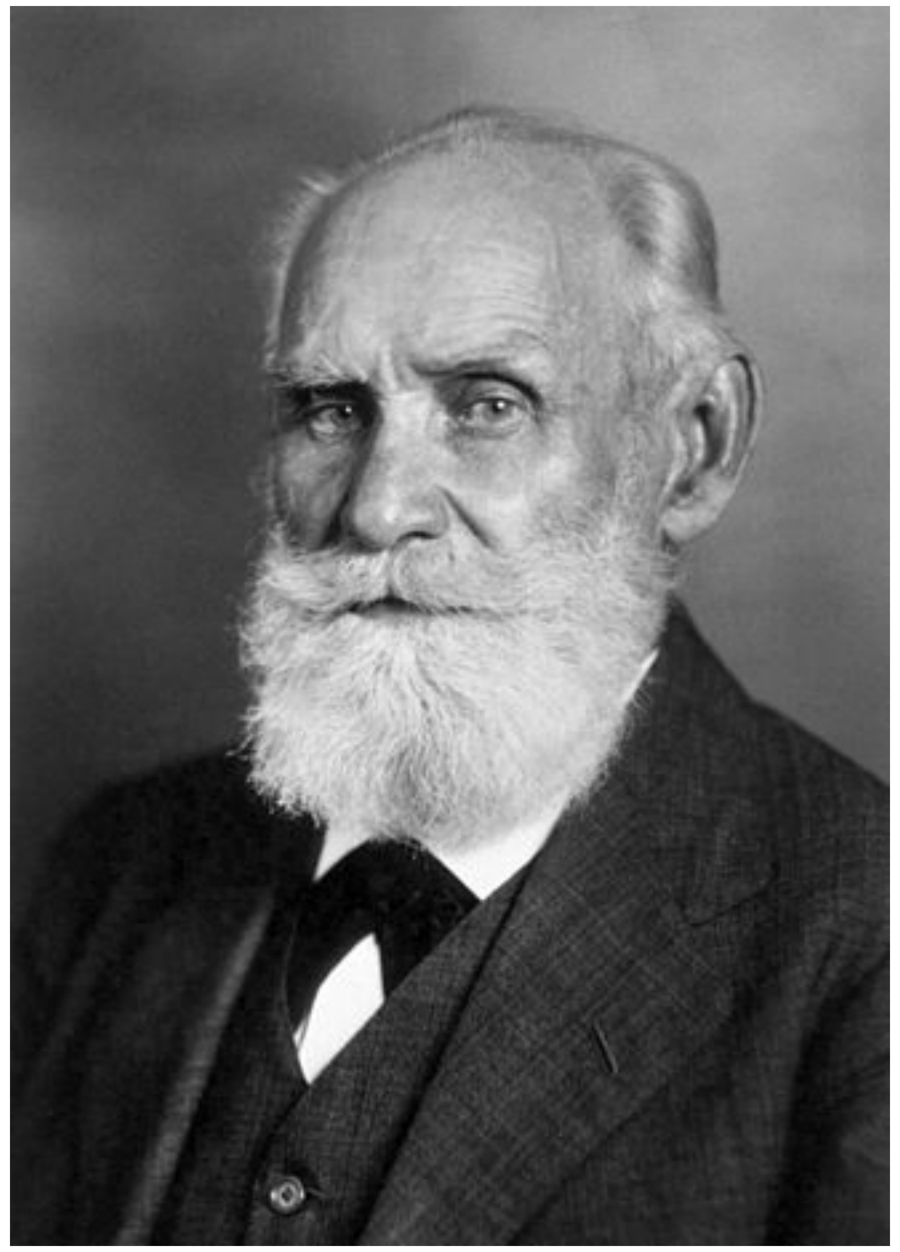


Pavlov's Dog: Classical conditioning

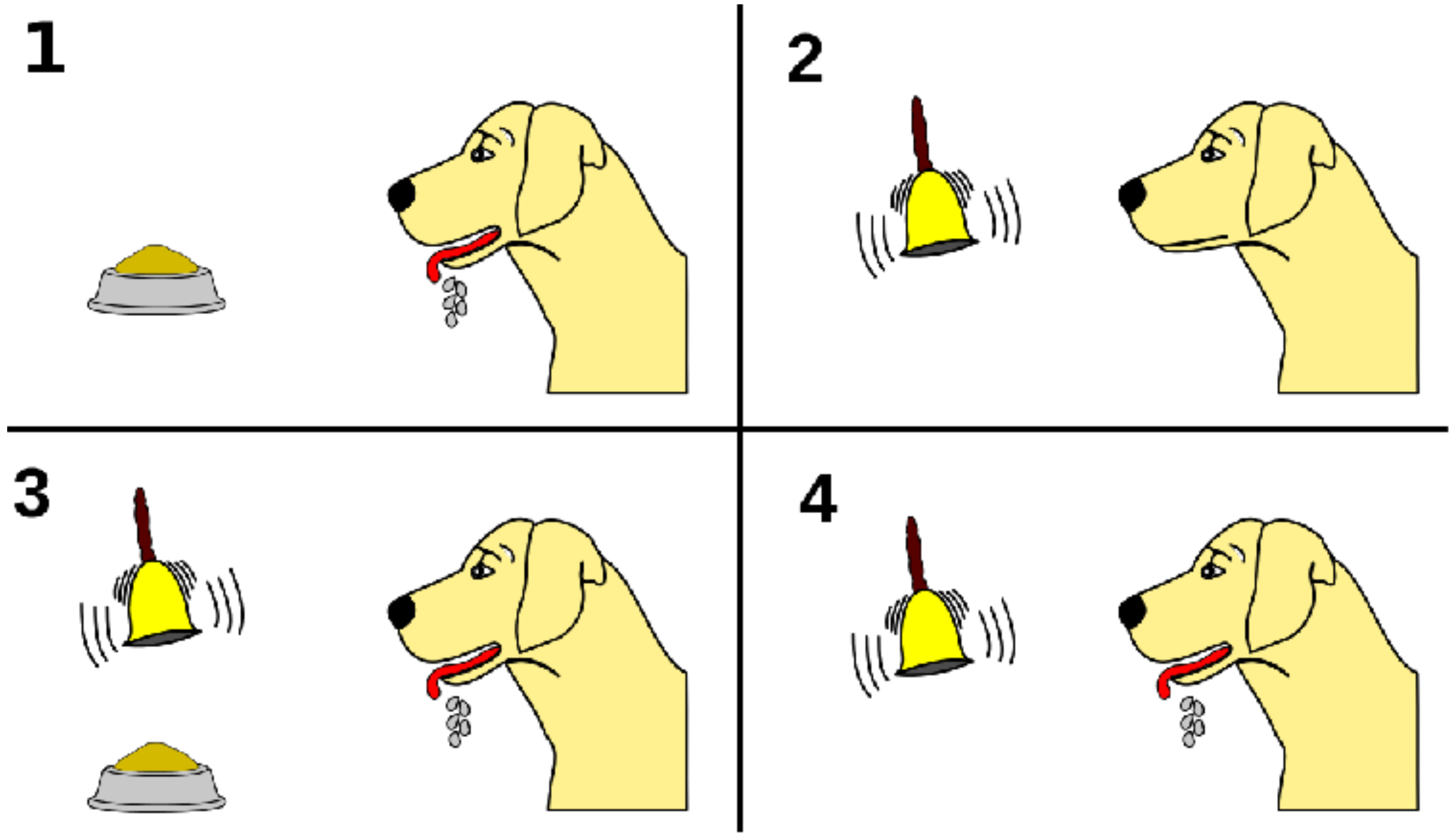
- Pavlov (1849-1936) approached learning from a different angle, focusing on automatic responses

1. The dog naturally salivates when presented with food (unconditioned stimulus; US)
2. No initial response to a bell (conditioned stimulus; CS)
3. When the dog is trained to associate a bell with the delivery of food…
4. … it learns to anticipate food when a bell rings and begins to salivate



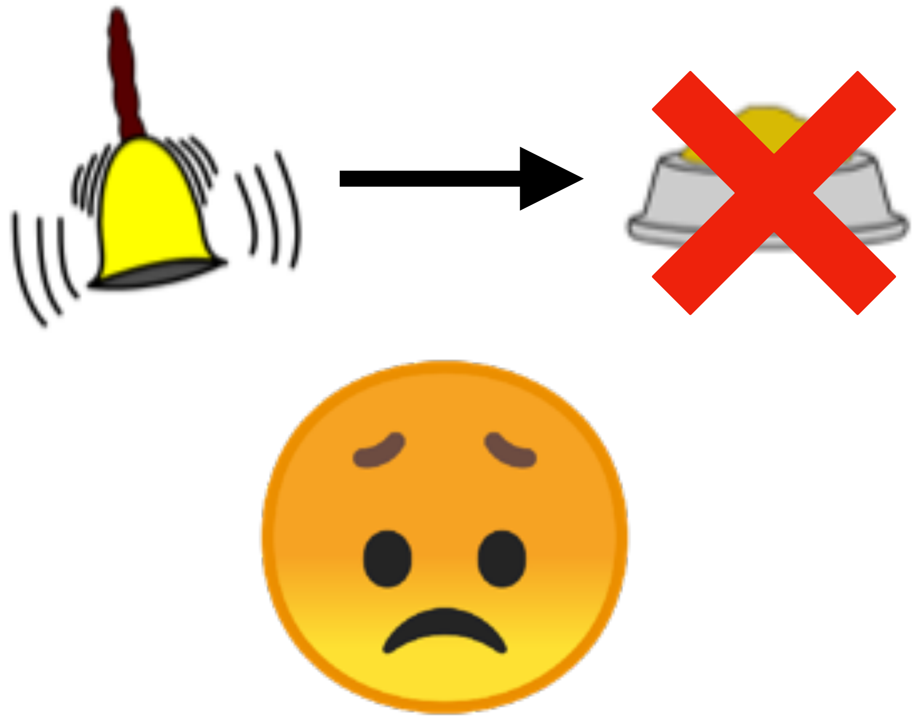


Ivan Pavlov



Key ideas: Classical conditioning

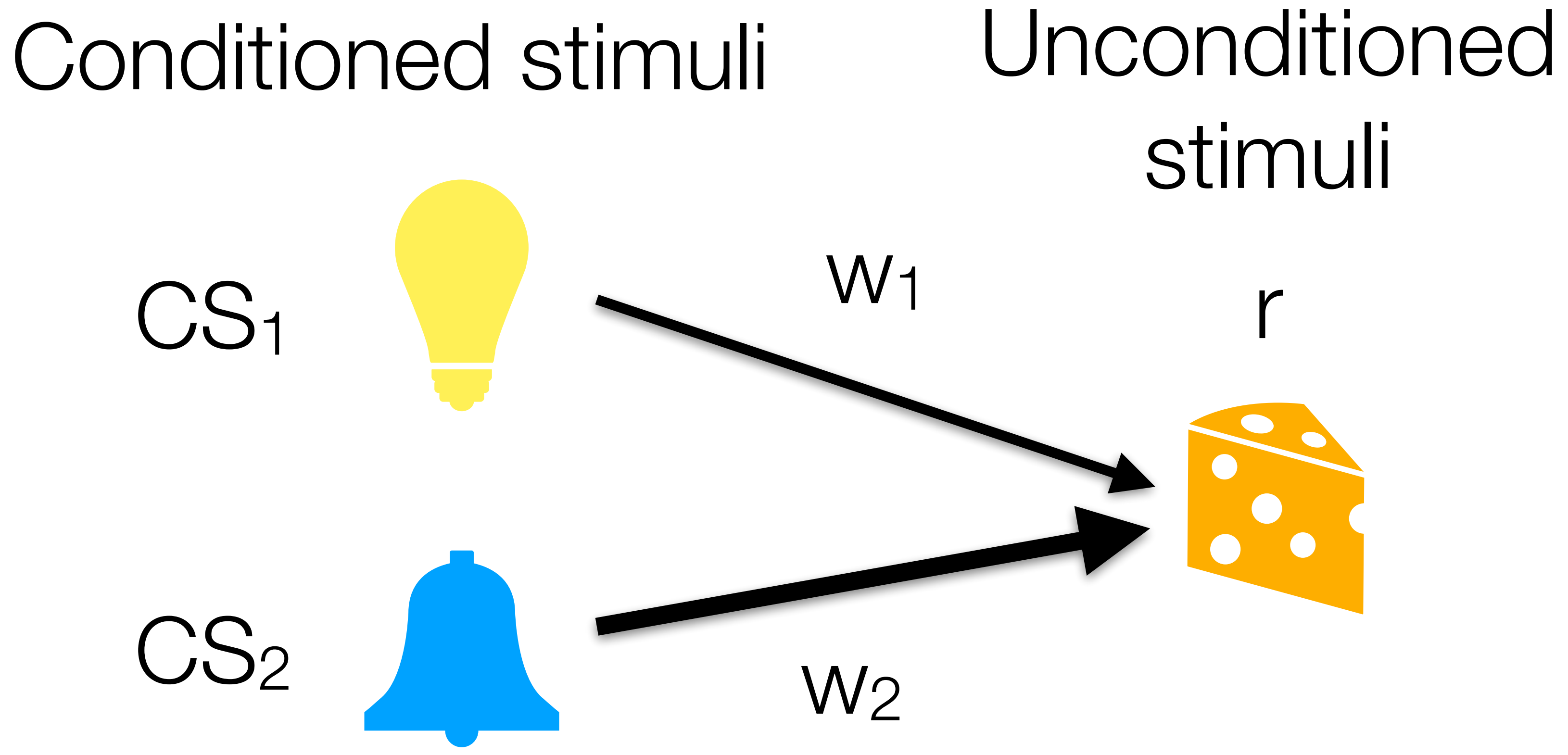
Pavlovian responses are driven by predictions about expected outcomes



Learning is driven by reward predictions and (as we will see) shaped by prediction error

Cues compete for shared credit in predicting reward outcomes

Rescorla-Wagner



Rescorla-Wagner model
(Bush & Mosteller, 1951; Rescorla & Wagner, 1972)

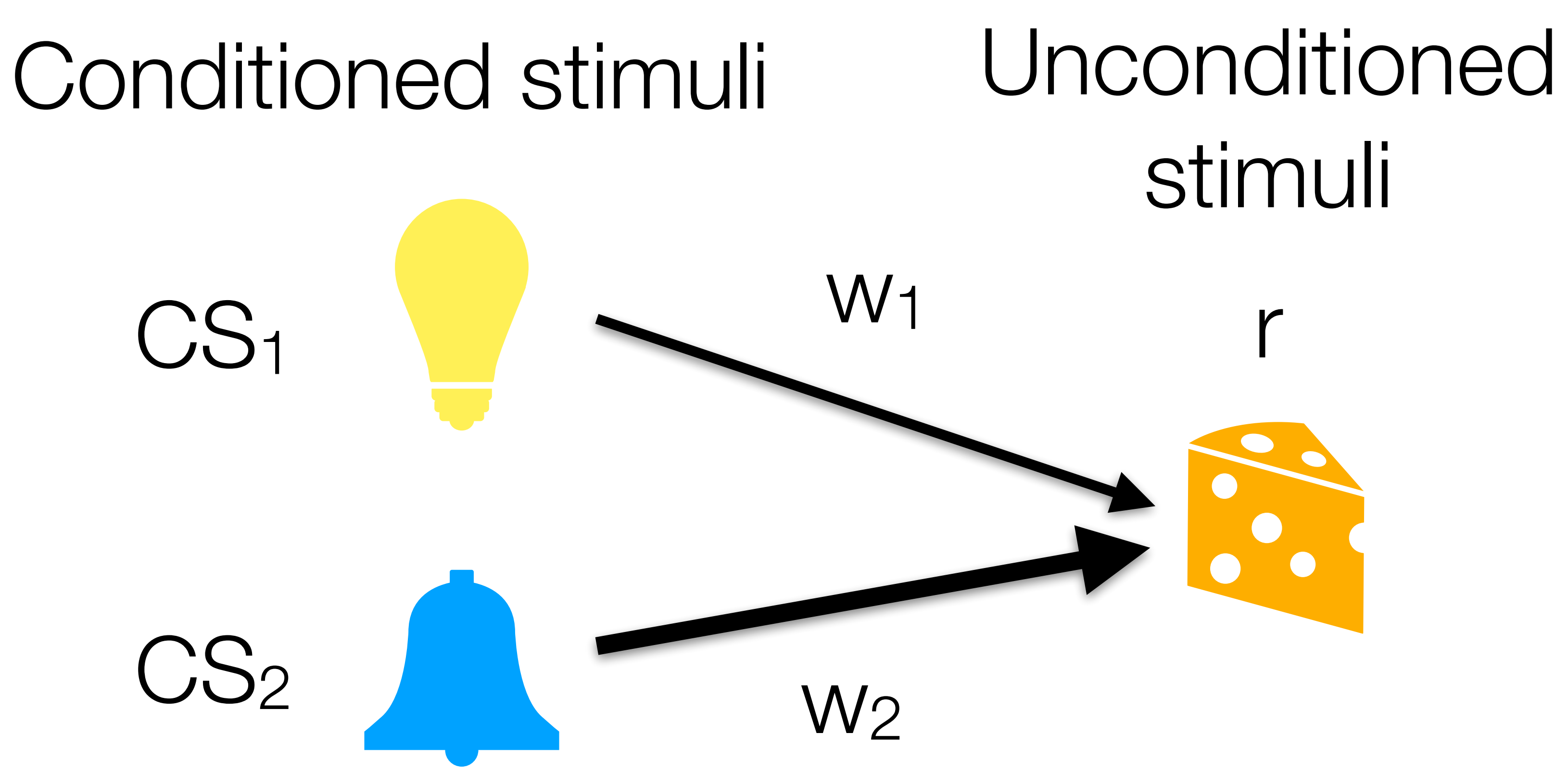
Reward prediction

$$\hat{r}_t = \sum_i \mathrm{CS}_i^t w_i$$

Weight update

$$w_i \leftarrow w_i + \eta(r_t - \hat{r}_t)$$

Rescorla-Wagner



Rescorla-Wagner model

(Bush & Mosteller, 1951; Rescorla & Wagner, 1972)

Reward prediction

$$\hat{r}_t = \sum_i \mathrm{CS}_i^t w_i$$

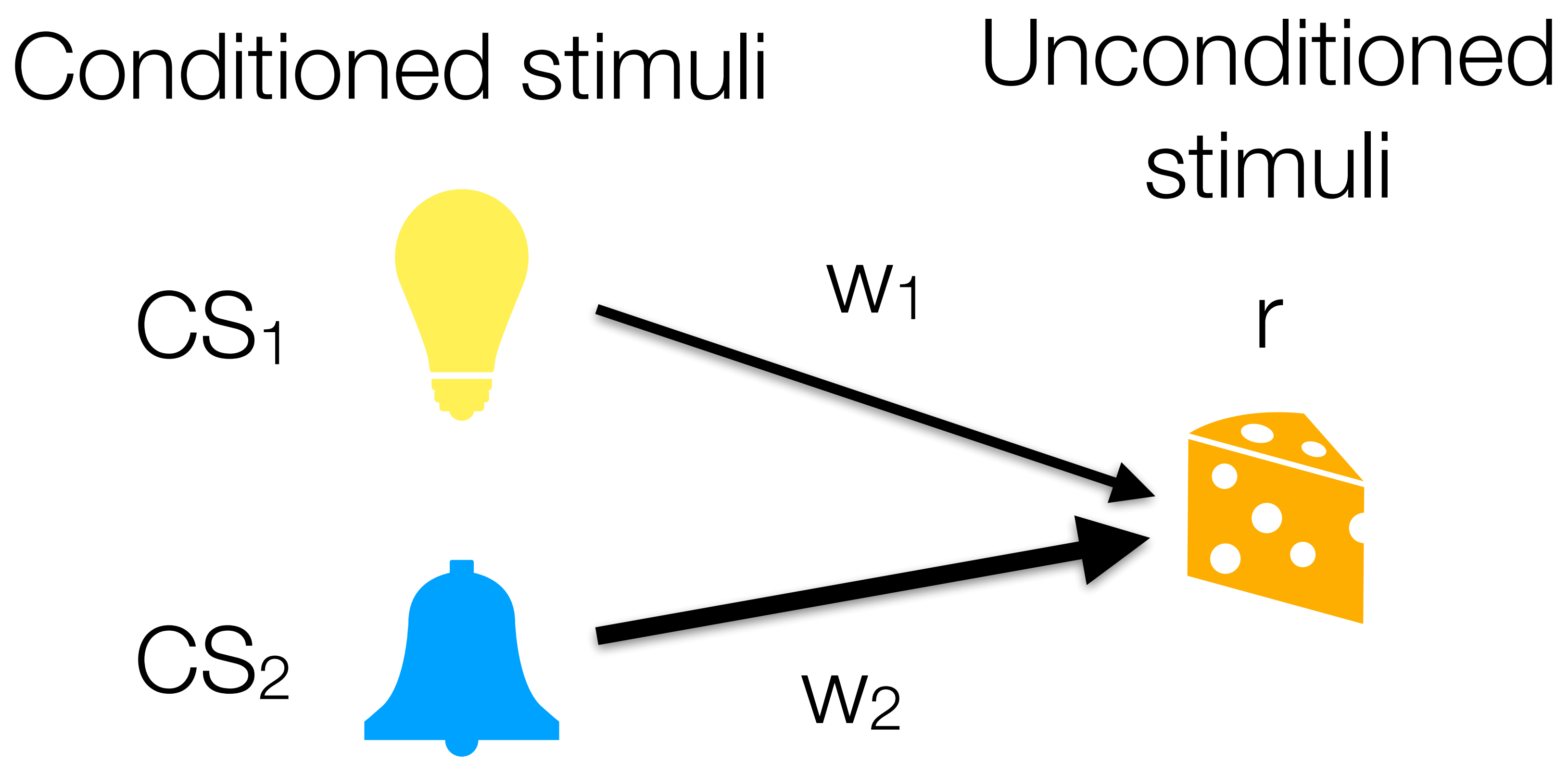
Weight update

$$w_i \leftarrow w_i + \eta(r_t - \hat{r}_t)$$

RW Model

- [left] Reward expectations are the sum of CS stimuli x weights
- [right] Weights are updated via the delta-rule

Rescorla-Wagner



Rescorla-Wagner model

(Bush & Mosteller, 1951; Rescorla & Wagner, 1972)

Reward prediction

$$\hat{r}_t = \sum_i \mathrm{CS}_i^t w_i$$

- $\hat{r}_t$: Reward expectation
- CS_i^t: CS i on trial t
- w_i: Associative strength or weight

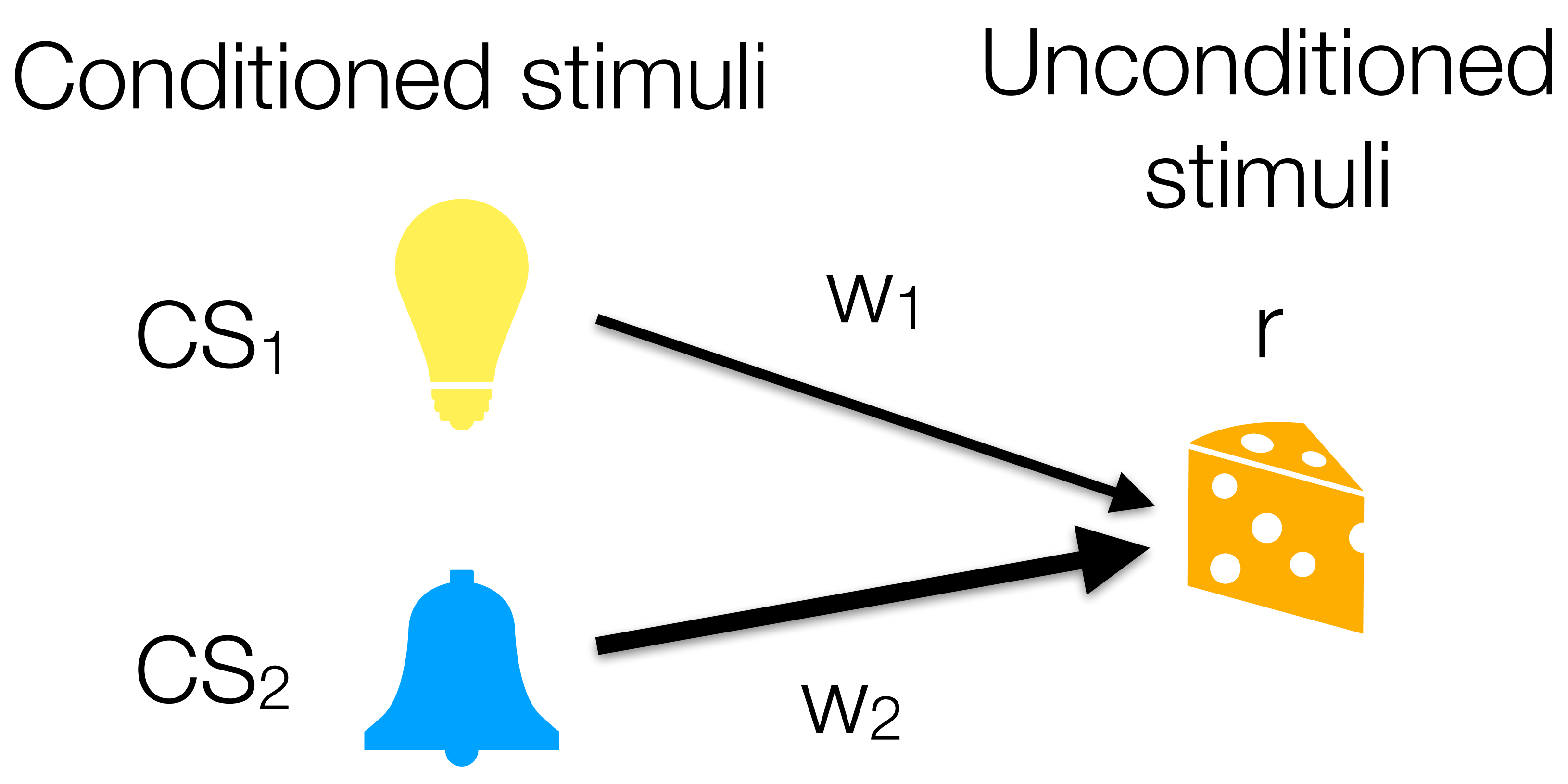
Weight update

$$w_i \leftarrow w_i + \eta(r_t - \hat{r}_t)$$

RW Model

- [left] Reward expectations are the sum of CS stimuli x weights
- [right] Weights are updated via the delta-rule

Rescorla-Wagner



Rescorla-Wagner model

(Bush & Mosteller, 1951; Rescorla & Wagner, 1972)

Reward prediction

$$\hat{r}_t = \sum_i \mathrm{CS}_i^t w_i$$

- $\hat{r}_t$: Reward expectation
- CS_i^t: CS i on trial t
- w_i: Associative strength or weight

Weight update

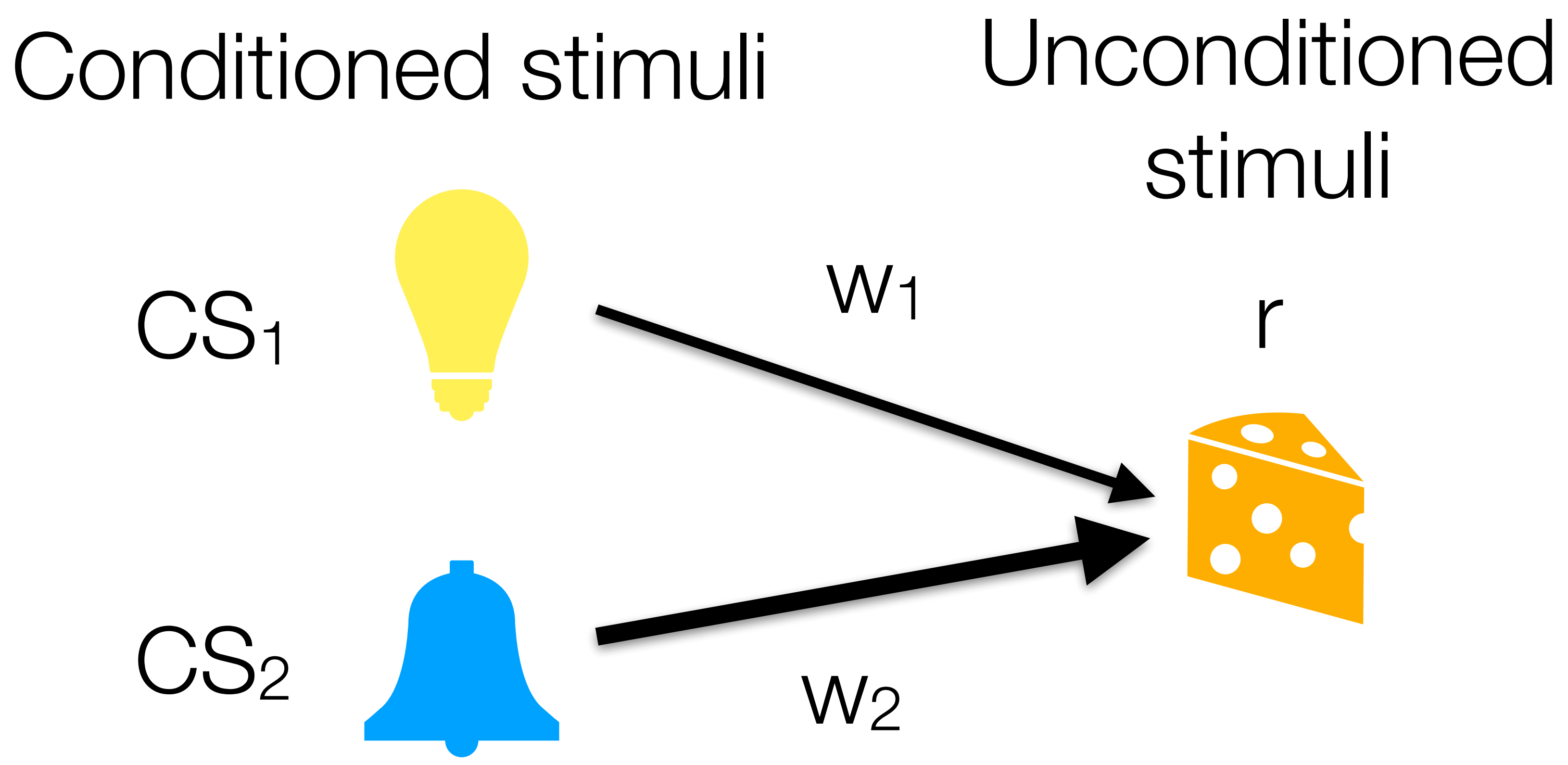
$$w_i \leftarrow w_i + \eta(r_t - \hat{r}_t)$$

- η: Learning rate
- r_t: Observed outcome
- $\hat{r}_t$: Predicted outcome

RW Model

- [left] Reward expectations are the sum of CS stimuli x weights
- [right] Weights are updated via the delta-rule

Rescorla-Wagner



Rescorla-Wagner model

(Bush & Mosteller, 1951; Rescorla & Wagner, 1972)

Reward prediction

$$\hat{r}_t = \sum_i \mathrm{CS}_i^t w_i$$

- $\hat{r}_t$ ↑ Reward expectation
- CS_i^t ↑ CS i on trial t
- w_i ↖ Associative strength or weight

Weight update

$$w_i \leftarrow w_i + \eta(r_t - \hat{r}_t)$$

- η ↗ Learning rate
- r_t ↑ Observed outcome
- $\hat{r}_t$ ↑ Predicted outcome

Observed outcome − Predicted outcome = δ : Reward prediction error (RPE)

RW Model

- [left] Reward expectations are the sum of CS stimuli x weights
- [right] Weights are updated via the delta-rule

The delta-rule of learning:

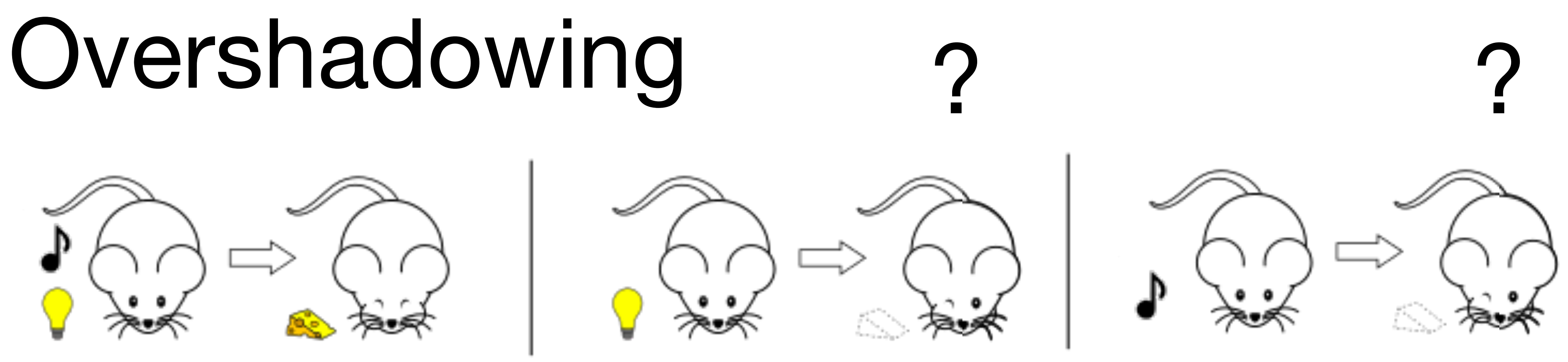
- Learning occurs only when events violate expectations ($\delta \neq 0$)
- The magnitude of the error corresponds to how much we update our beliefs

Implications: Cue competition

If multiple stimuli cues predict an outcome, they will share credit

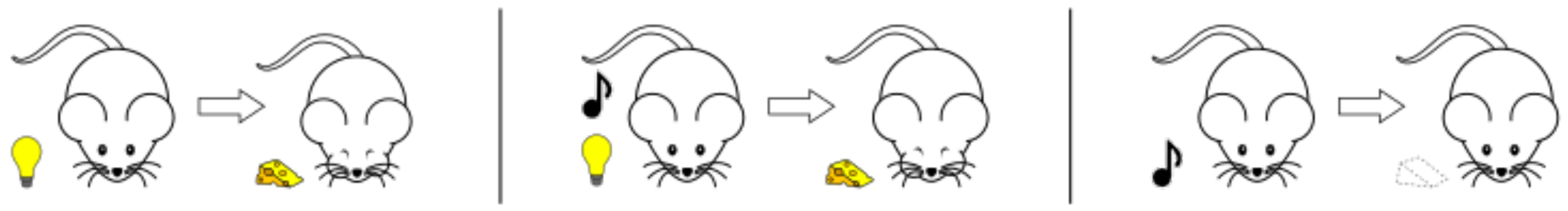
Overshadowing:

- If sound and light are both associated with reward, then presenting individual cues will result in weaker responses



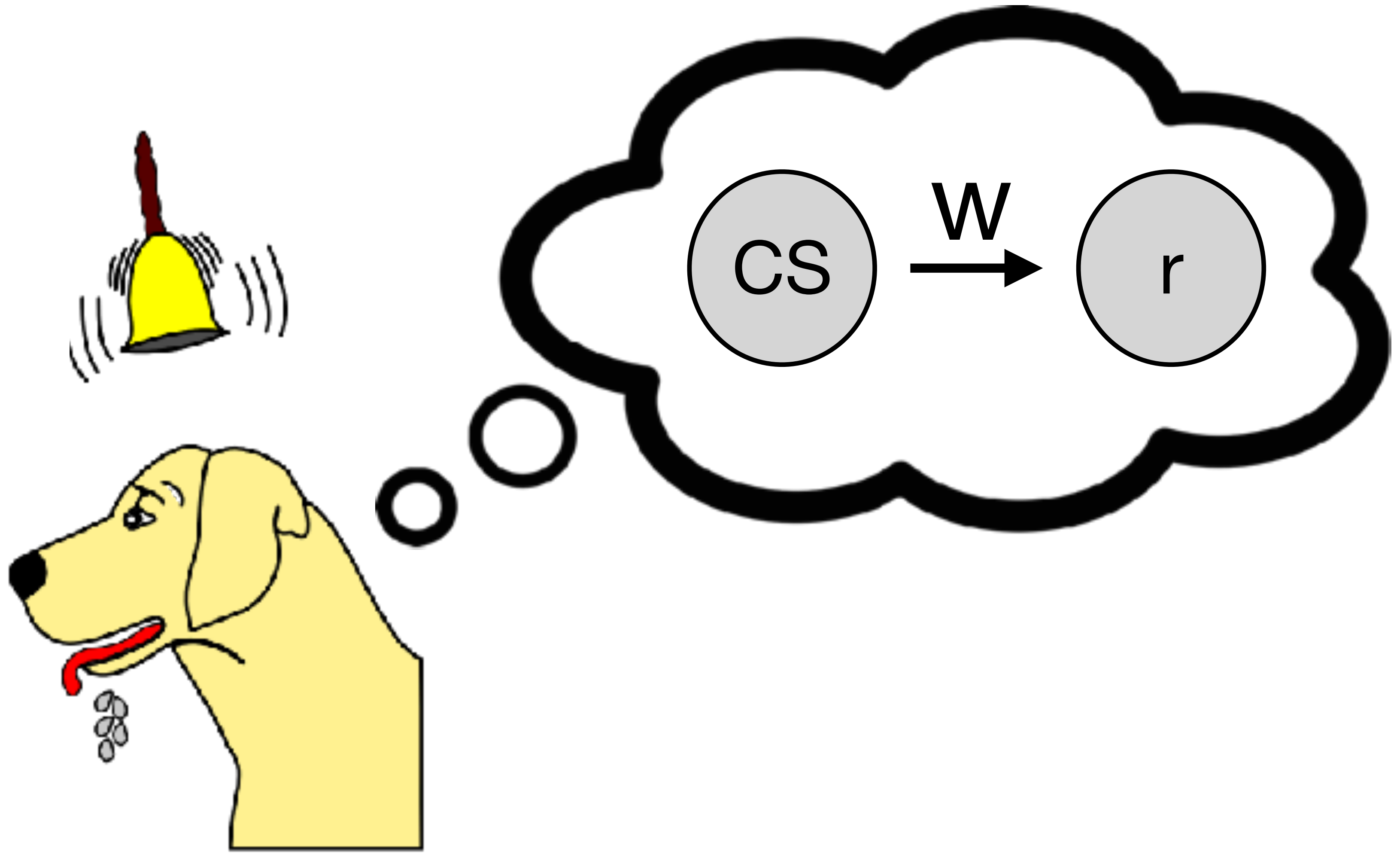
Blocking

- If light is first associated with reward, and then later both light and sound, there will be less associating of sound with reward than if sound were conditioned alone



Reward learning as refining an internal representation of the world

- Internal hypotheses about how sensory data $\mathscr{D}$ were generated
- The parameters w are unknown and must be estimated to maximize the likelihood of the data $P(\mathscr{D} \mid w)$
  - This is known as maximum likelihood estimation (MLE):
    $\hat{w} = \arg\max_{w} P(\mathscr{D} \mid w)$
- Under certain assumptions[1], RW implements a MLE through gradient descent
- *Thus, RW learning is similar to how neural networks learn*



not on the exam

Loss function

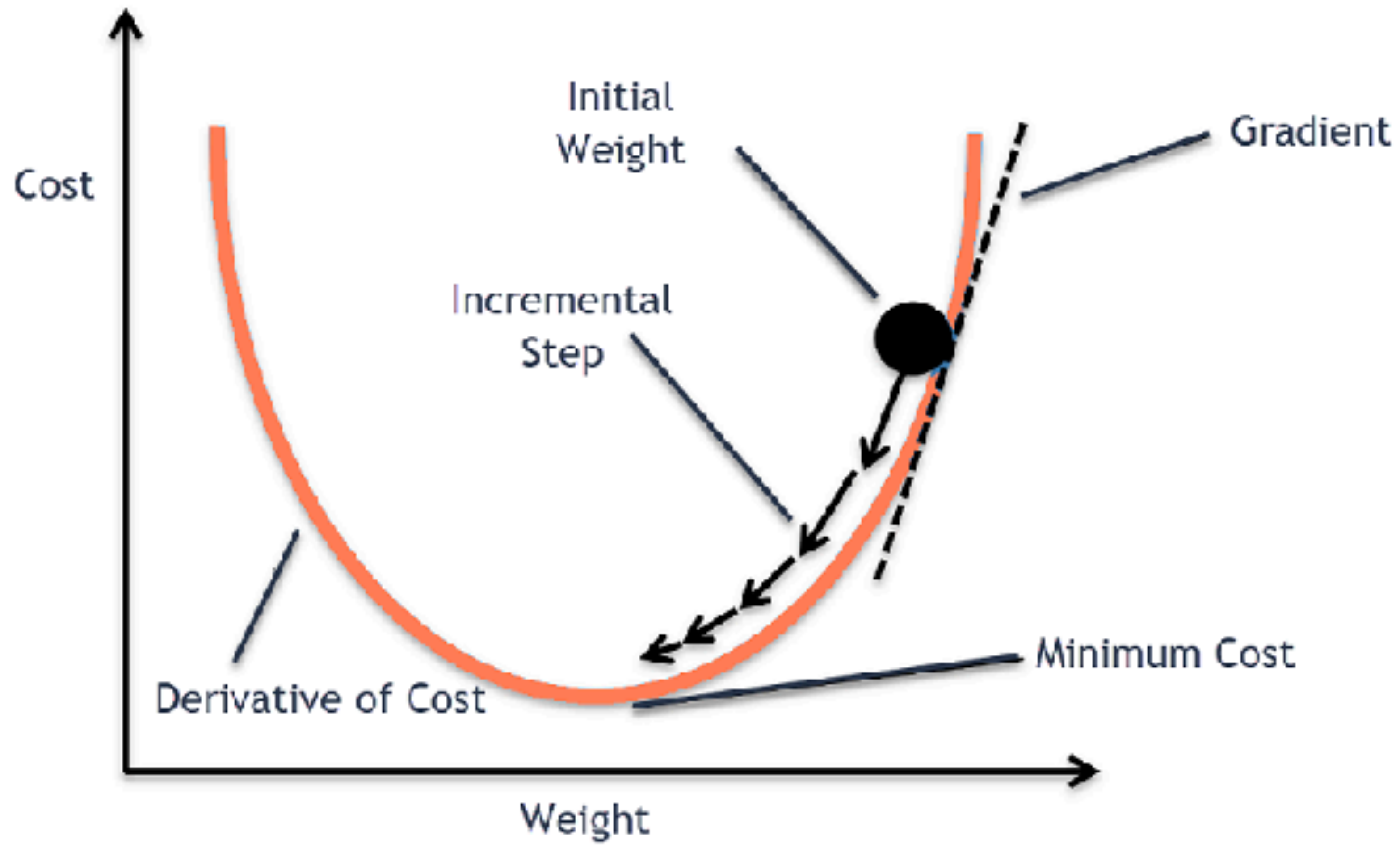
$$\mathscr{L}(w) = -\log P(\mathscr{D} \mid w)$$

Gradient update

$$\Delta \hat{w}_i \propto -\nabla_{w_i} \mathscr{L}(w) = \text{CS}_i(r - \hat{r})$$

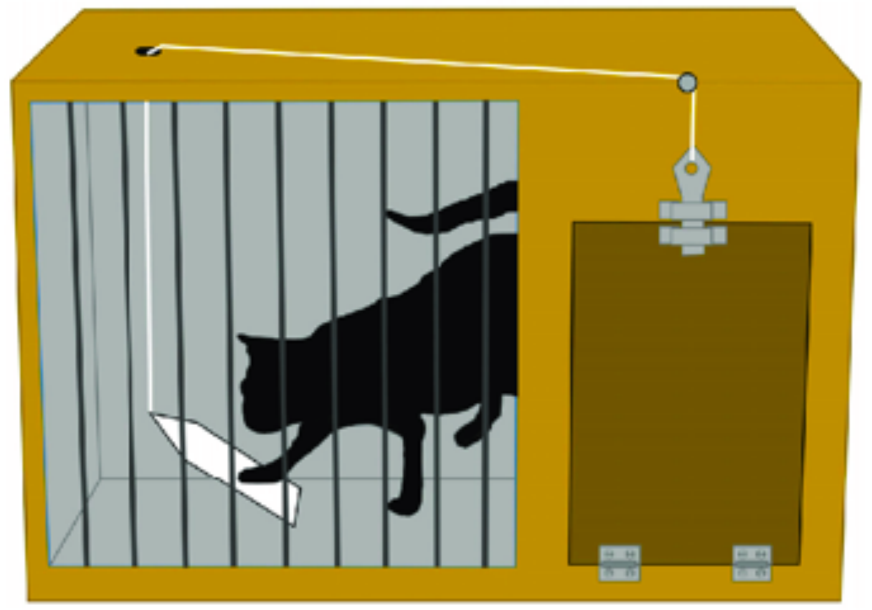
[1] linear Gaussian assumptions

Gradient descent



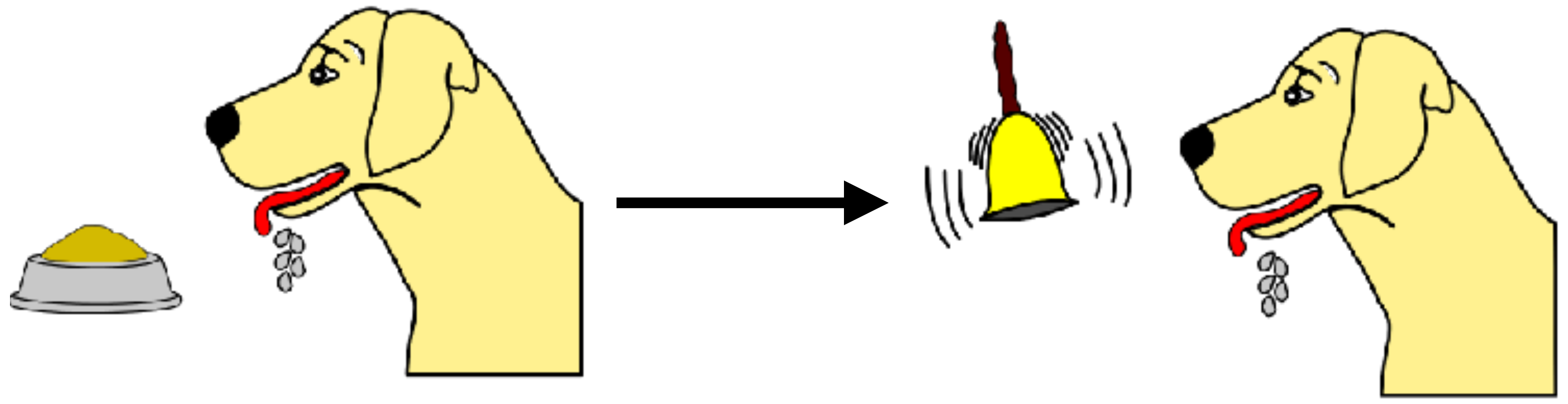
The story so far …

Thorndike's cats



- Law of effect
- Law of exercise

Pavlov's dog



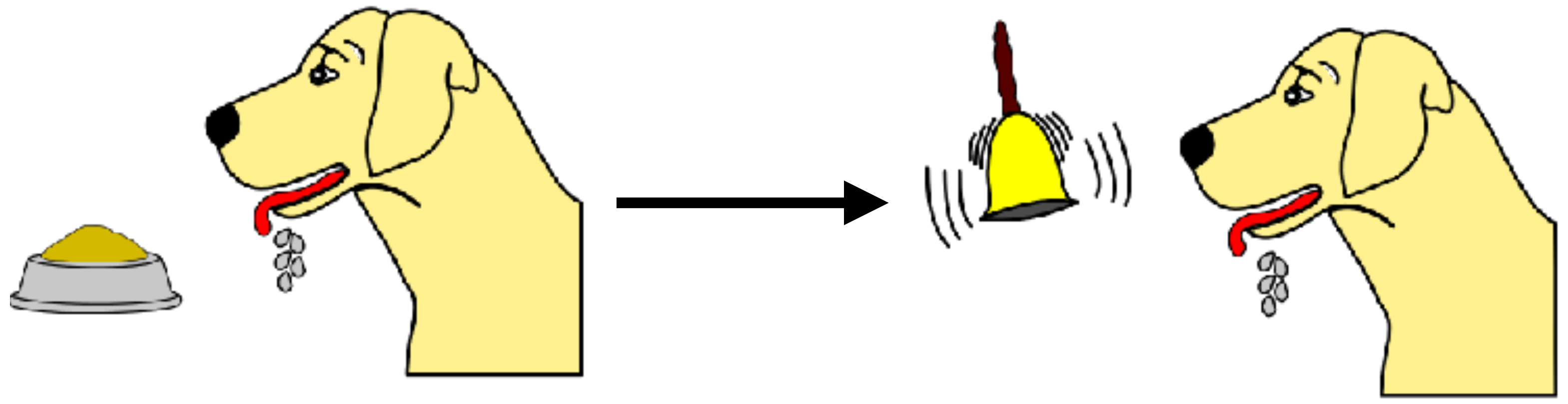
- Classical conditioning, where automatic response of US (salivation when given food) becomes associated with arbitrary CS (bell)
- Prediction error drives learning

The story so far …

Thorndike's cats

- Law of effect: repeat successful actions
- Law of exercise

Pavlov's dog



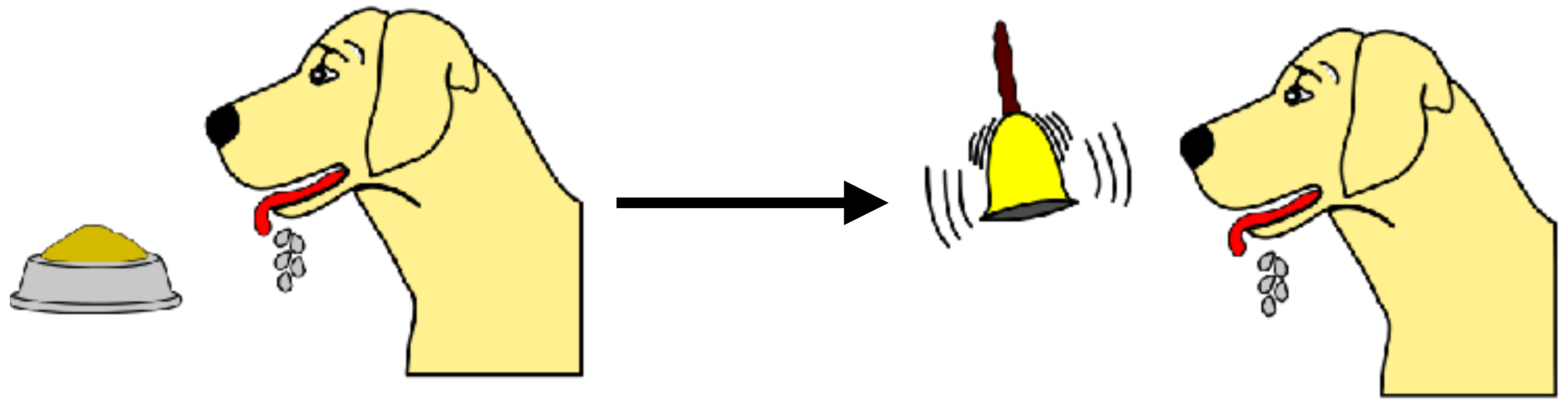
- Classical conditioning, where automatic response of US (salivation when given food) becomes associated with arbitrary CS (bell)
- Prediction error drives learning

The story so far …

Thorndike's cats

- Law of effect: repeat successful actions
- Law of exercise: repeat past actions, regardless of outcome

Pavlov's dog



- Classical conditioning, where automatic response of US (salivation when given food) becomes associated with arbitrary CS (bell)
- Prediction error drives learning

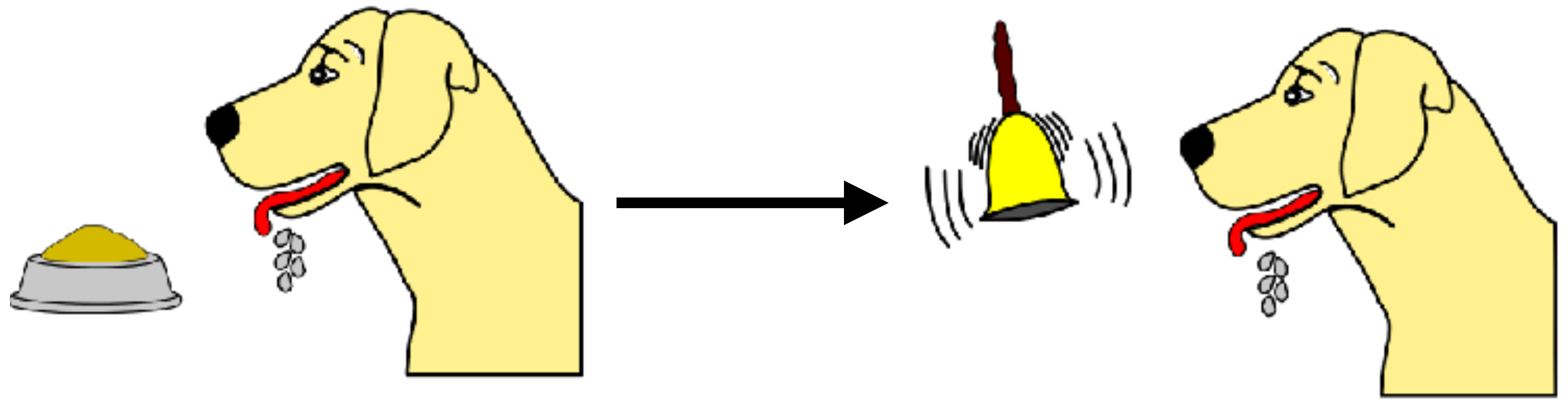
The story so far …

Thorndike’s cats

- Law of effect: repeat successful actions
- Law of exercise: repeat past actions, regardless of outcome

Pavlov’s dog

- Classical conditioning, where automatic response of US (salivation when given food) becomes associated with arbitrary CS (bell)
- Prediction error drives learning



Skinner’s pigeons

- Operant conditioning

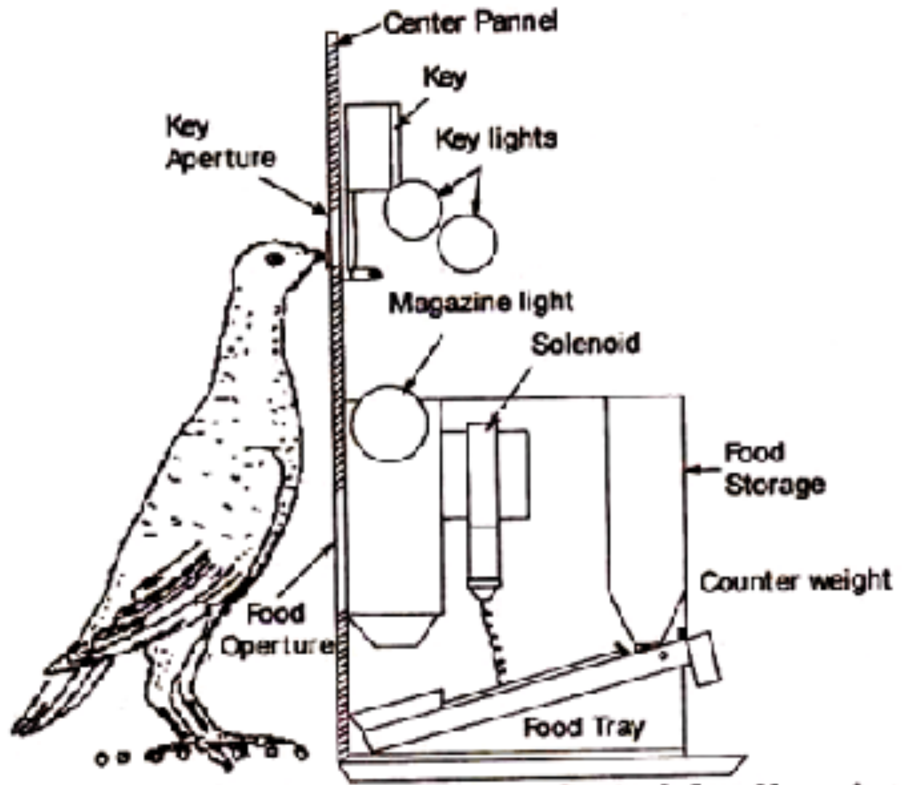


Illustration. Skinner box as adapted for the pigeon.

Operant Conditioning Skinner (1938)

- Building off of Thorndike's Law of Effect, operant conditioning studies how rewards shape the animal's behavior
- Operant conditioning describes the *active* selection of actions in response to rewards/ punishments
  - rather than only their *passive* association with stimuli (like in classical conditioning under Pavlov)
- This allows us to describe how animals learn to perform *actions* (conditioned on stimuli) that are predictive of reward

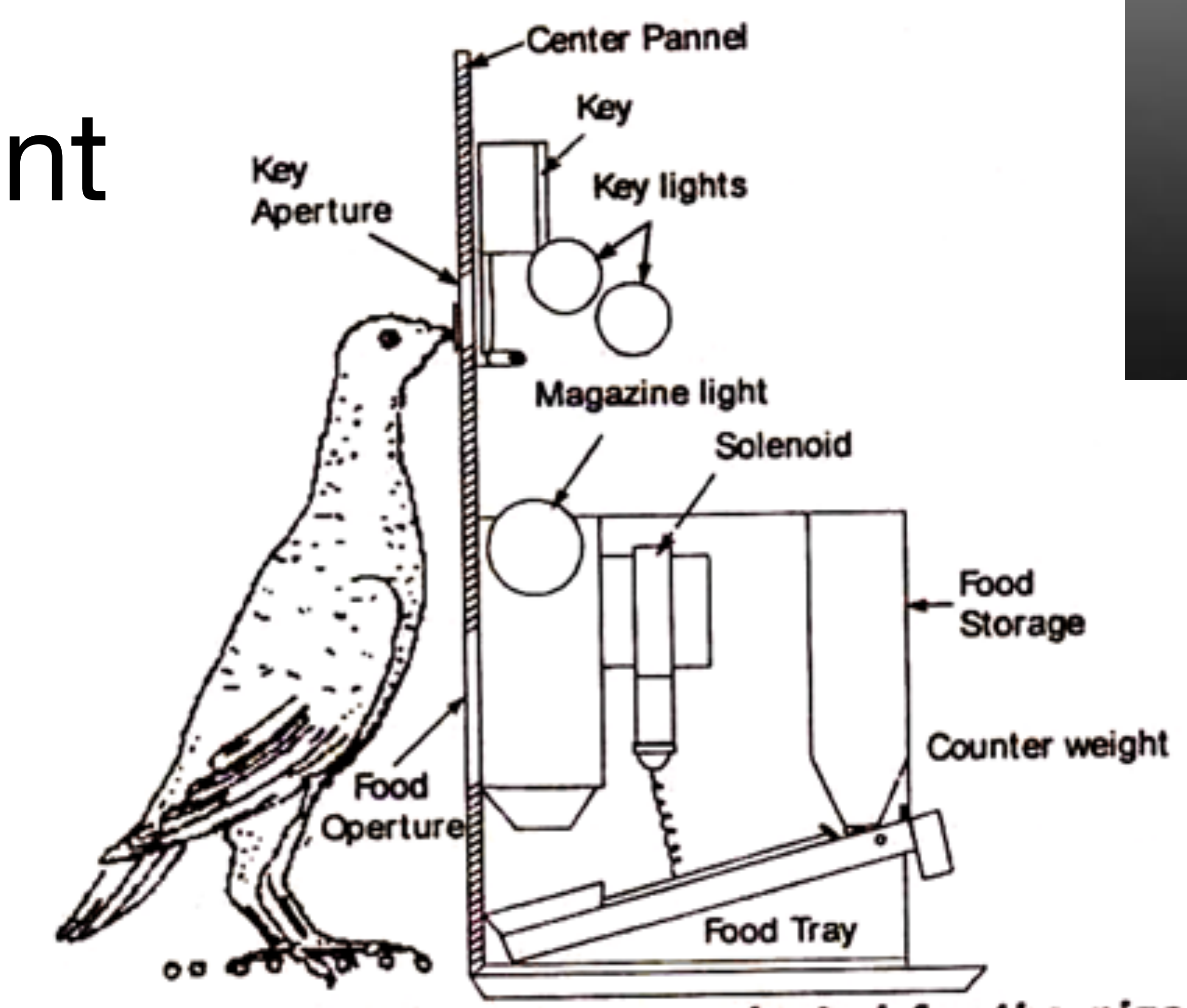
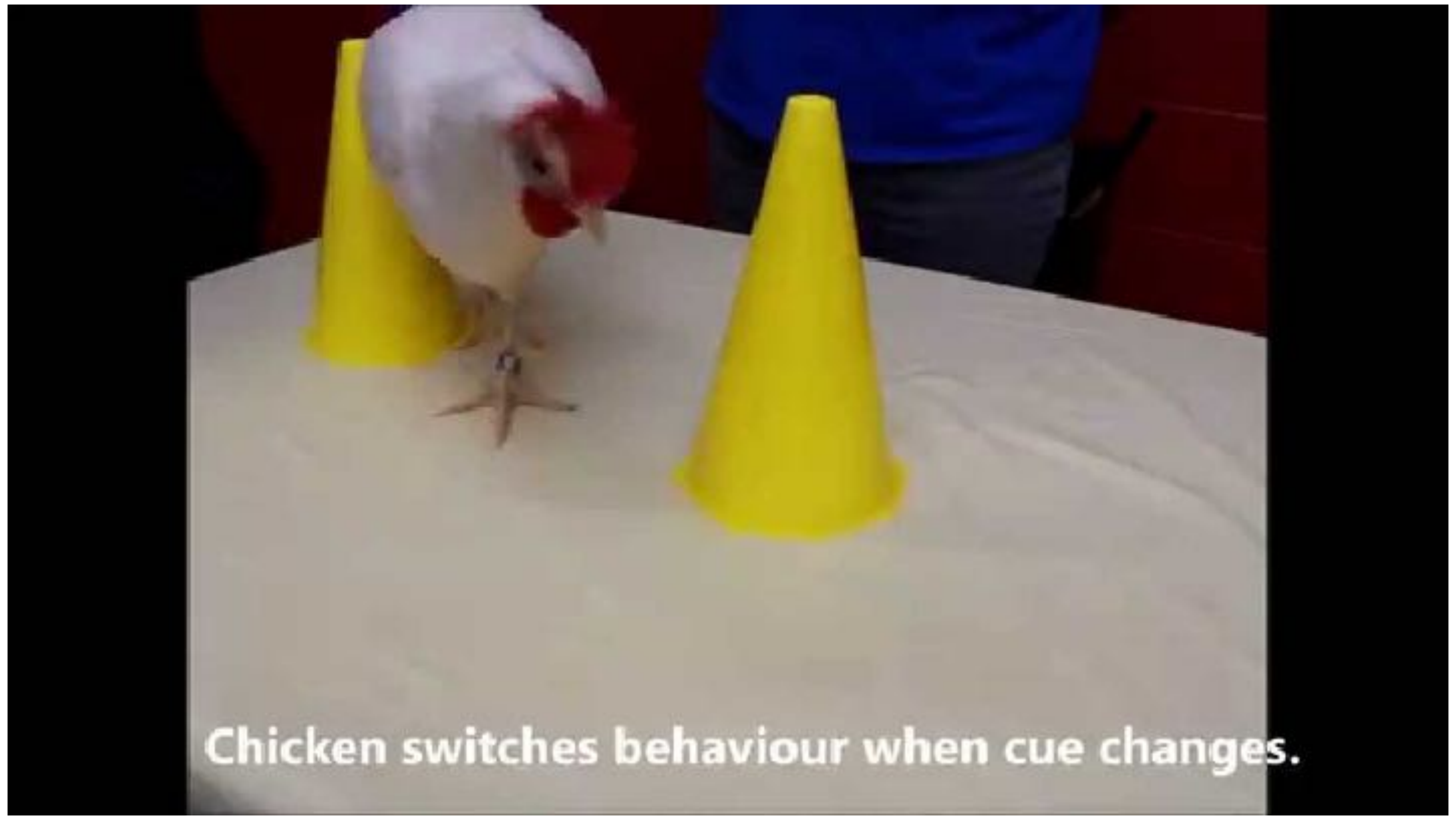


Illustration. Skinner box as adapted for the pigeon.



Operant Conditioning Skinner (1938)

- Building off of Thorndike's Law of Effect, operant conditioning studies how rewards shape the animal's behavior

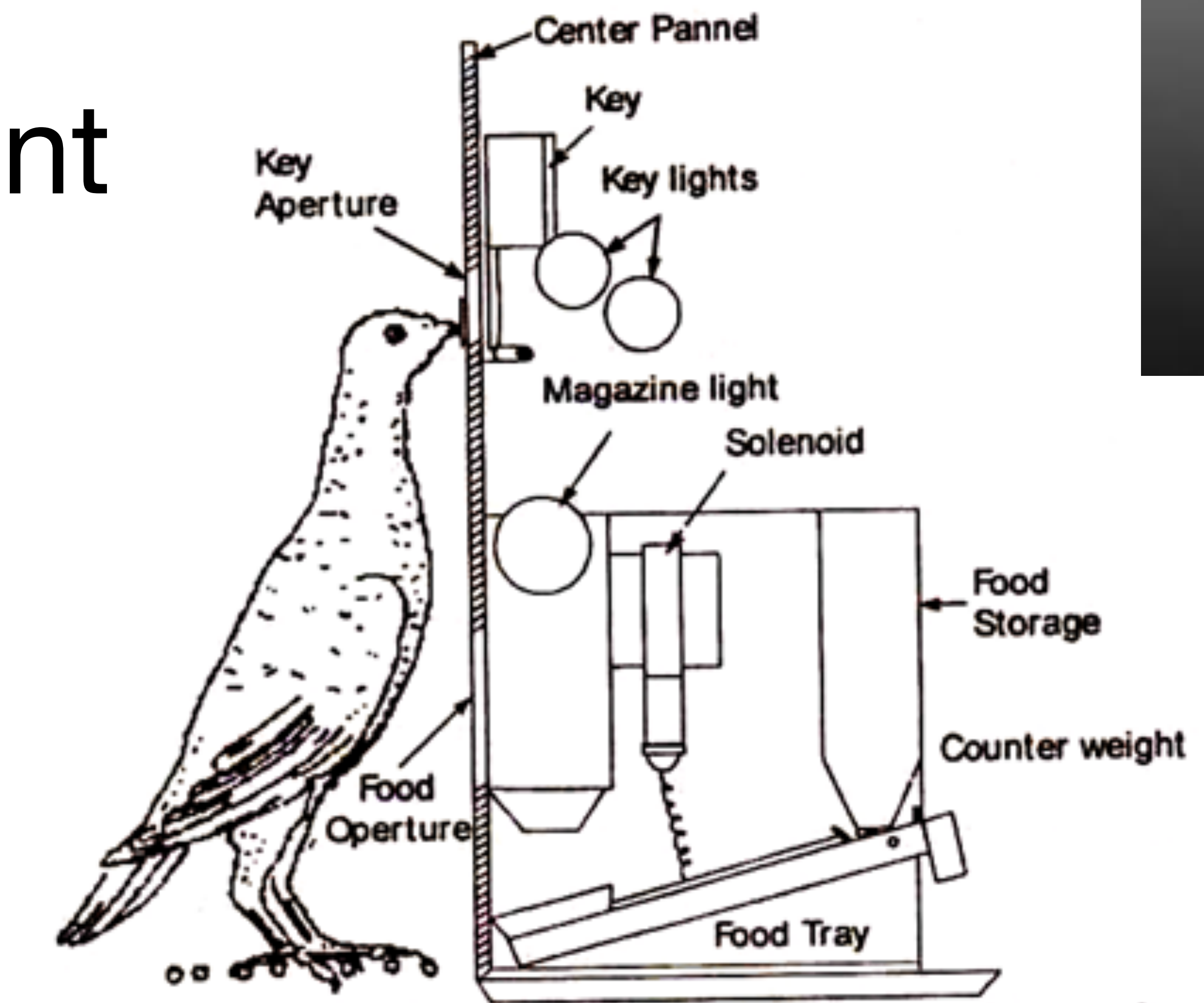
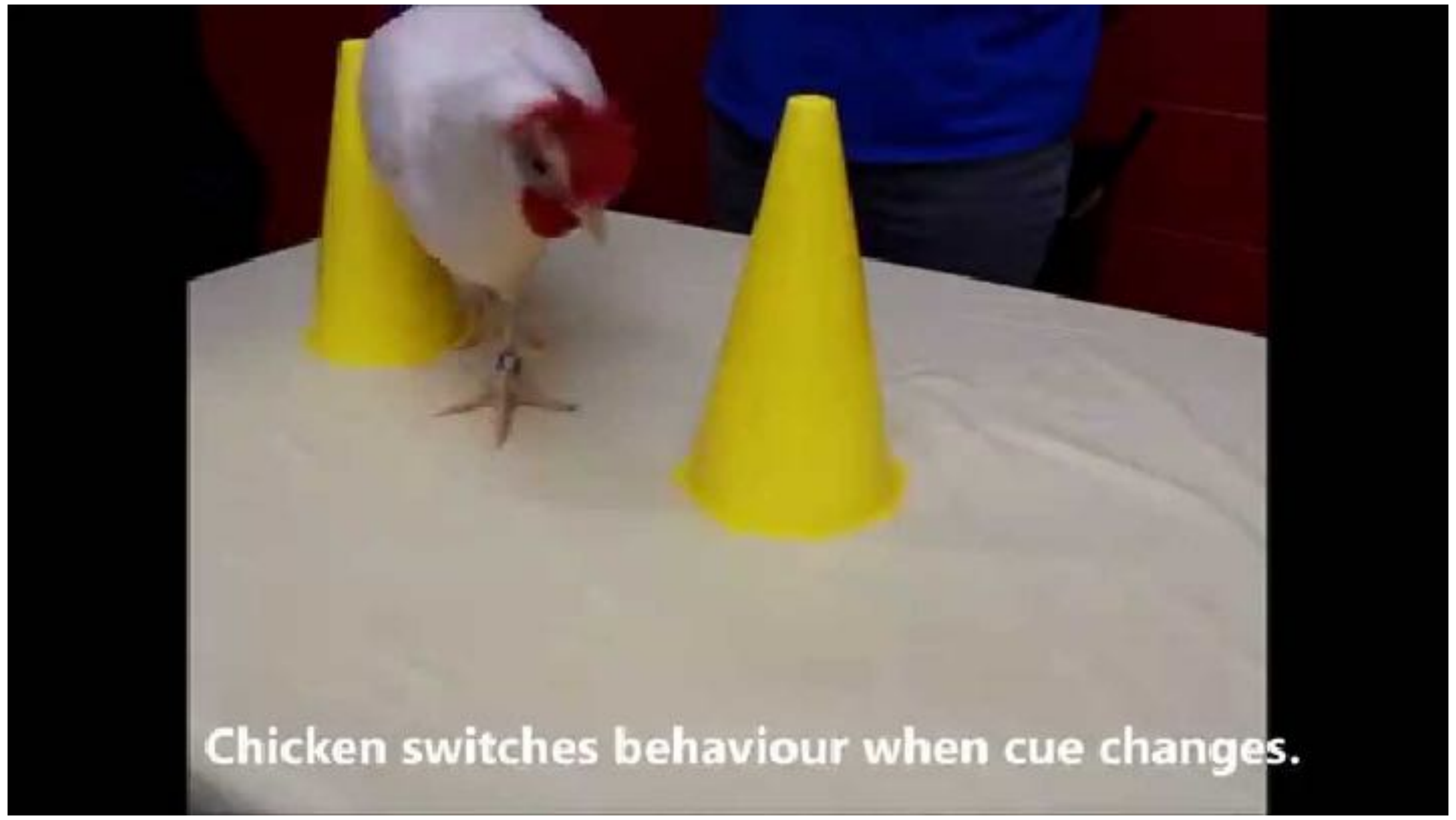


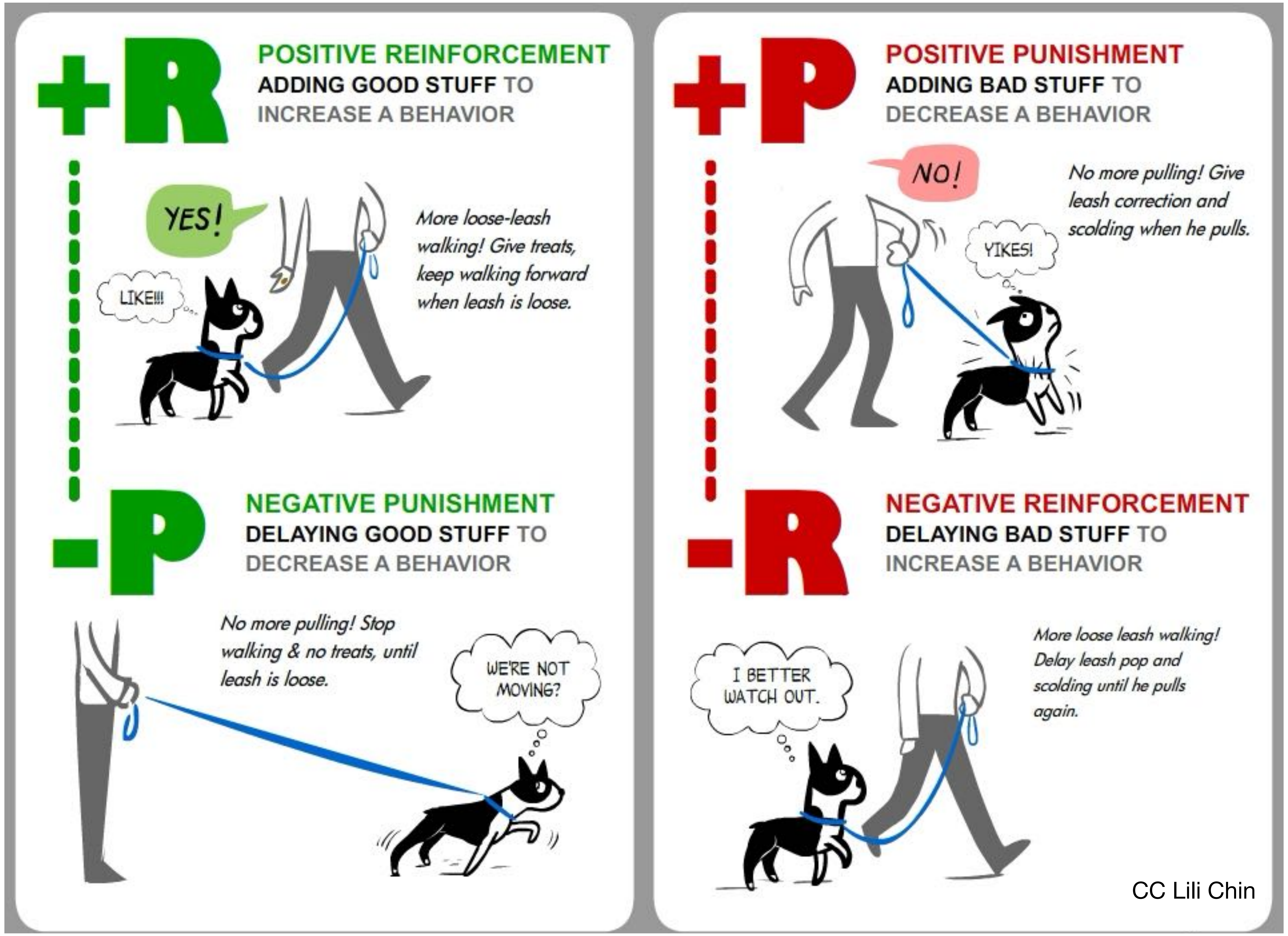
Illustration. Skinner box as adapted for the pigeon.

- Operant conditioning describes the *active* selection of actions in response to rewards/ punishments
  - rather than only their *passive* association with stimuli (like in classical conditioning under Pavlov)
- This allows us to describe how animals learn to perform *actions* (conditioned on stimuli) that are predictive of reward

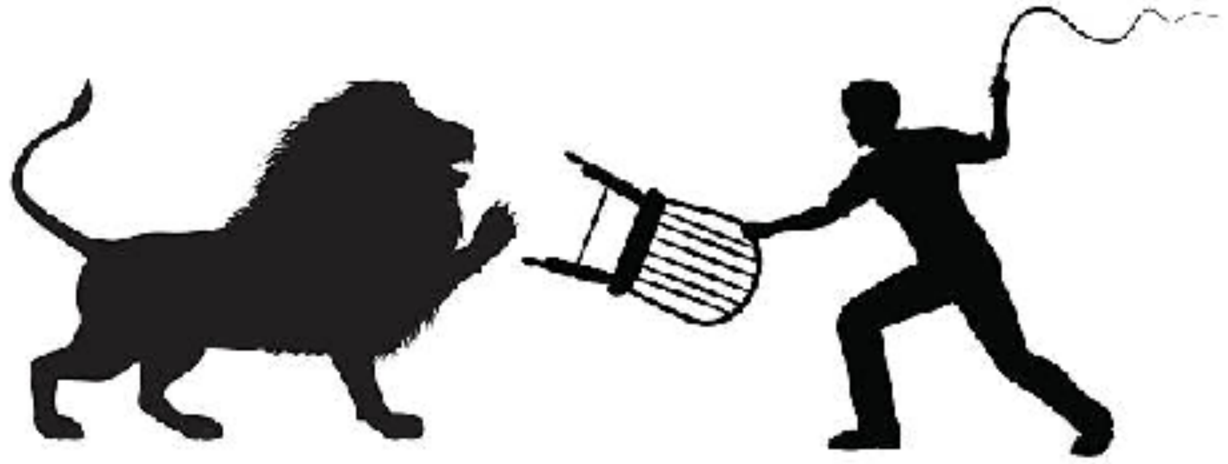


Operant conditioning in action

- Both **rewards** and **punishments** can be used to encourage desired behaviors
- **Rewards**/**punishments** can be either added or delayed, with different implications



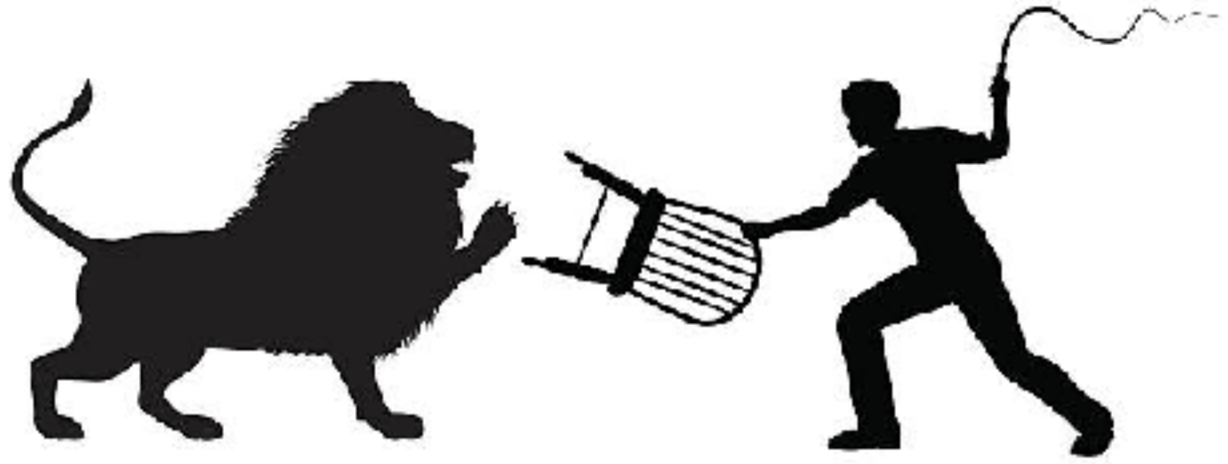
Behavioral Shaping



- Learning is slow when the space of possible actions is very large
- **Shaping** is a technique pioneered by Skinner to train a target behavior by rewarding *successive approximations*
  - adding rewards for smaller, intermediate steps to encourage exploration towards the target behavior

  1. Reinforce any response that resembles the desired behavior
  2. Iteratively reinforce responses that more selectively resemble the target behavior, and remove reinforcement from previously reinforced responses (causing *extinction*)

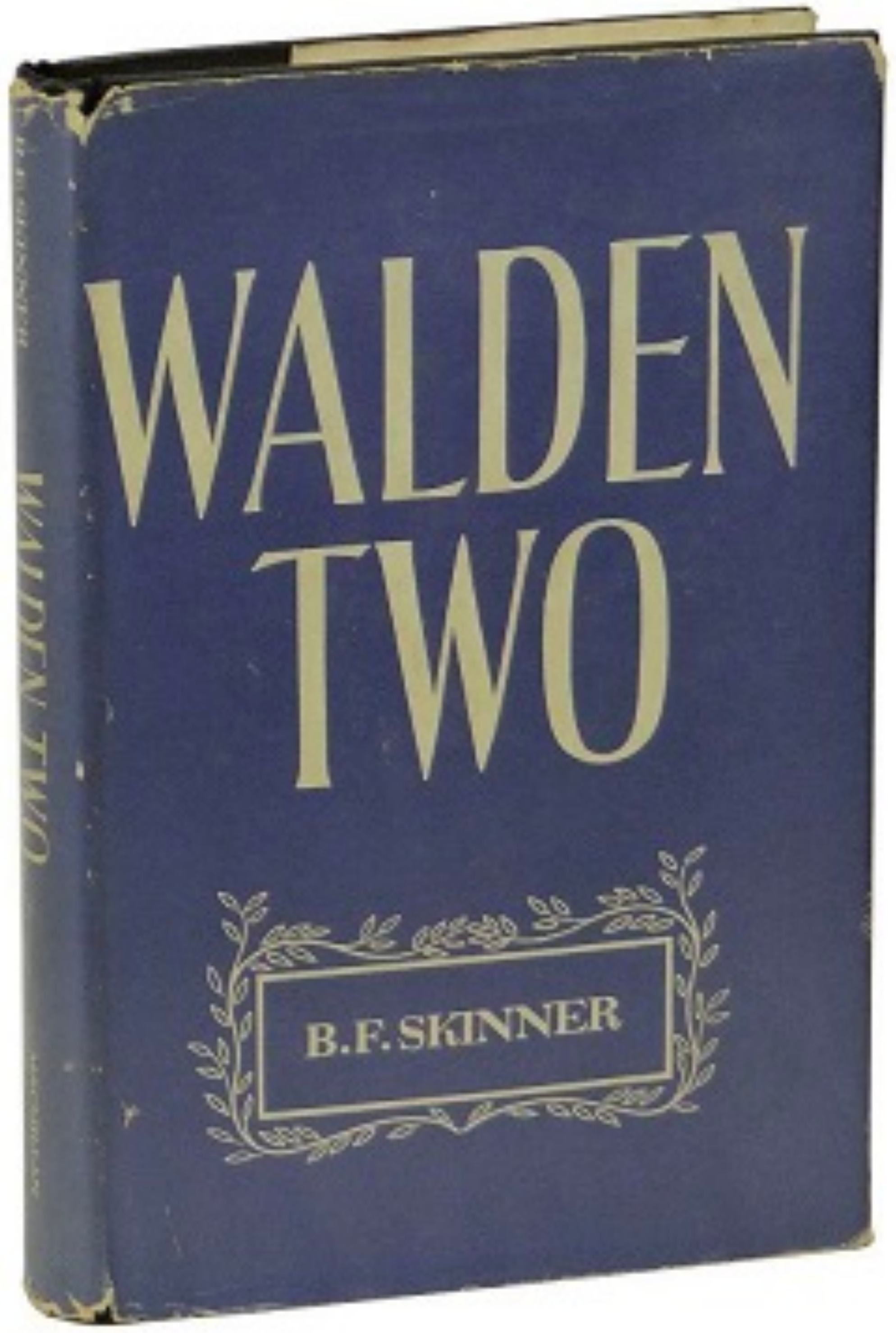
Behavioral Shaping



- Learning is slow when the space of possible actions is very large
- **Shaping** is a technique pioneered by Skinner to train a target behavior by rewarding *successive approximations*
  - adding rewards for smaller, intermediate steps to encourage exploration towards the target behavior
  1. Reinforce any response that resembles the desired behavior
  2. Iteratively reinforce responses that more selectively resemble the target behavior, and remove reinforcement from previously reinforced responses (causing *extinction*)

Dark side of Behavioralism

- Walden Two (1948) describes a Utopia, where behavioral engineering is used to shape a perfect society
  - From childhood, citizens are crafted through rewards and punishment into the ideal citizens and to value benefit for the common good
  - Rejection of free will, and has been criticized as creating a “perfectly efficient anthill”
- Is intelligence just learning to acquire reward and avoiding punishment?



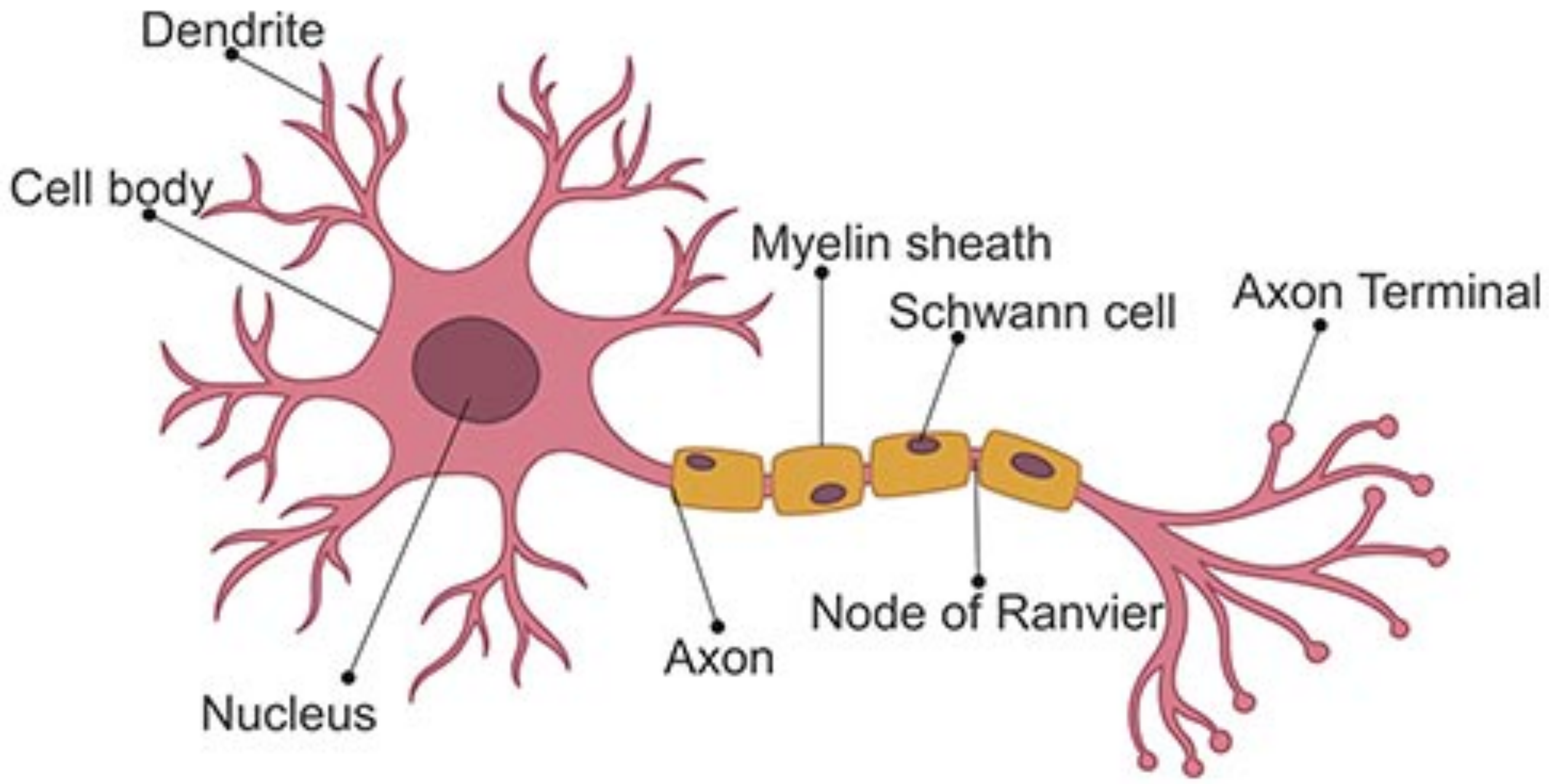
Summary so far

- **Behavioralism** tries to understand intelligence and learning by bracketing out unobservable mental phenomena. How far can we get with this approach?
- **Thorndike's Laws** describes two pathways for learning
  - Law of effect: Learning to repeat successful actions via trial and error learning
  - Law of exercise: Learning to repeat past actions (regardless of outcome)
- **Pavlovian (Classical) Conditioning** describes the association between stimuli and rewards based on predictions of reward
  - Rescorla Wagner (RW) model formalizes this theory based on *reward prediction error* (RPE) updating, which can be related to rational principles of maximum likelihood estimation and gradient descent
- **Operant conditioning** relates stimuli-reward associations to the active shaping of behavior, to acquire rewards and avoid punishment

5 minute break

Neural networks

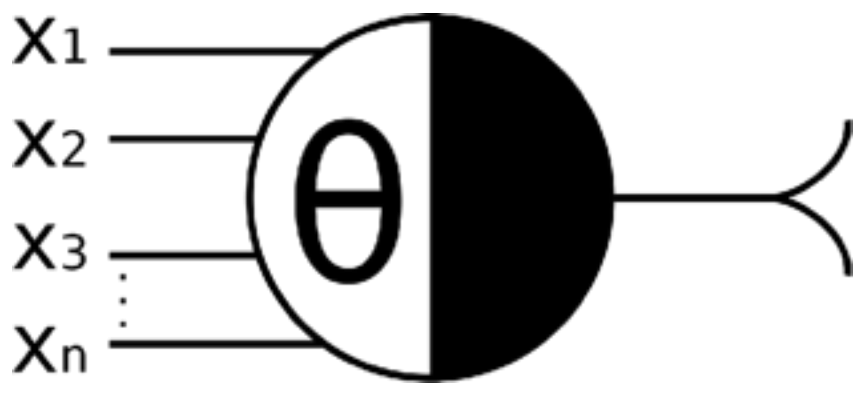
- Neurons are specialized cells that transmit information through electrical impulses
  - Roughly speaking, the dendrites receive information, which is processed in the cell body, and then propogated through the axon and synapses with other neurons
- Human perception, reasoning, emotions, actions, memory, and much more are governed by neural activity
- Whereas behaviorists focused on outward behavior, neuroscientists have been peering into black box for centuries in order to understand how neural activity gives rise to intelligence
- More recently (mid 1900s), artificial neural networks have been developed as computational tool for solving problems



Rosenblatt's Perceptron Mark I

Timeline of Artificial Neural Networks

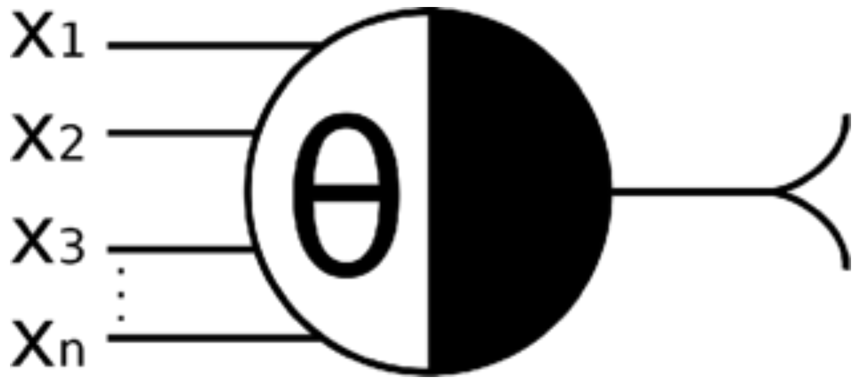
Timeline of Artificial Neural Networks



McCulloch & Pitts
(1943) Perceptron

Timeline of Artificial Neural Networks

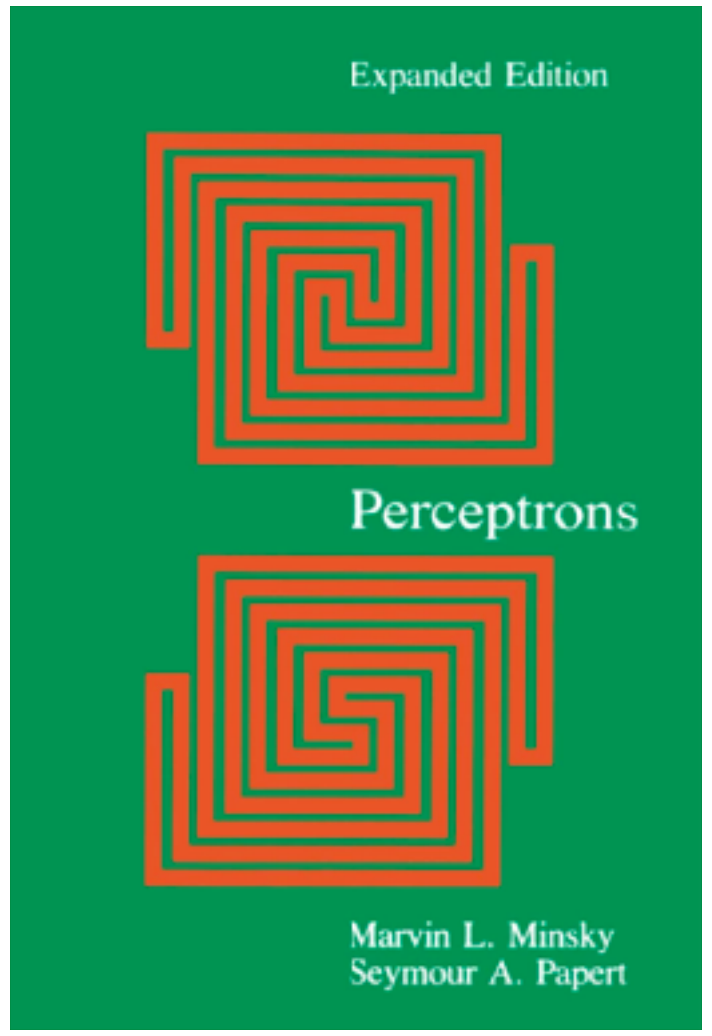
Rosenblatt (1958) Perceptron



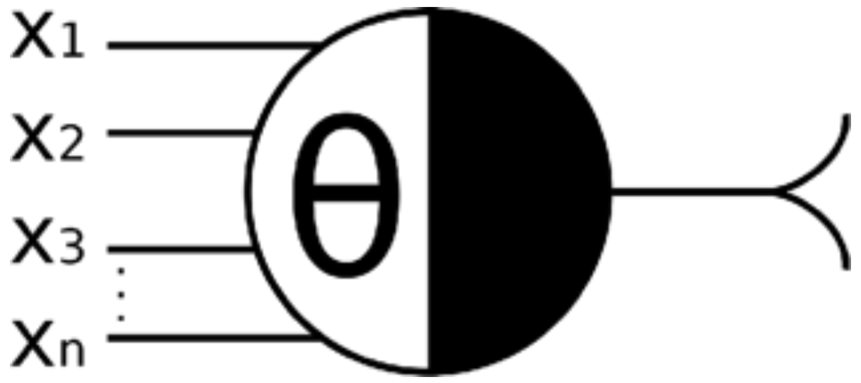
McCulloch & Pitts
(1943) Perceptron

Timeline of Artificial Neural Networks

Minsky & Parpert (1969)



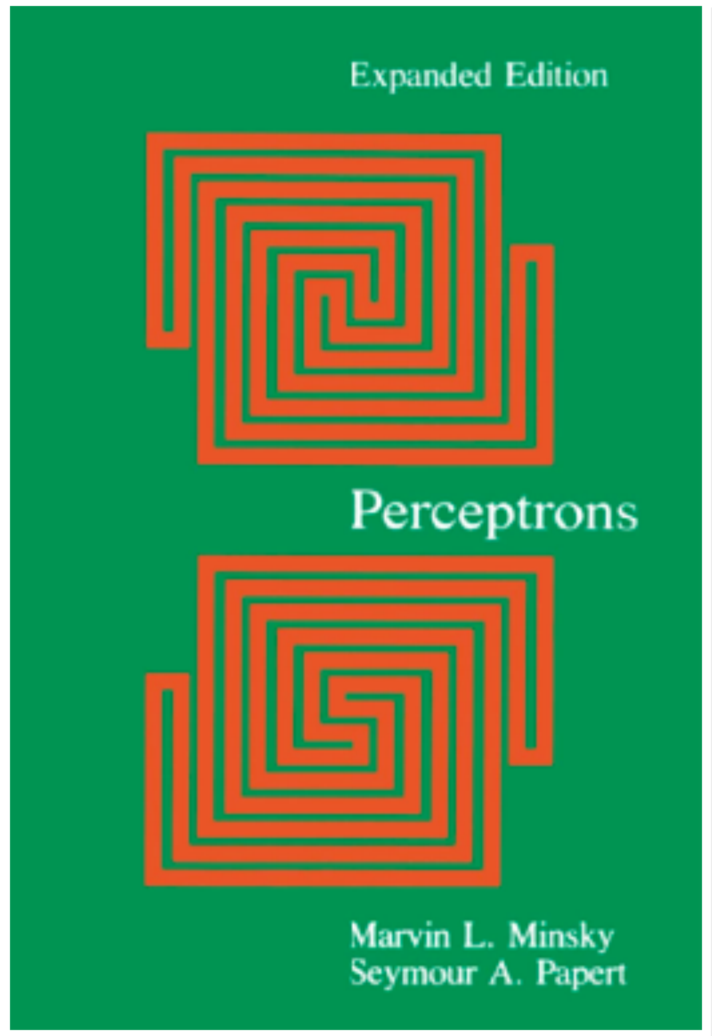
Rosenblatt (1958) Perceptron



McCulloch & Pitts
(1943) Perceptron

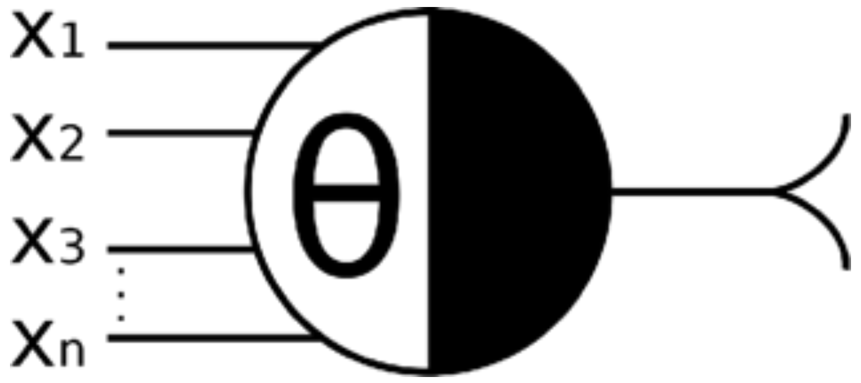
Timeline of Artificial Neural Networks

Minsky & Parpert (1969)



Rosenblatt (1958) Perceptron

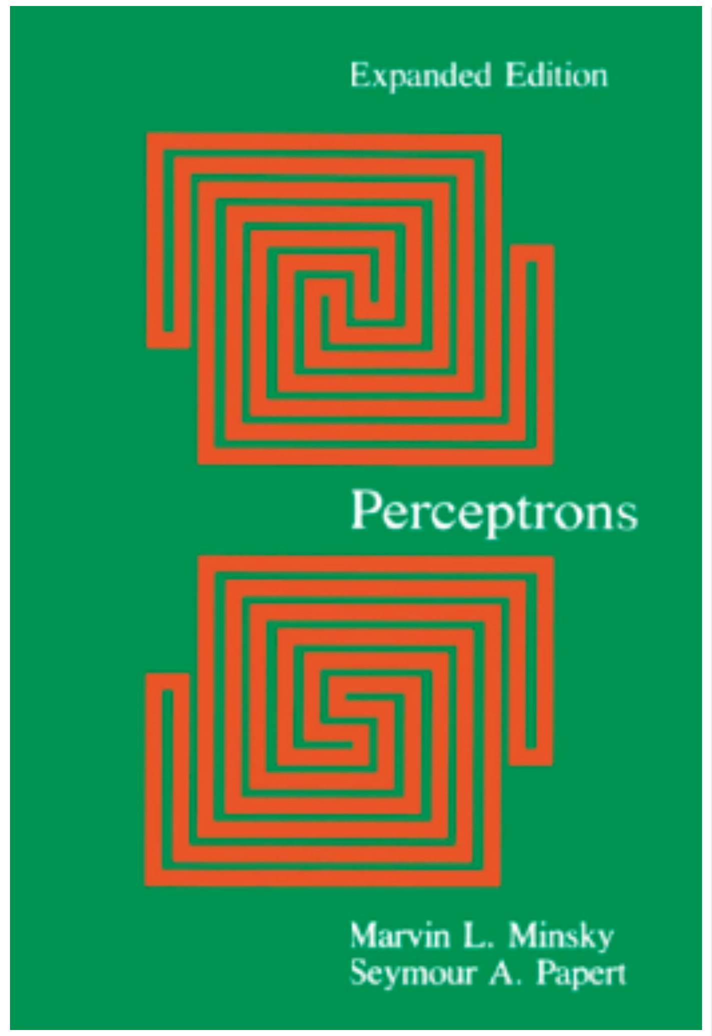
AI Winter



McCulloch & Pitts
(1943) Perceptron

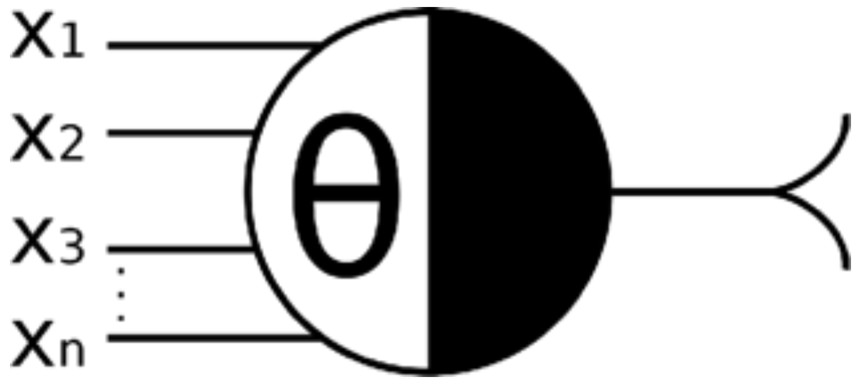
Timeline of Artificial Neural Networks

Minsky & Parpert (1969)

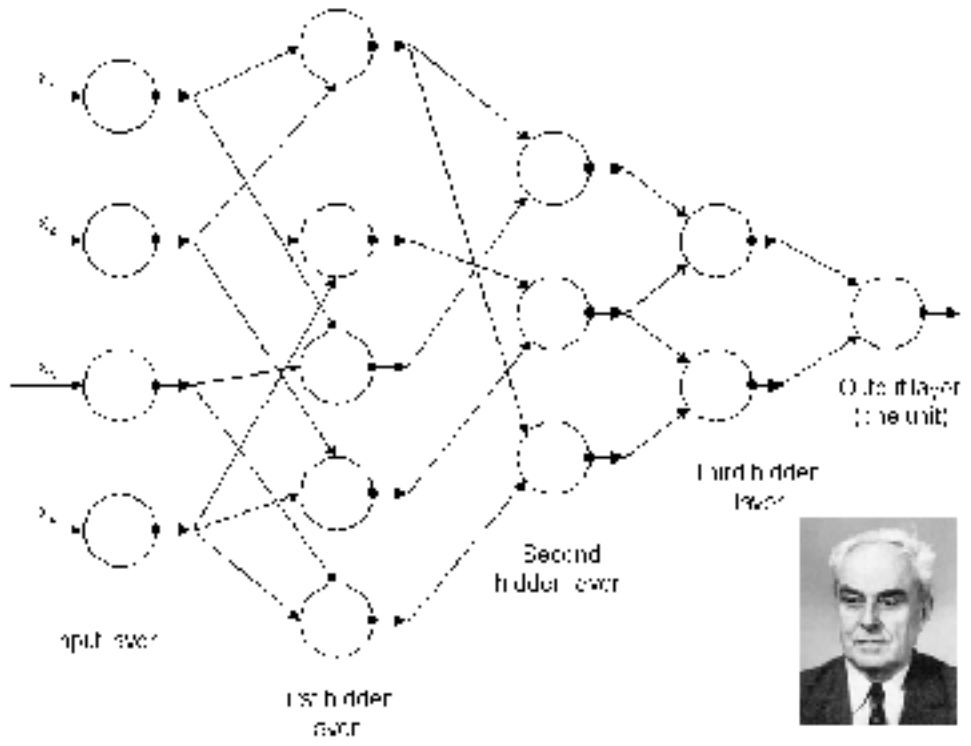


Rosenblatt (1958) Perceptron

AI Winter



McCulloch & Pitts
(1943) Perceptron

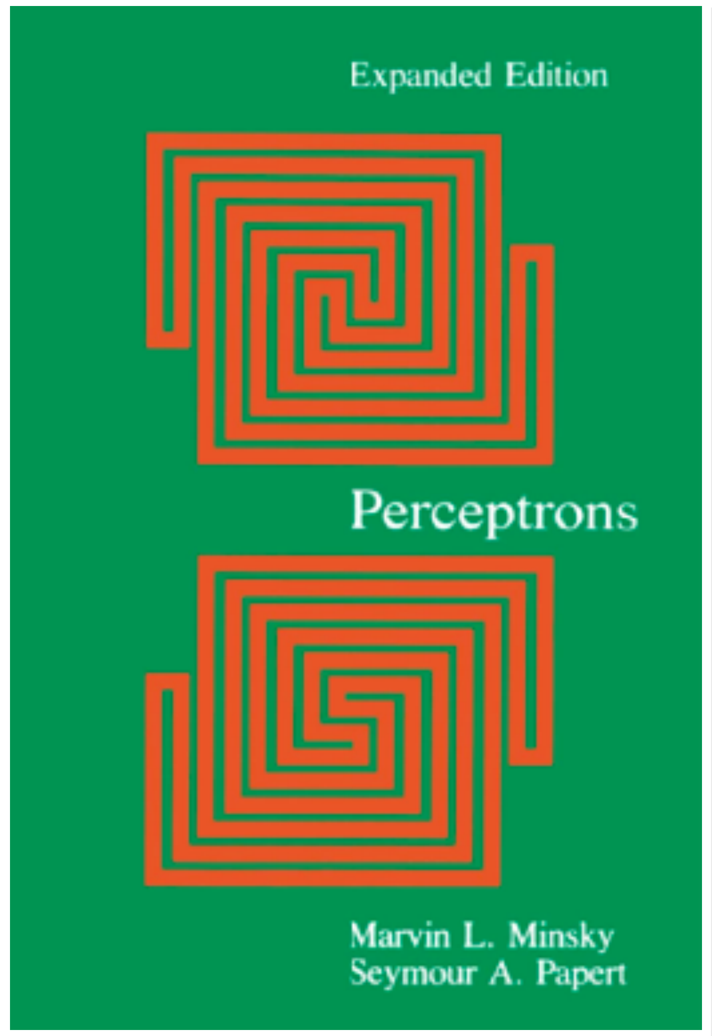


First deep network (Ivakhnenko & Lapa 1965)

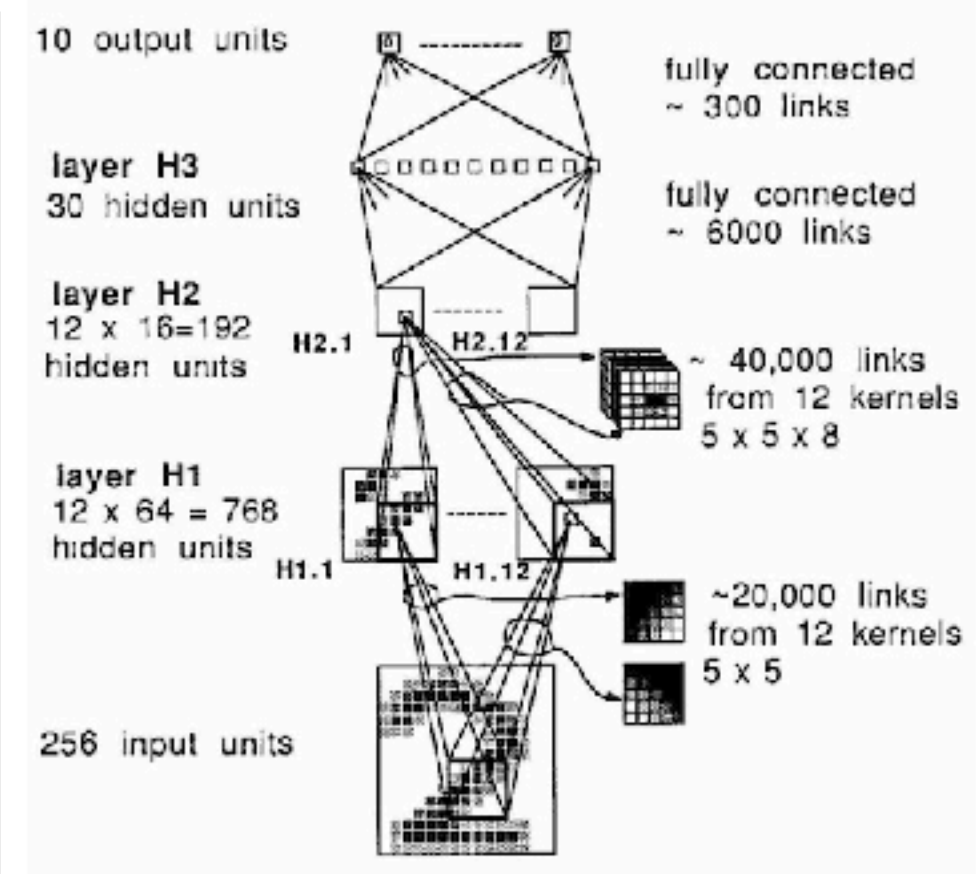
Timeline of Artificial Neural Networks

Rosenblatt (1958) Perceptron

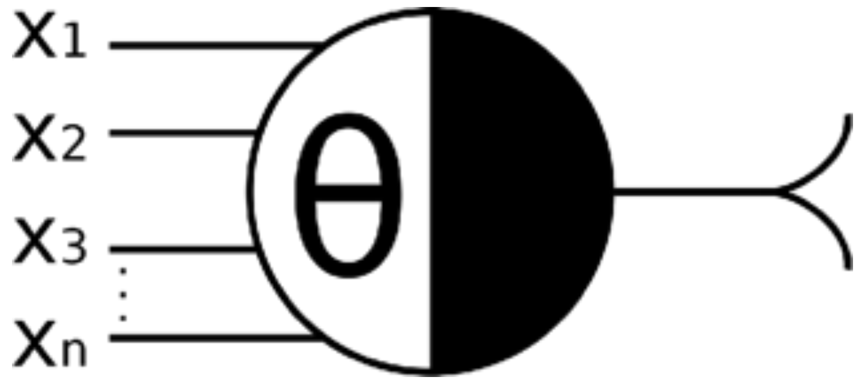
Minsky & Parpert (1969)



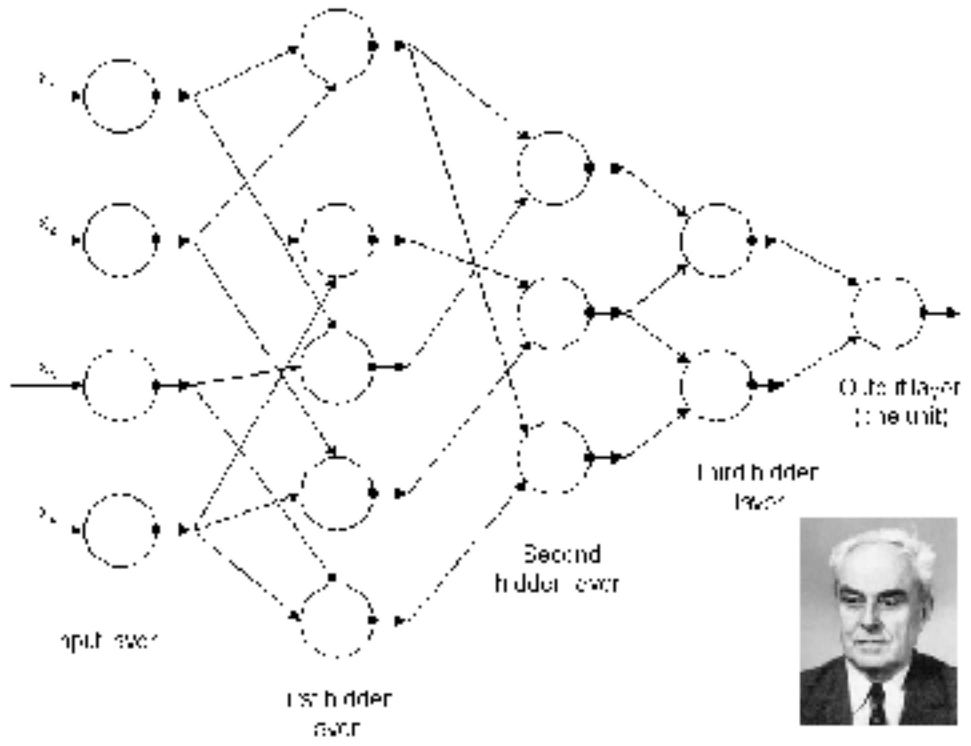
Convnets for MNIST (LeCun et al., 1989)



AI Winter



McCulloch & Pitts (1943) Perceptron



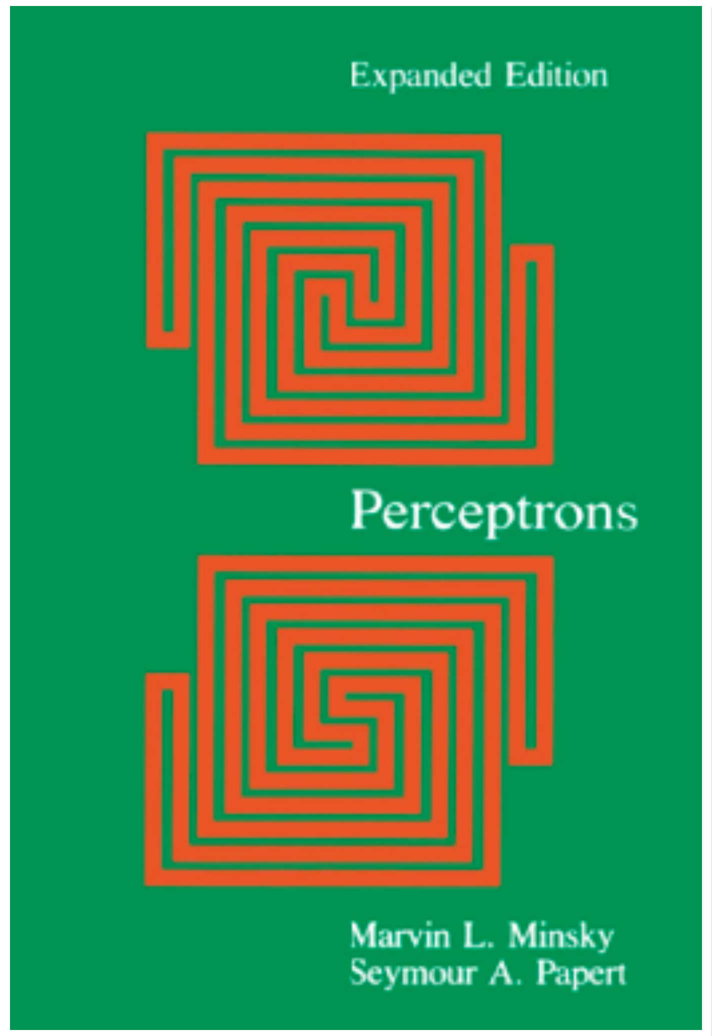
First deep network (Ivakhnenko & Lapa 1965)

Timeline of Artificial Neural Networks

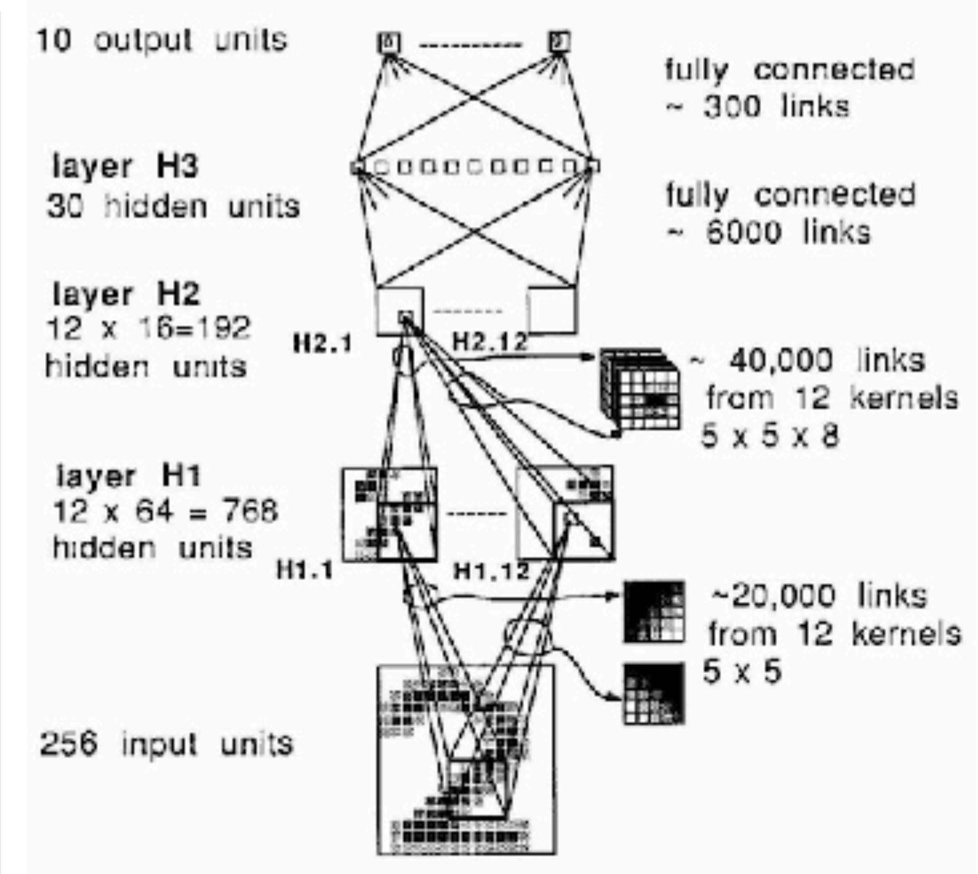
Deep Learning revolution

Rosenblatt (1958) Perceptron

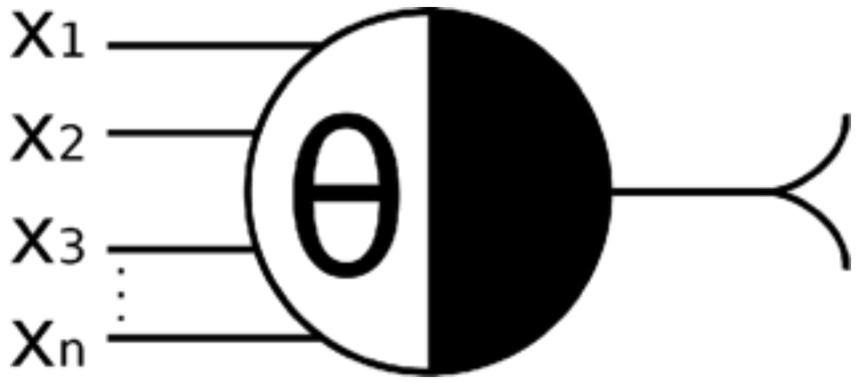
Minsky & Parpert (1969)



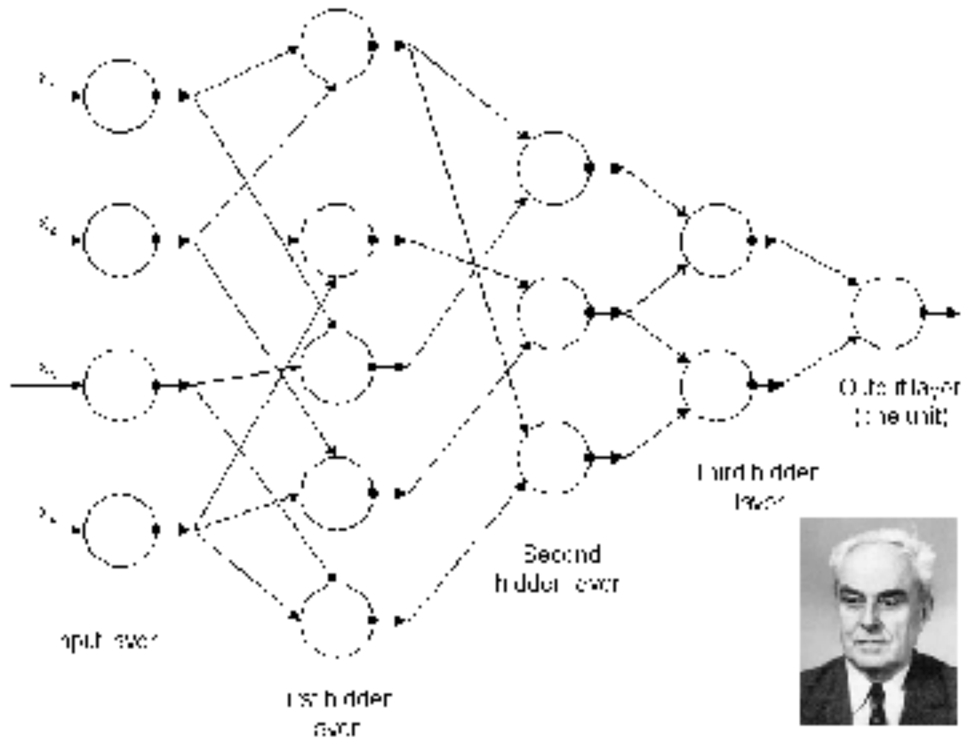
Convnets for MNIST (LeCun et al., 1989)



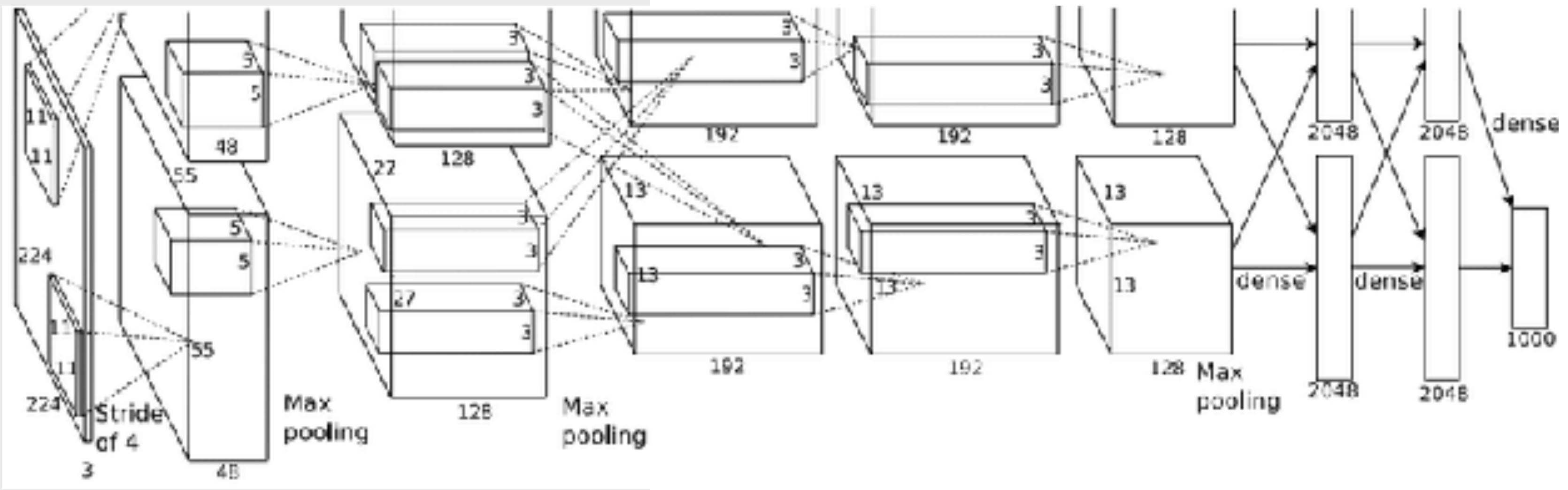
AI Winter



McCulloch & Pitts (1943) Perceptron

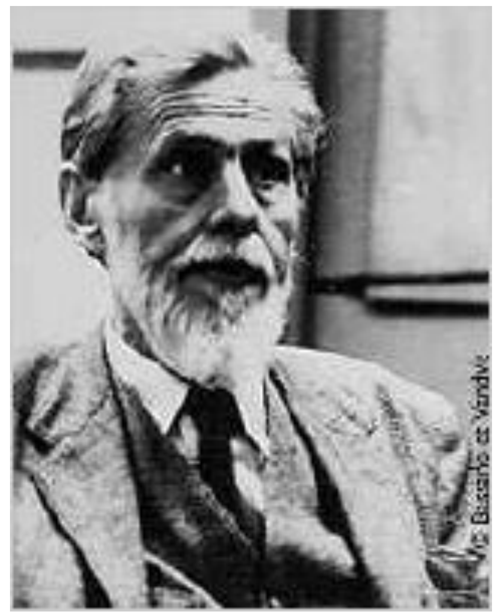


First deep network (Ivakhnenko & Lapa 1965)



ReLU & Dropout (Krizhevsky, Sutskever, & Hinton, 2012)

McCulloch & Pitts (1943)



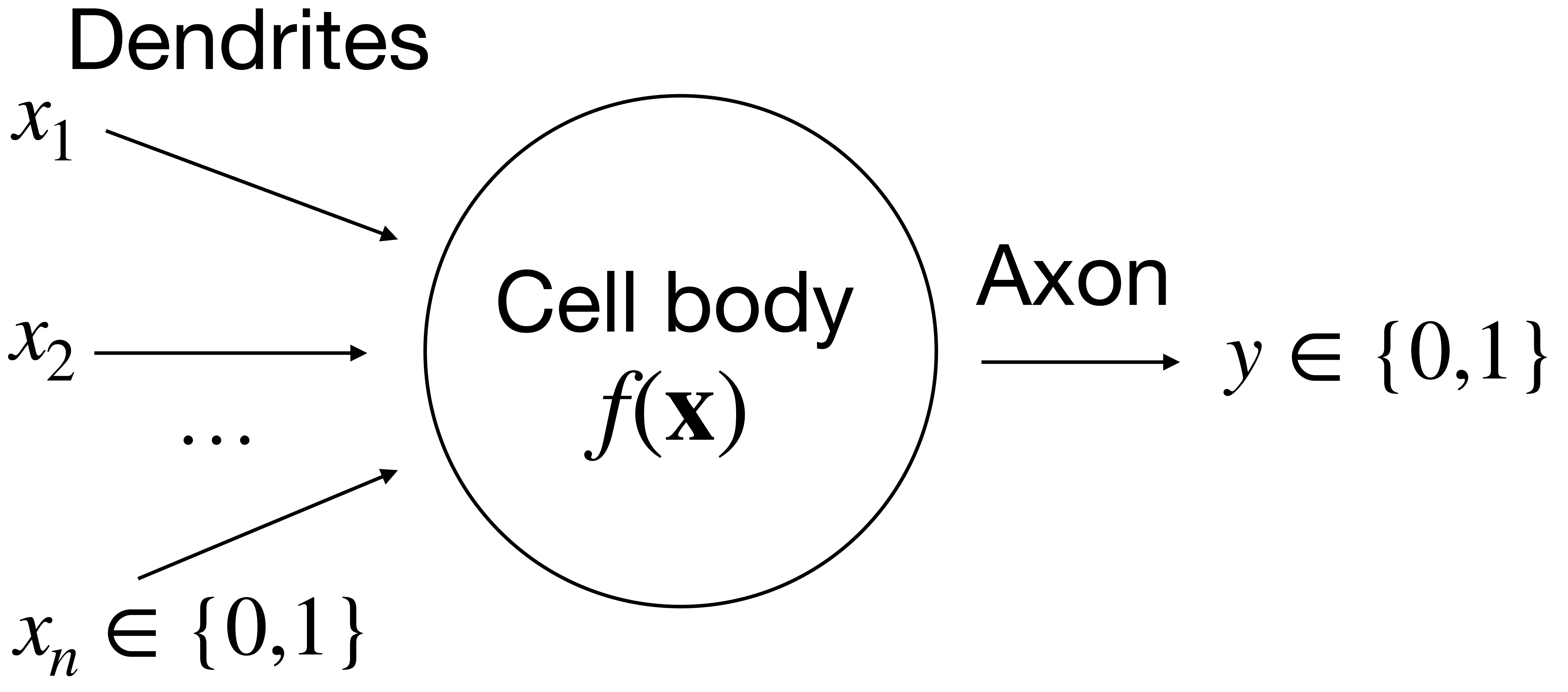
Warren McCulloch

Walter Pitts

- First computational model of a neuron
- The dendritic inputs $\{x_1, \dots, x_n\}$ provide the input signal
- The cell body processes the signal

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum x_i \geq \theta \\ 0 & \text{else} \end{cases}$$

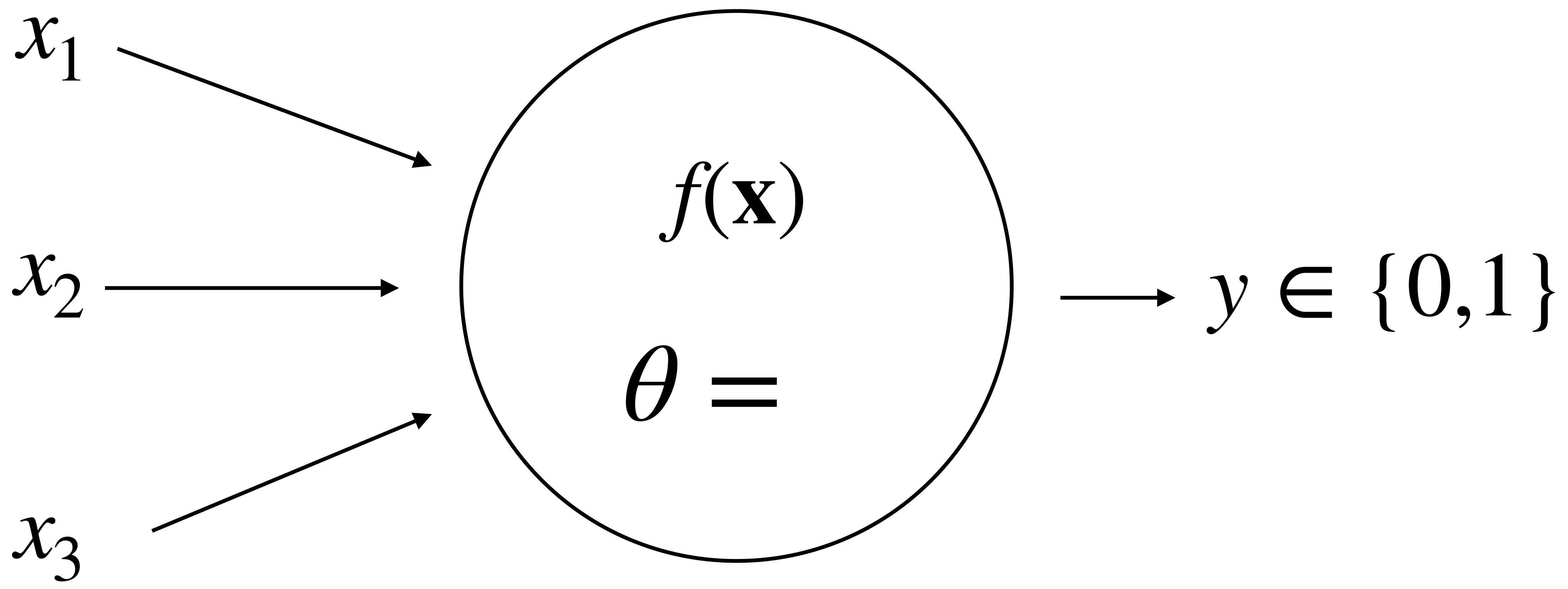
- If the sum of the inputs is greater or equal to some *threshold* θ, then the axon produces the output



McCulloch & Pitts (1943)

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if} \sum x_i \geq \theta \\ 0 & \text{else} \end{cases}$$

AND function

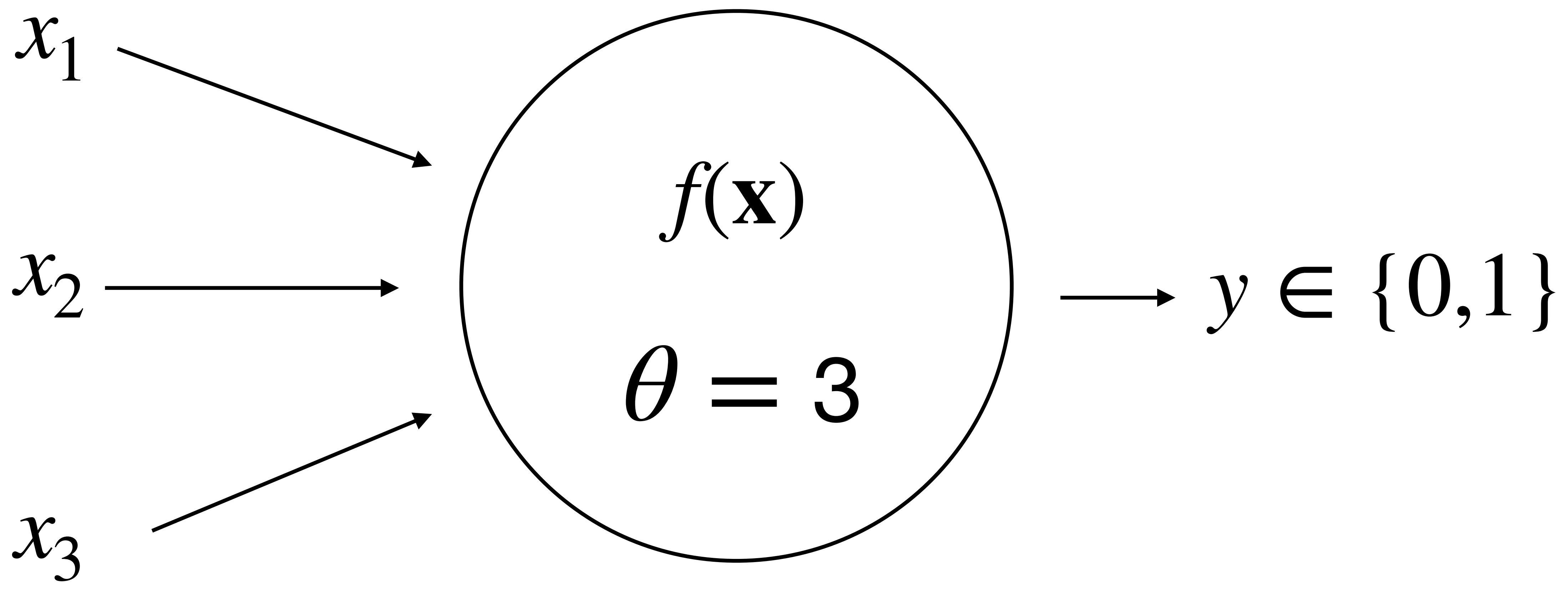


All inputs need to be on for the neuron to fire

McCulloch & Pitts (1943)

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum x_i \geq \theta \\ 0 & \text{else} \end{cases}$$

AND function

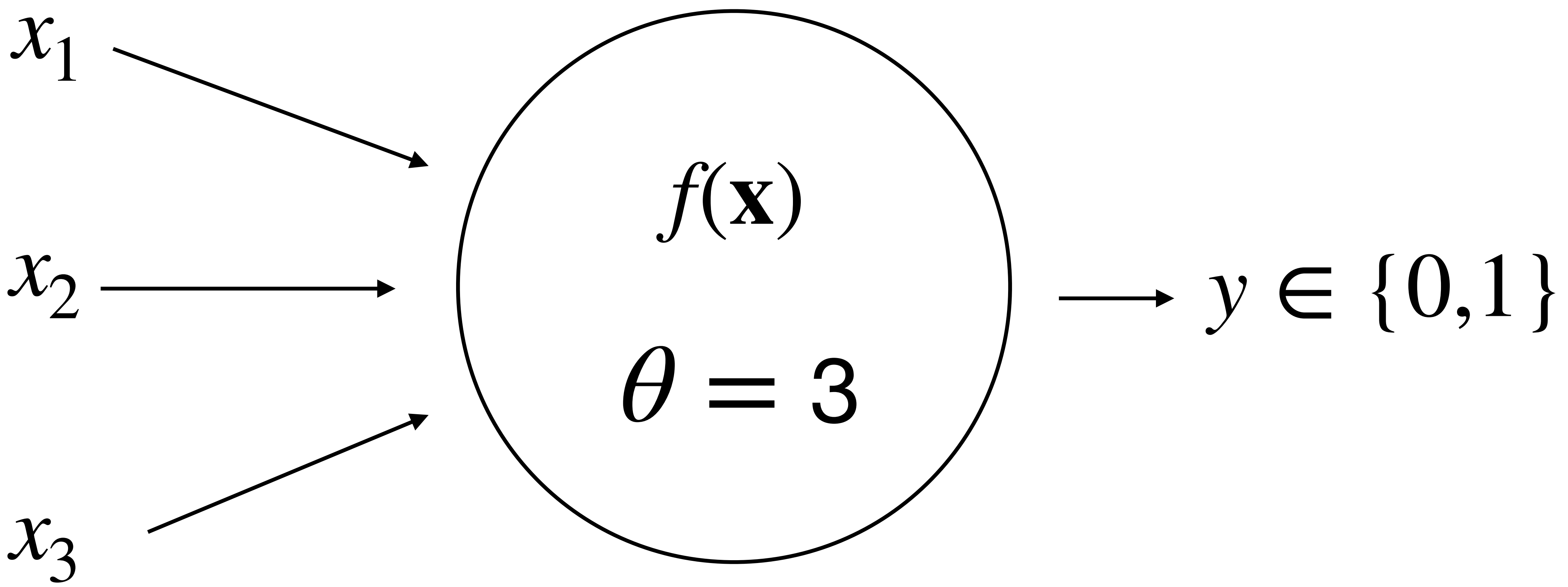


All inputs need to be on for the neuron to fire

McCulloch & Pitts (1943)

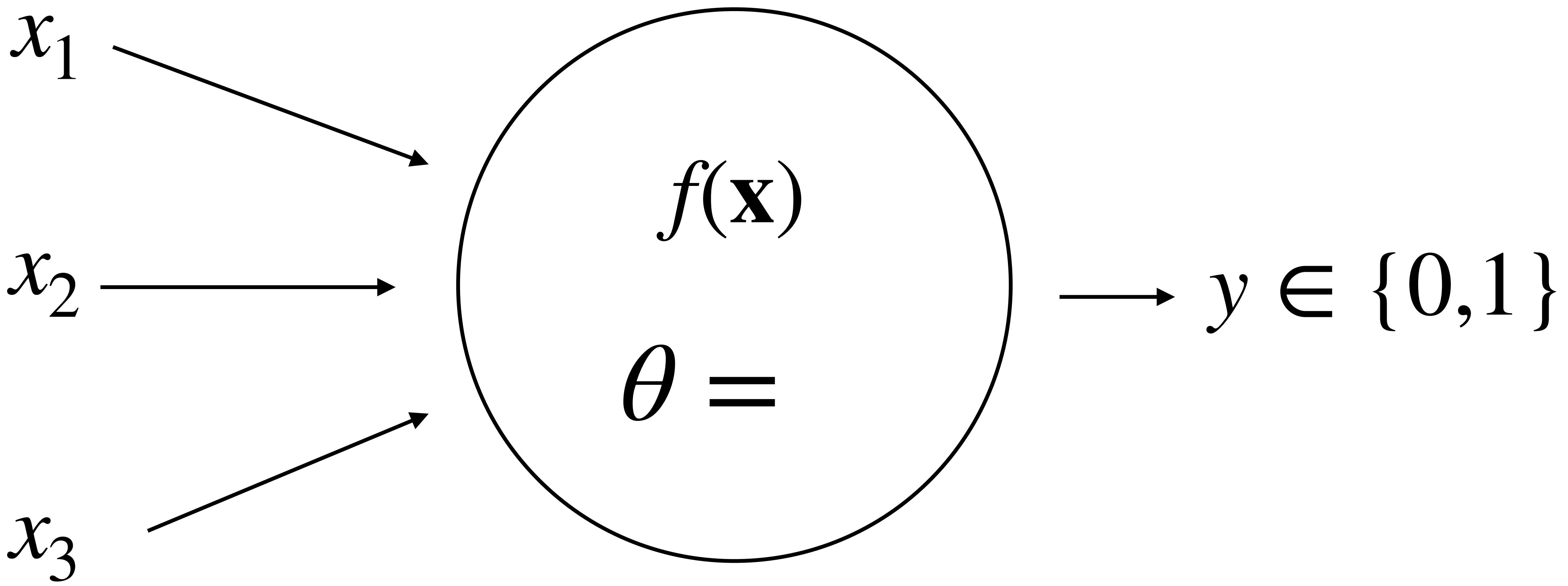
$$f(\mathbf{x}) = \begin{cases} 1 & \text{if} \sum x_i \geq \theta \\ 0 & \text{else} \end{cases}$$

AND function



All inputs need to be on for the neuron to fire

OR function

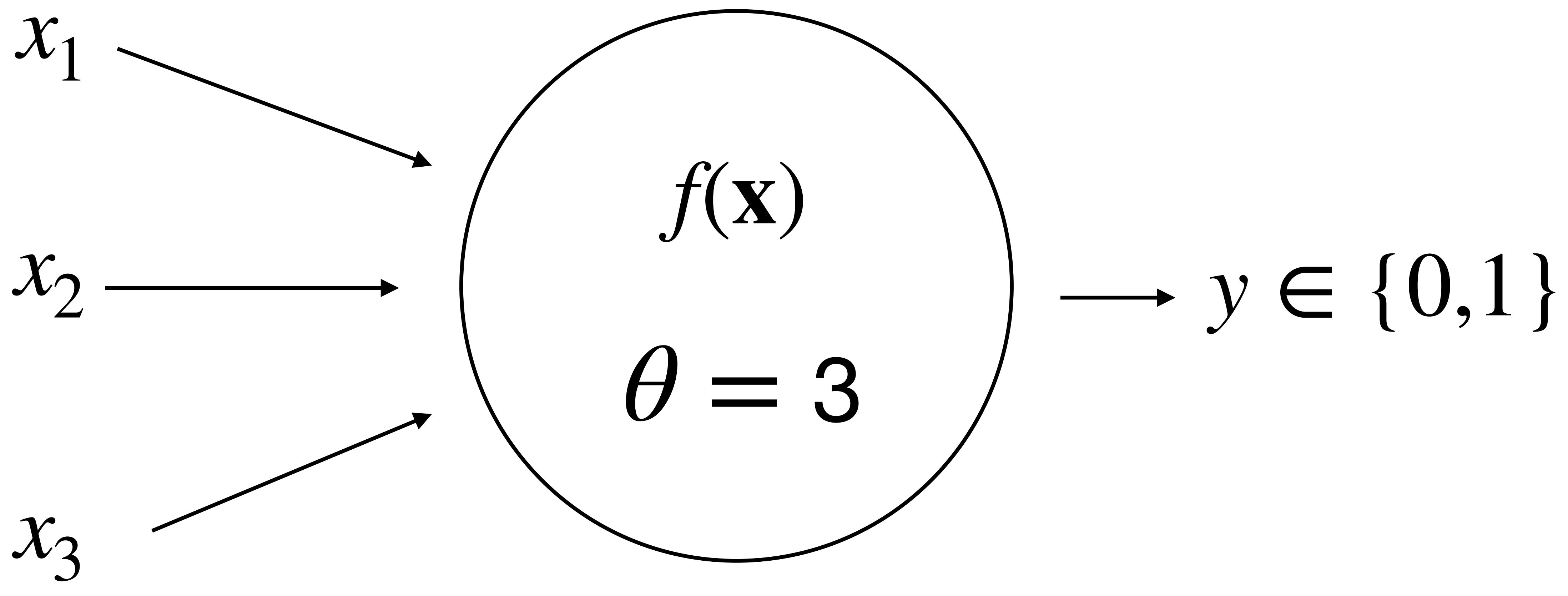


Neuron fires if any input is on

McCulloch & Pitts (1943)

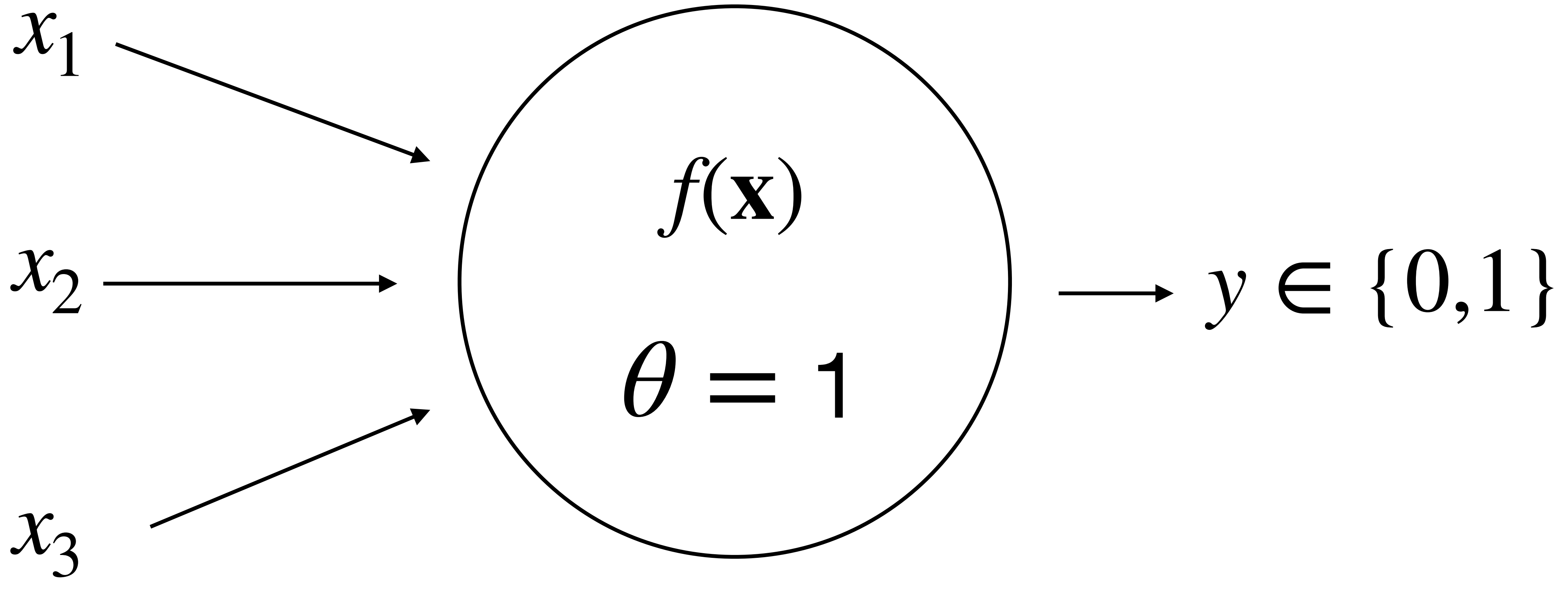
$$f(\mathbf{x}) = \begin{cases} 1 & \text{if} \sum x_i \geq \theta \\ 0 & \text{else} \end{cases}$$

AND function



All inputs need to be on for the neuron to fire

OR function



Neuron fires if any input is on

McCulloch & Pitts (1943)

NOT function

?

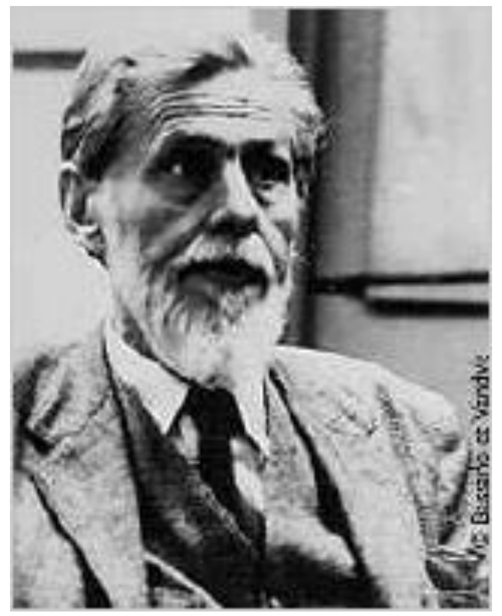
Neuron fires if no inputs are on

NAND

?

Neuron fires when x_1 is on AND x_2 not on

McCulloch & Pitts (1943)



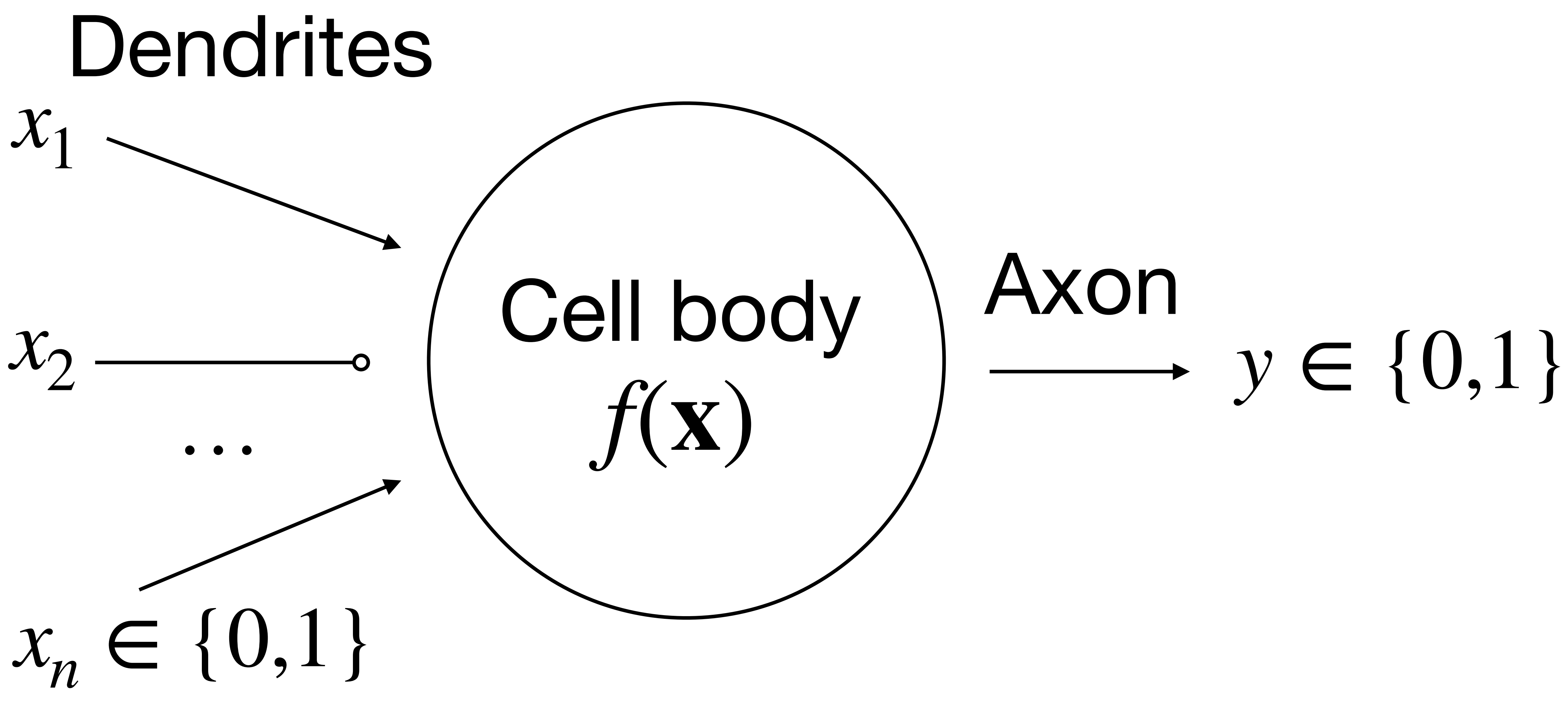
Warren McCulloch

Walter Pitts

- First computational model of a neuron
- The dendritic inputs $\{x_1, \ldots, x_n\}$ provide the input signal
  - Excitatory $\longrightarrow$ $w = 1$
  - Inhibitory $\multimap$ $w = -1$
- The cell body processes the signal

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum w_i x_i \geq \theta \\ 0 & \text{else} \end{cases}$$

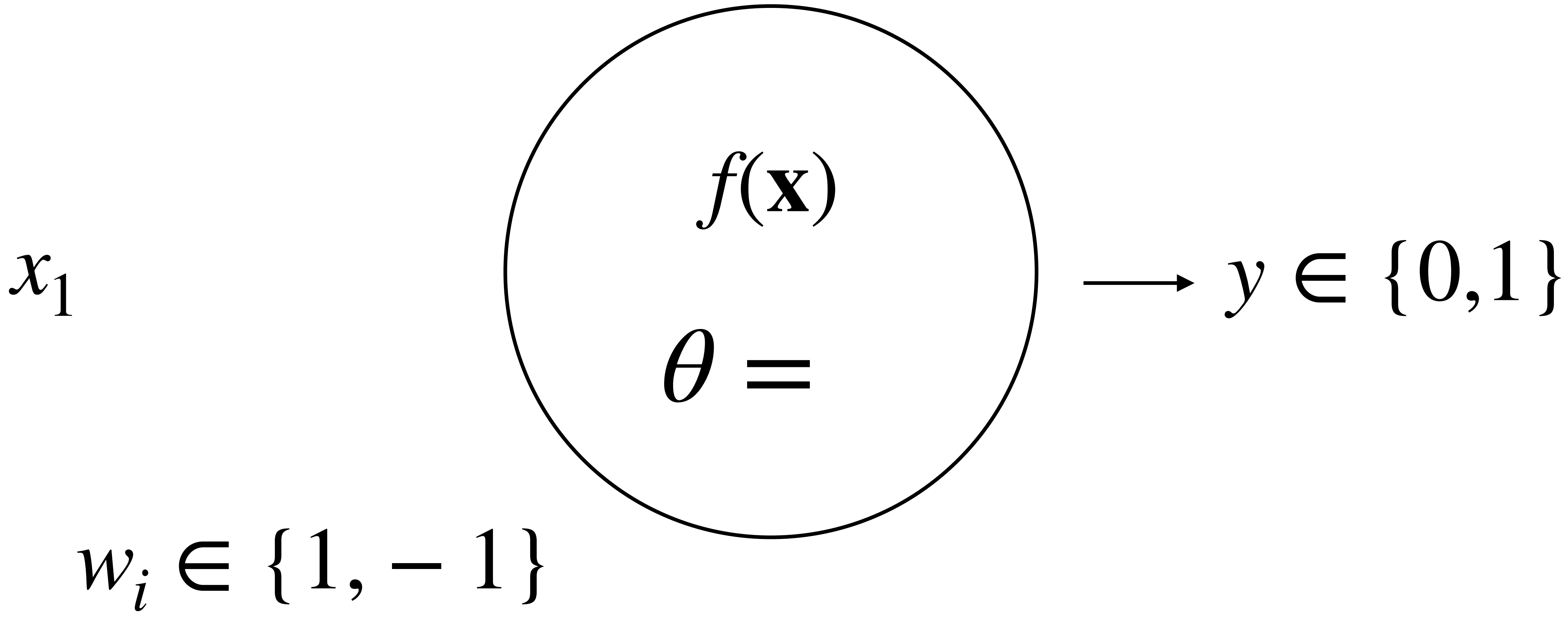
- If the sum of the inputs x weights is greater or equal to some *threshold* θ, then the axon produces the output



McCulloch & Pitts (1943)

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum w_i x_i \geq \theta \\ 0 & \text{else} \end{cases}$$

NOT function

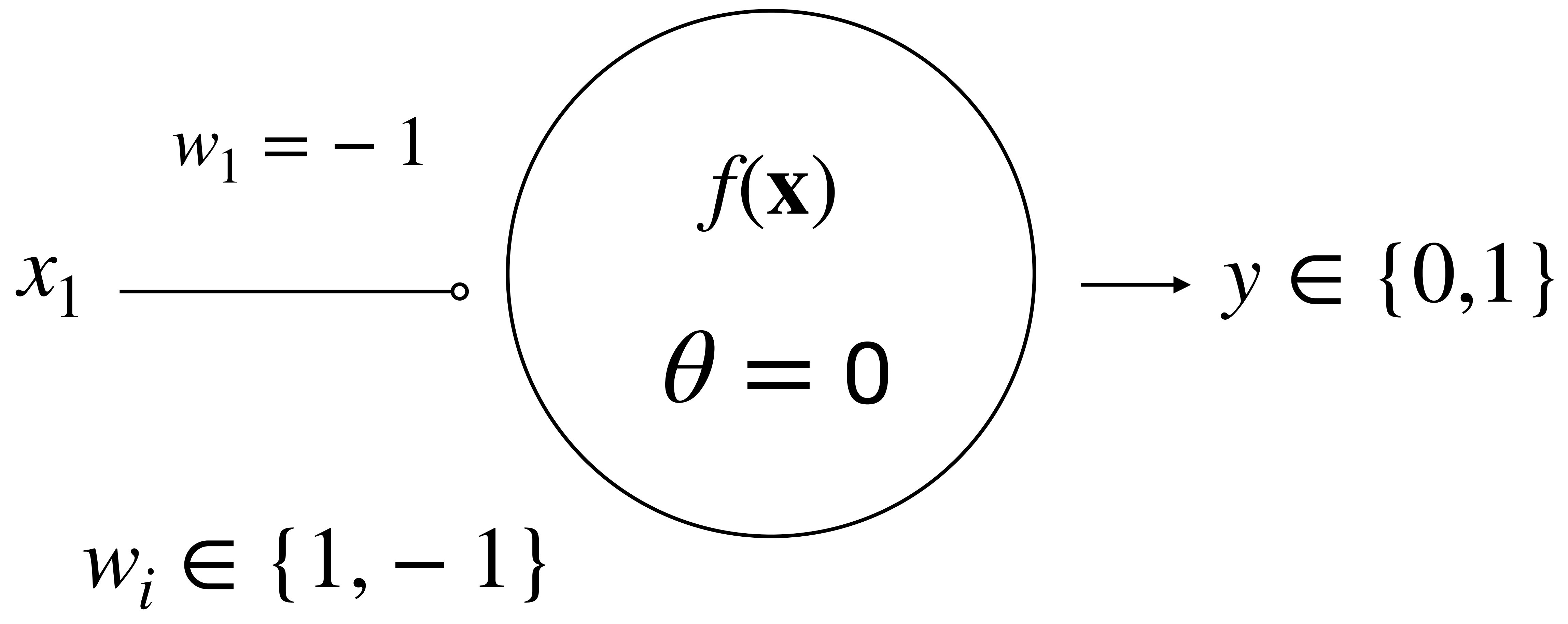


Neuron fires if no inputs are on

McCulloch & Pitts (1943)

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if} \sum w_i x_i \geq \theta \\ 0 & \text{else} \end{cases}$$

NOT function

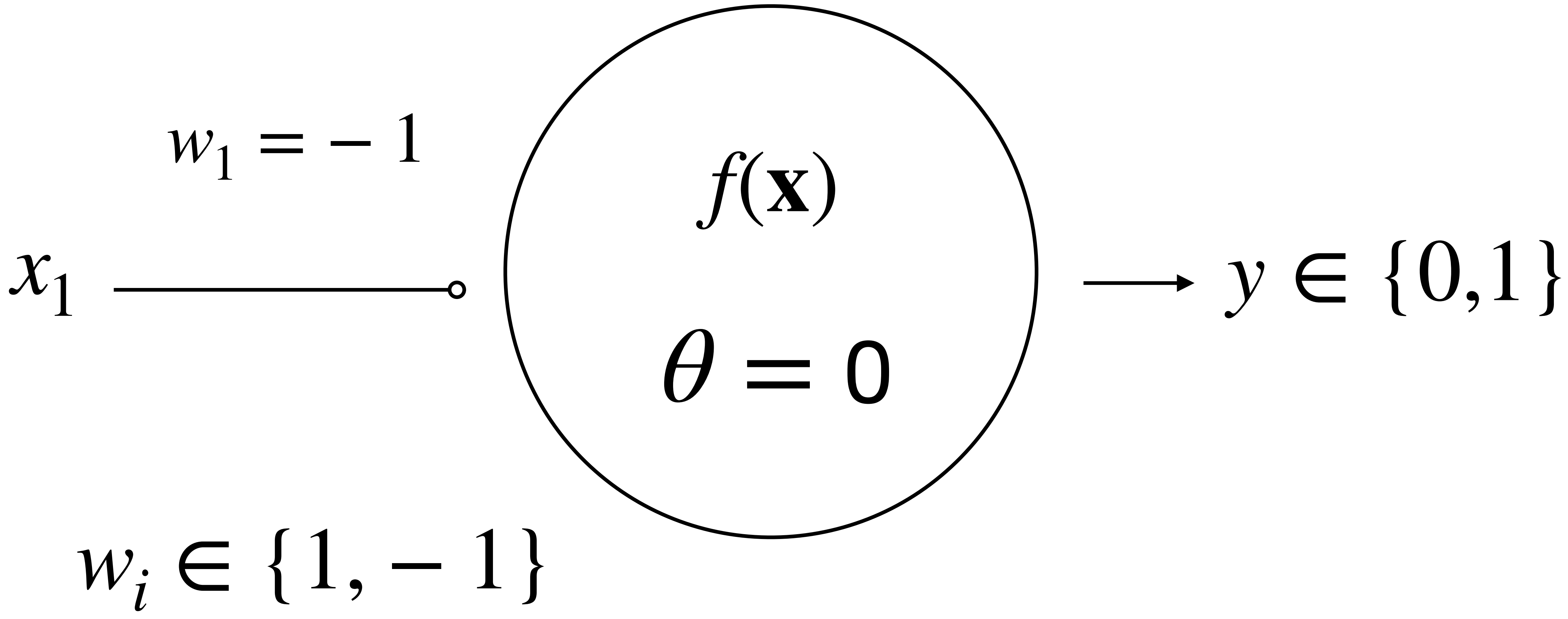


Neuron fires if no inputs are on

McCulloch & Pitts (1943)

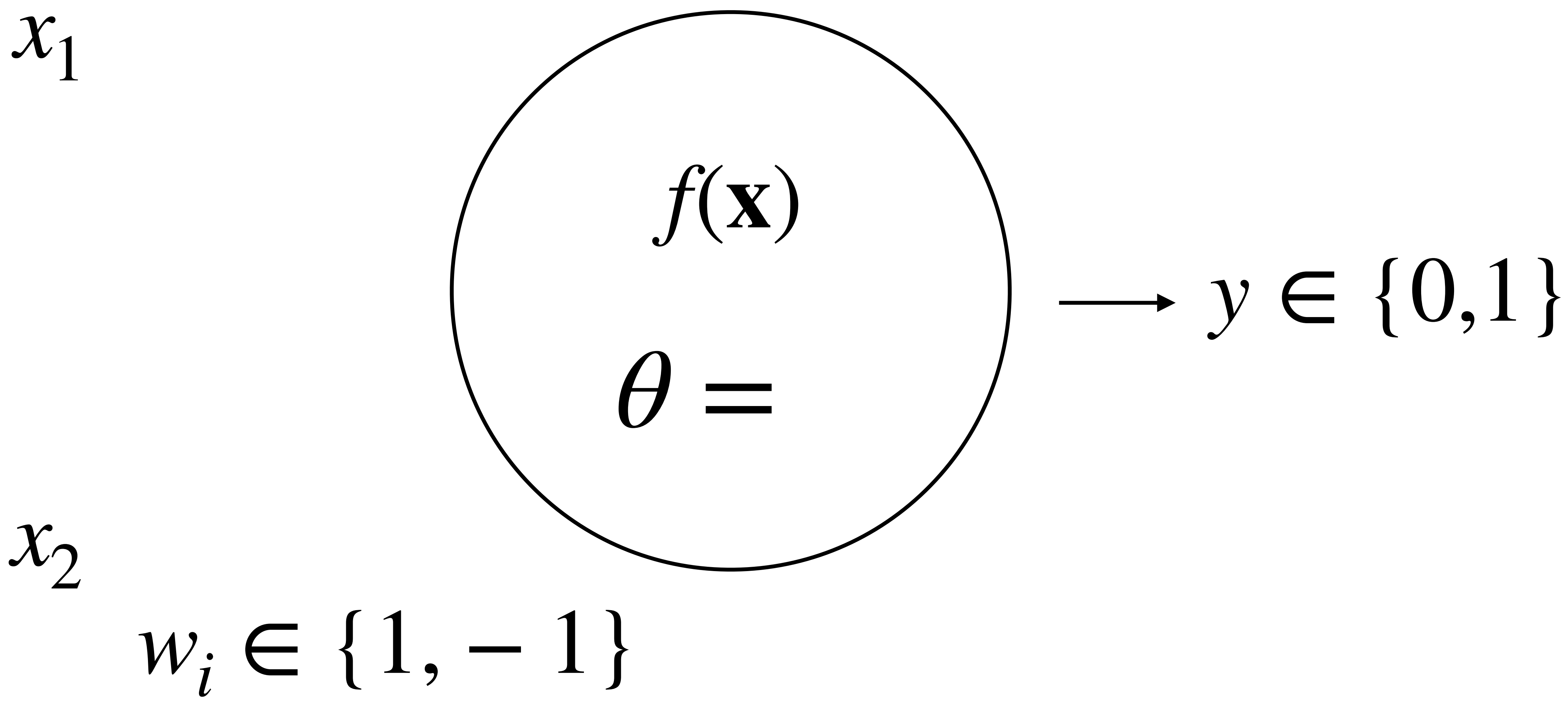
$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum w_i x_i \geq \theta \\ 0 & \text{else} \end{cases}$$

NOT function



Neuron fires if no inputs are on

NAND

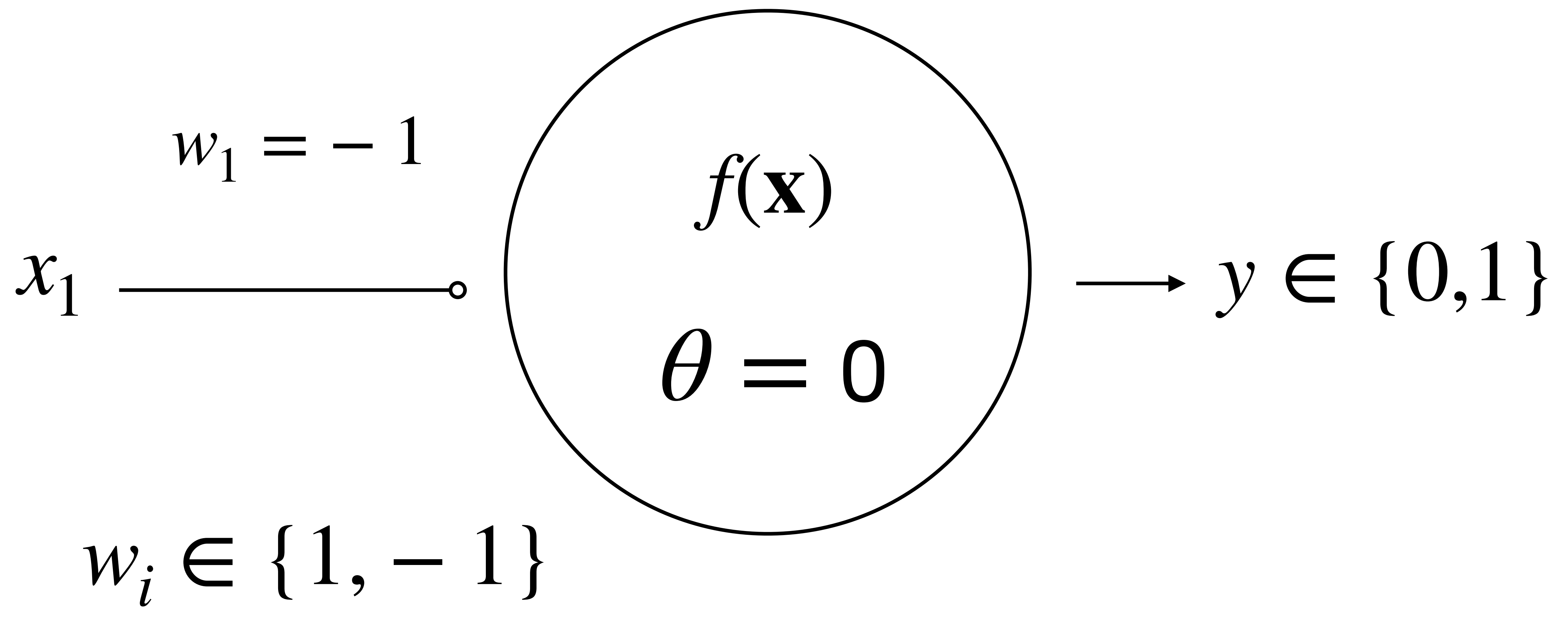


Neuron fires when x_1 is on AND x_2 not on

McCulloch & Pitts (1943)

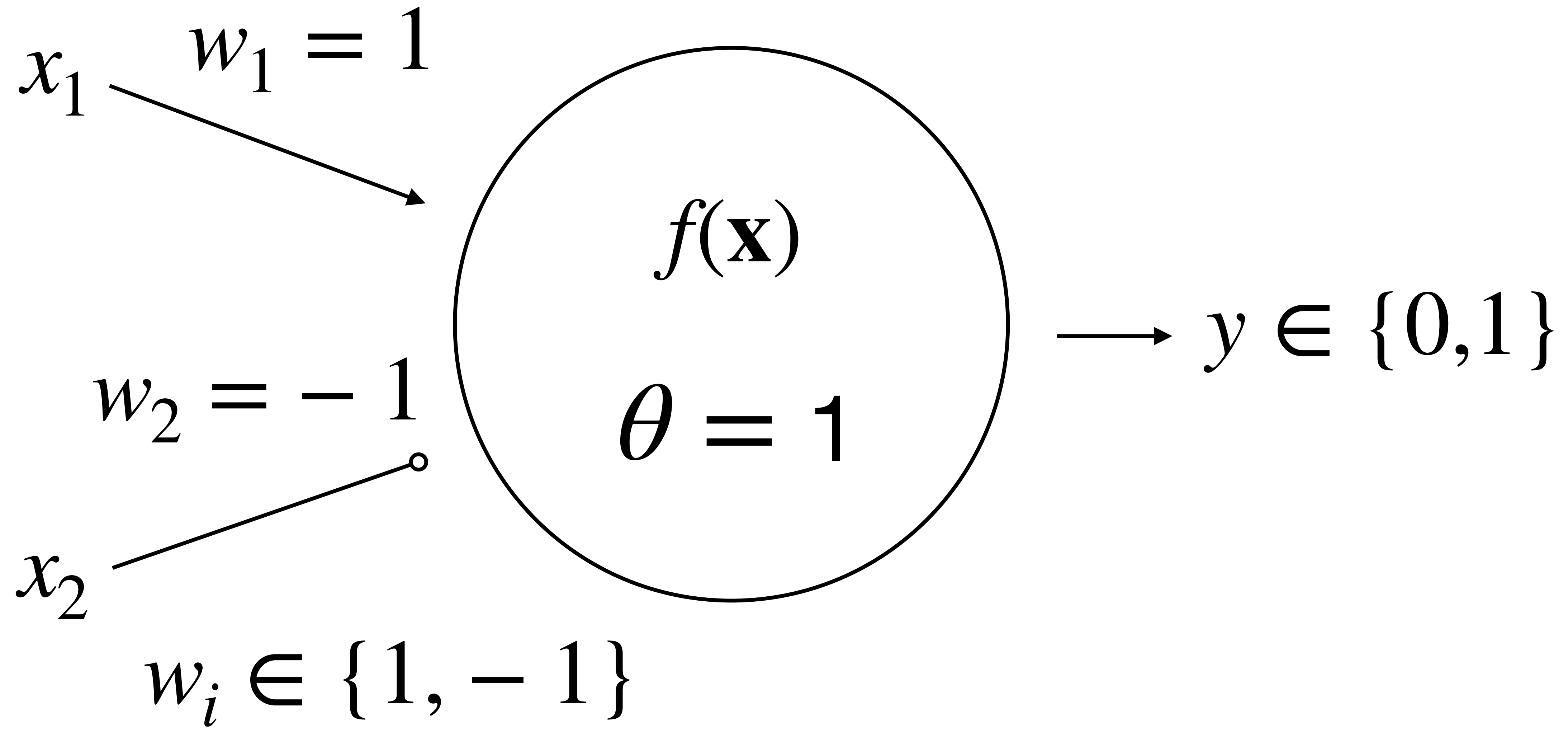
$$f(\mathbf{x}) = \begin{cases} 1 & \text{if} \sum w_i x_i \geq \theta \\ 0 & \text{else} \end{cases}$$

NOT function



Neuron fires if no inputs are on

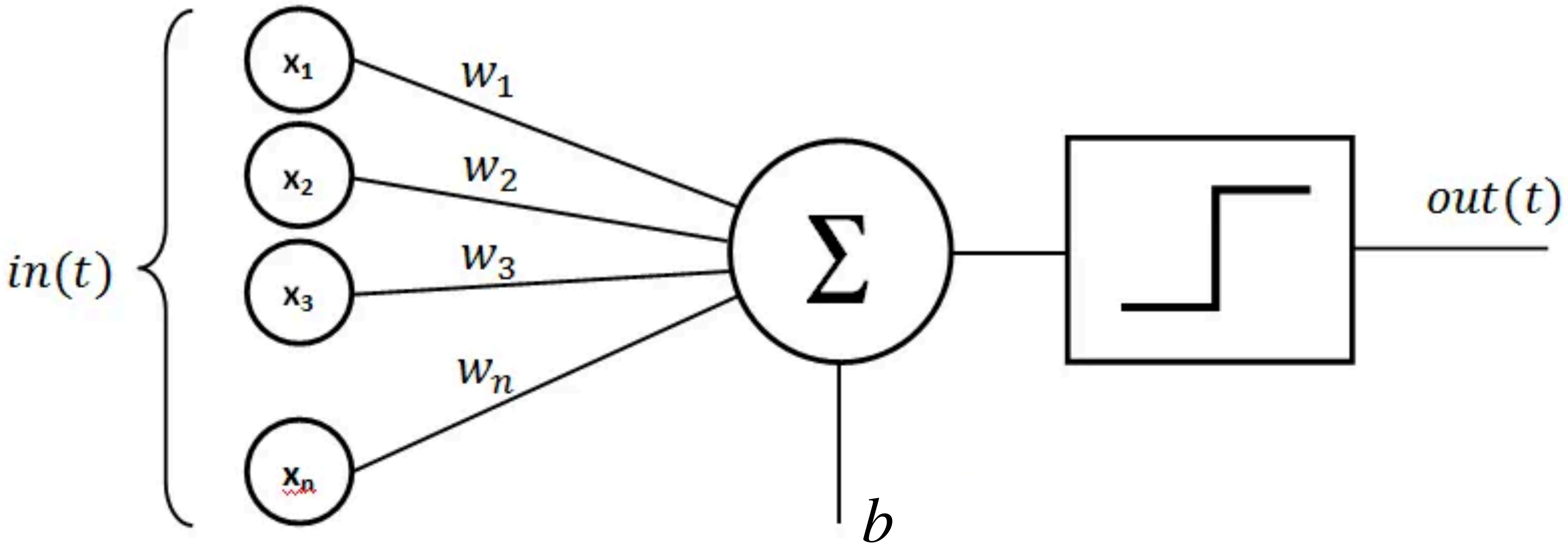
NAND



Neuron fires when x_1 is on AND x_2 not on

Rosenblatt's Perceptron

- Added a learning rule, allowing it to learn any binary classification problem *with linear seperability*
- Very similar to McCulloch & Pitts', but with some key differences:
  - A bias term b is added, effectively replacing θ
  - $\sigma(\mathbf{w}^\top \mathbf{x} + b) = \begin{cases} 1 & \text{if} \quad \mathbf{w}^\top \mathbf{x} + b > 0 \\ 0 & \text{else} \end{cases}$
  - Weights w_i aren't only $\in \{-1,1\}$ but can be any real number
- Weights (and bias) are updated based on error



Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^{m}$.

Initialize $\mathbf{w}$ and b randomly.

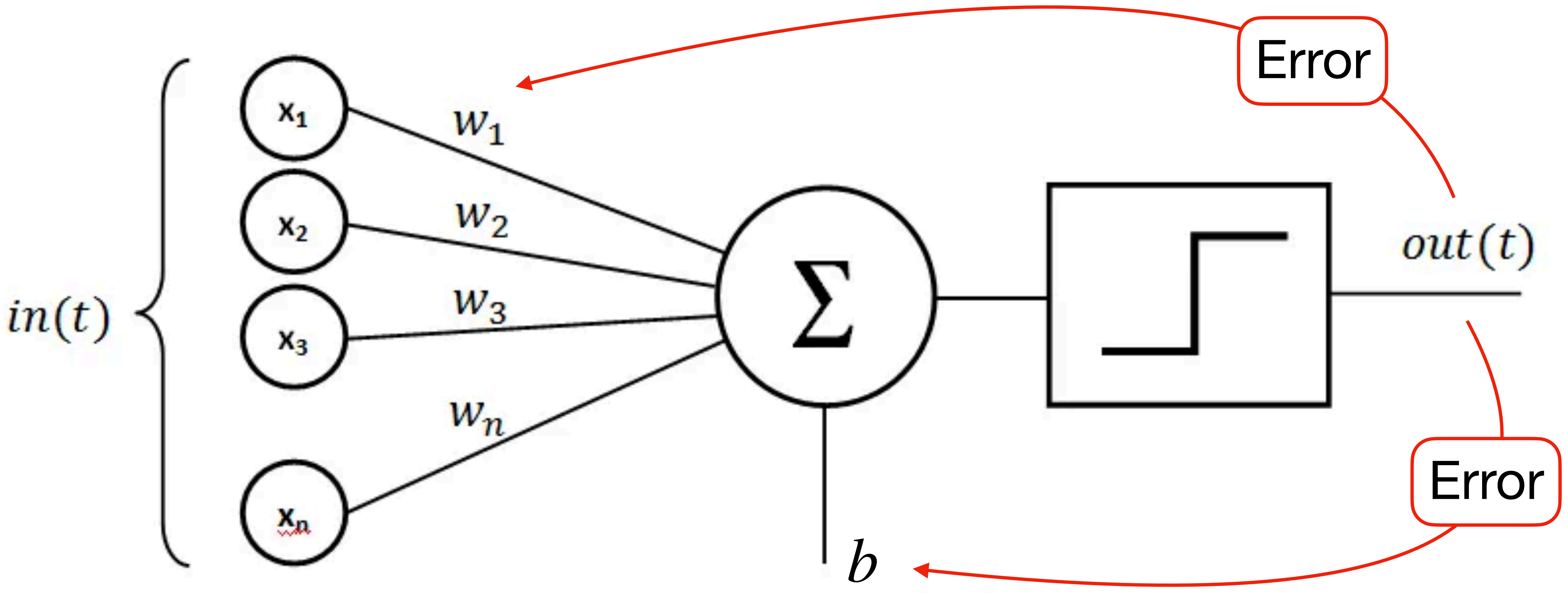
while *not converged* **do**

  # # # Loop through the examples.
  for $j = 1, m$ **do**
    # # # Compare the true label and the prediction.
    $error = y_j - \sigma(\mathbf{w}^T \mathbf{x}_j + b)$
    ### If the model wrongly predicts the class, we update the weights and bias.
    if *error != 0* **then**
      ### Update the weights.
      $\mathbf{w} = \mathbf{w} + error \times x_j$
      ### Update the bias.
      $b = b + error$
  Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.

Rosenblatt's Perceptron

- Added a learning rule, allowing it to learn any binary classification problem *with linear seperability*
- Very similar to McCulloch & Pitts', but with some key differences:
  - A bias term b is added, effectively replacing θ
  - $\sigma(\mathbf{w}^\top \mathbf{x} + b) = \begin{cases} 1 & \text{if} \quad \mathbf{w}^\top \mathbf{x} + b > 0 \\ 0 & \text{else} \end{cases}$
  - Weights w_i aren't only $\in \{-1, 1\}$ but can be any real number
- Weights (and bias) are updated based on error



Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^m$.

Initialize $\mathbf{w}$ and b randomly.

while *not converged* **do**

  ### Loop through the examples.
  for $j = 1, m$ **do**
    ### Compare the true label and the prediction.
    $error = y_j - \sigma(\mathbf{w}^T \mathbf{x}_j + b)$
    ### If the model wrongly predicts the class, we update the weights and bias.
    if *error != 0* **then**
      ### Update the weights.
      $\mathbf{w} = \mathbf{w} + error \times x_j$
      ### Update the bias.
      $b = b + error$
  Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.

Pablo Caceres

Perceptron learning rule

Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^{m}$.

Initialize $\mathbf{w}$ and b randomly.

while *not converged* **do**

    # # # Loop through the examples.
    for $j = 1, m$ **do**

        # # # Compare the true label and the prediction.
        $error = y_j - \sigma(\mathbf{w}^T \mathbf{x}_j + b)$

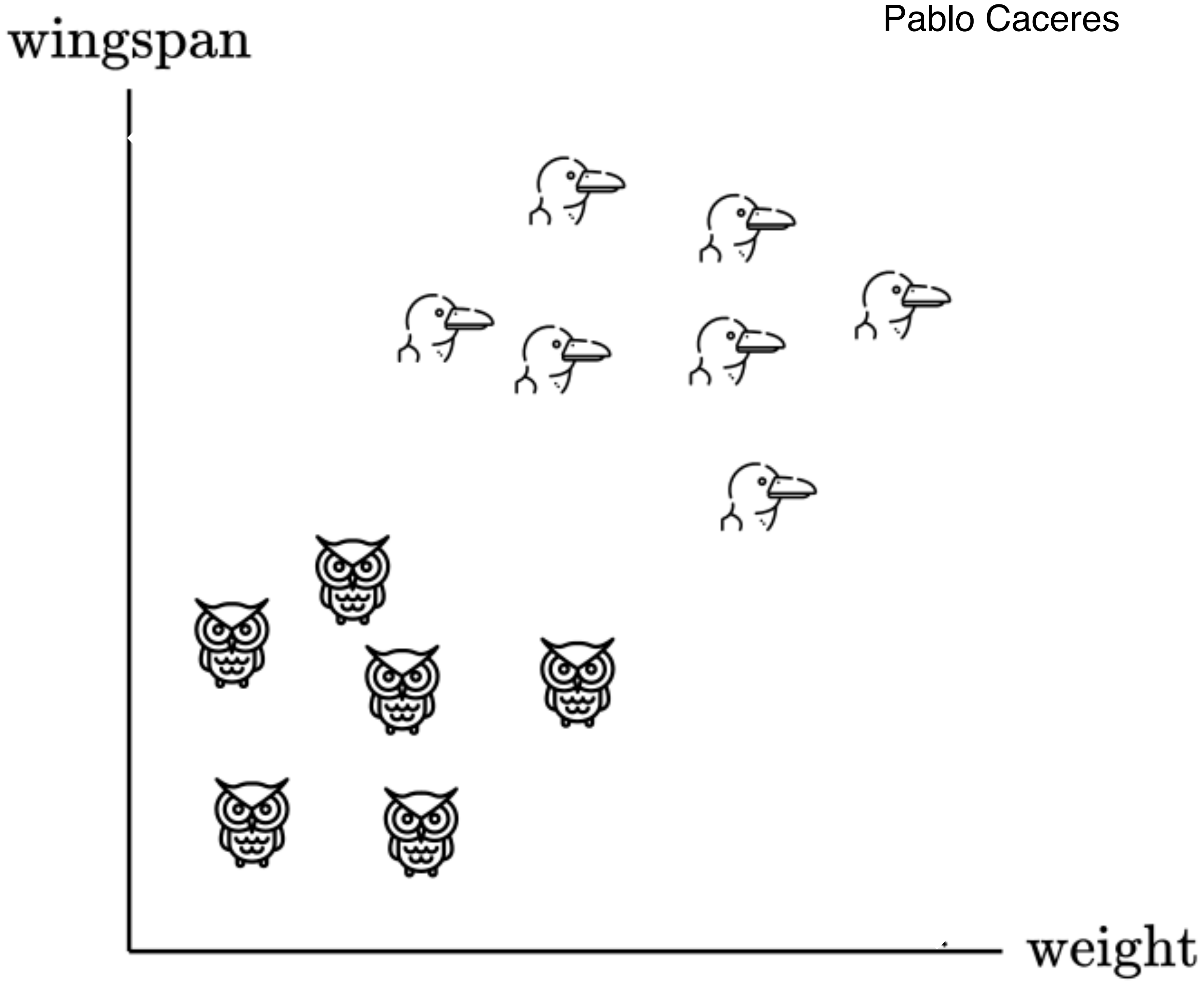
        ### If the model wrongly predicts the class, we update the weights and bias.
        if *error != 0* **then**

            ### Update the weights.
            $\mathbf{w} = \mathbf{w} + error \times x_j$

            ### Update the bias.
            $b = b + error$

    Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.



Pablo Caceres

Perceptron learning rule

Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^{m}$. (weight, wingspan)

Initialize $\mathbf{w}$ and b randomly.

while *not converged* **do**

    # # # Loop through the examples.
    for $j = 1, m$ **do**

        # # # Compare the true label and the prediction.
        $error = y_j - \sigma(\mathbf{w}^T\mathbf{x}_j + b)$

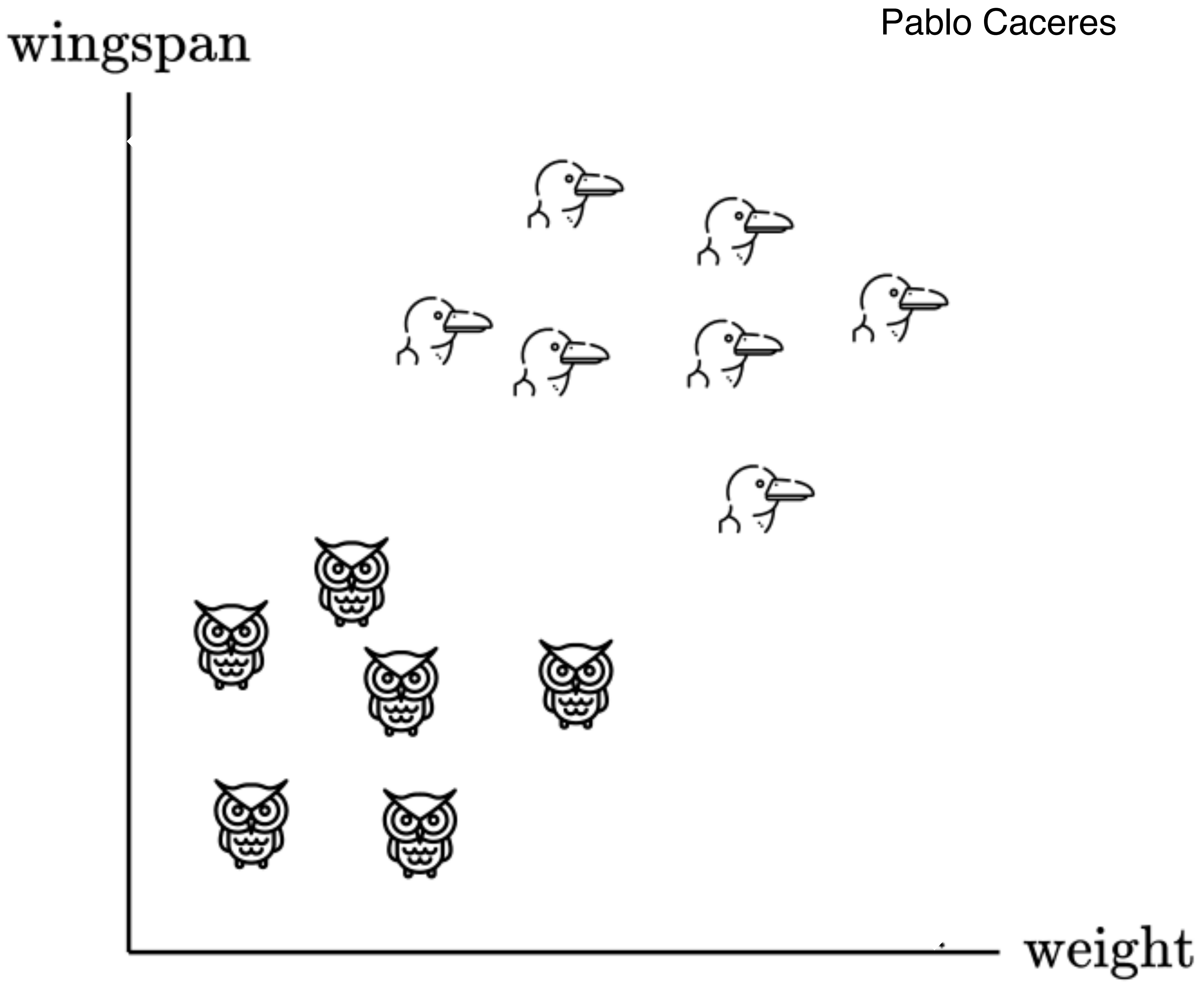
        ### If the model wrongly predicts the class, we update the weights and bias.
        if *error != 0* **then**

            ### Update the weights.
            $\mathbf{w} = \mathbf{w} + error \times x_j$

            ### Update the bias.
            $b = b + error$

    Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.



Perceptron learning rule

Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^{m}$. (weight, wingspan) Owl=0 vs. Albatross=1

Initialize $\mathbf{w}$ and b randomly.

while *not converged* **do**

  # # # Loop through the examples.
  for $j = 1, m$ **do**

    # # # Compare the true label and the prediction.
    $error = y_j - \sigma(\mathbf{w}^T\mathbf{x}_j + b)$

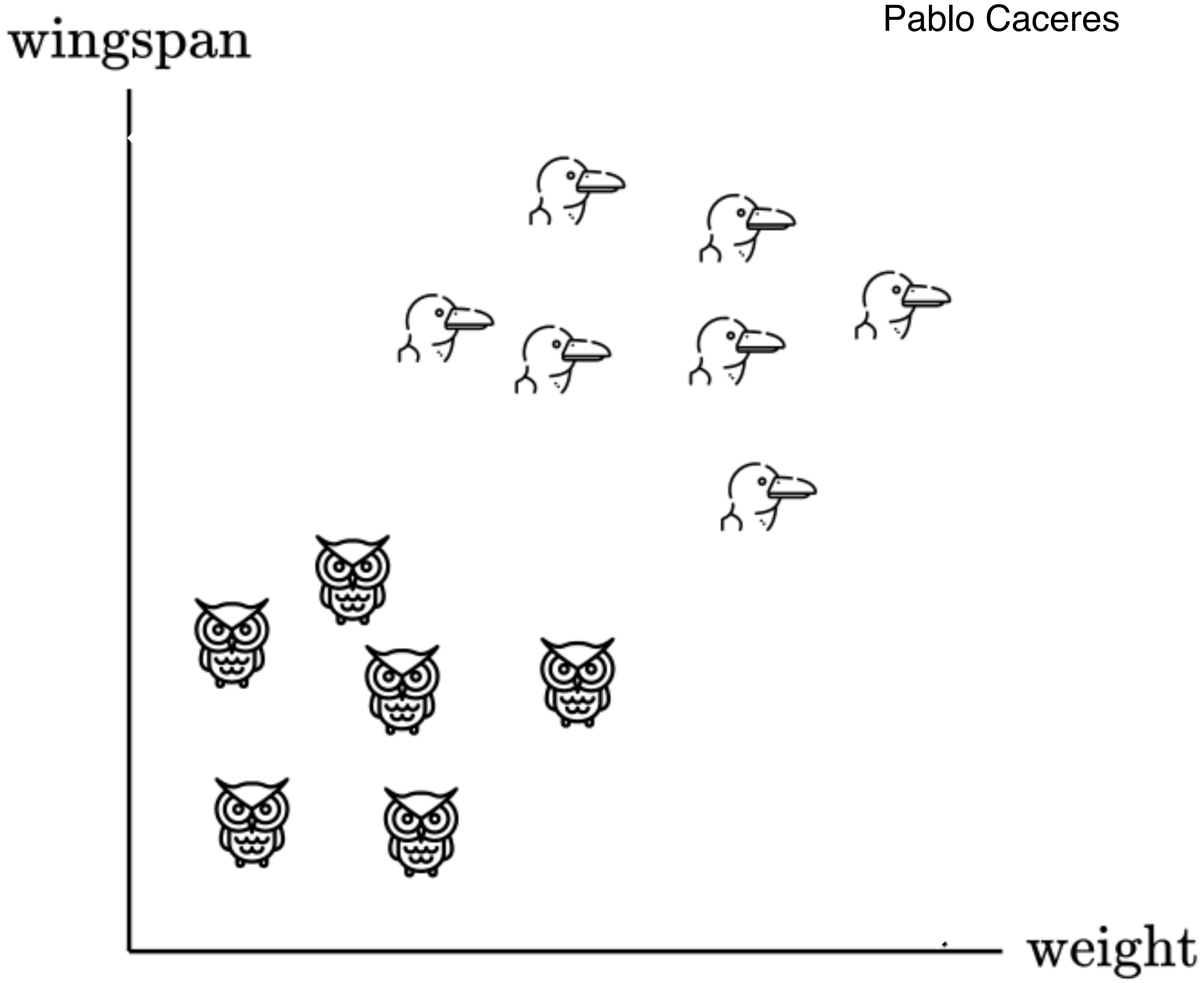
    ### If the model wrongly predicts the class, we update the weights and bias.
    if *error != 0* **then**

      ### Update the weights.
      $\mathbf{w} = \mathbf{w} + error \times x_j$

      ### Update the bias.
      $b = b + error$

  Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.



Pablo Caceres

Perceptron learning rule

Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^{m}$. (weight, wingspan) Owl=0 vs. Albatross=1

Initialize $\mathbf{w}$ and b randomly.

while *not converged* **do**

Loop through the examples.

for $j = 1, m$ **do**

Compare the true label and the prediction.

$error = y_j - \sigma(\mathbf{w}^T \mathbf{x}_j + b)$

If the model wrongly predicts the class, we update the weights and bias.

if *error != 0* **then**

Update the weights.

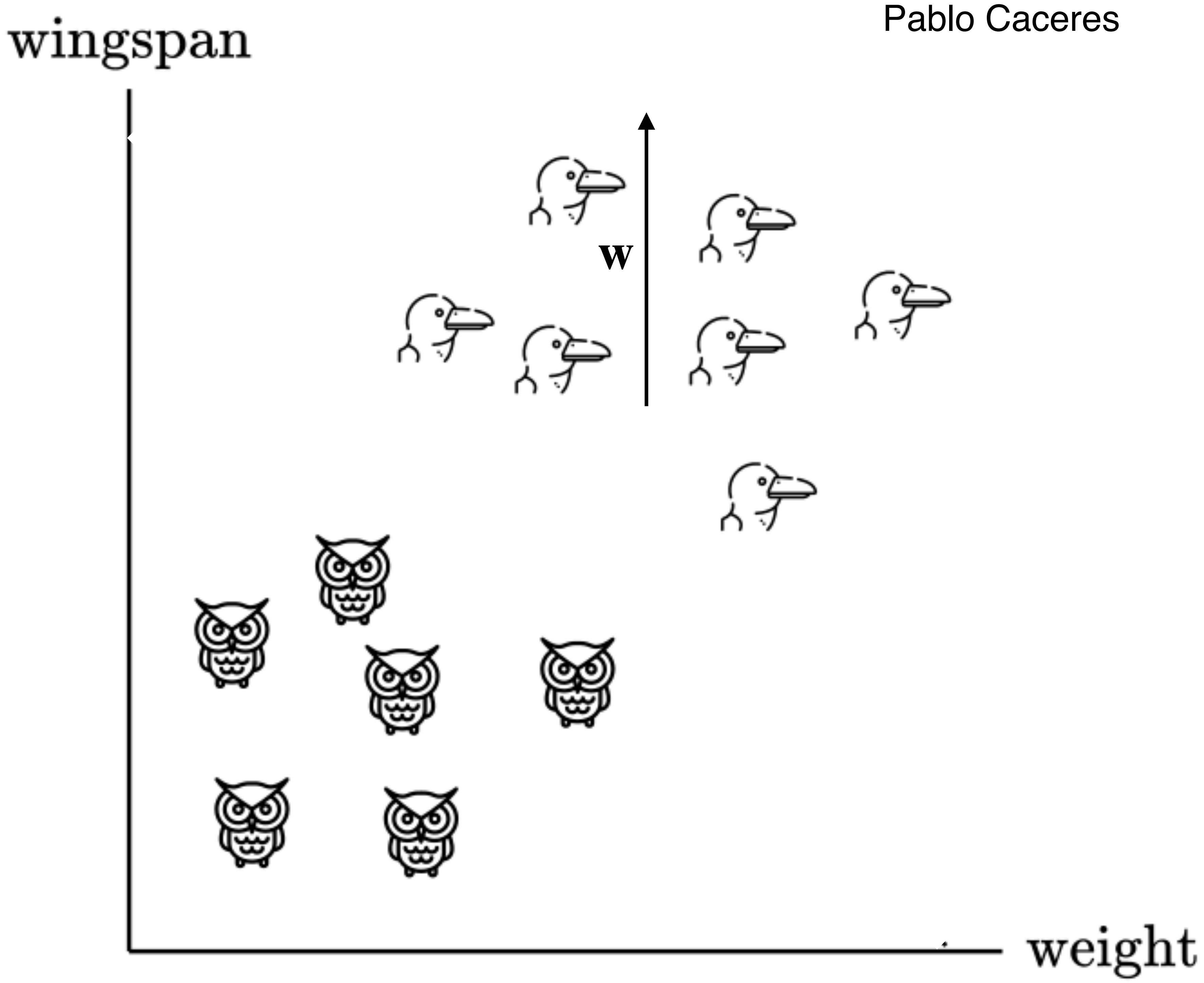
$\mathbf{w} = \mathbf{w} + error \times x_j$

Update the bias.

$b = b + error$

Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.



Pablo Caceres

Perceptron learning rule

Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^{m}$. (weight, wingspan) Owl=0 vs. Albatross=1

Initialize $\mathbf{w}$ and b randomly.

while *not converged* **do**

    # # # Loop through the examples.
    for $j = 1, m$ **do**

        # # # Compare the true label and the prediction.
        $error = y_j - \sigma(\mathbf{w}^T\mathbf{x}_j + b)$

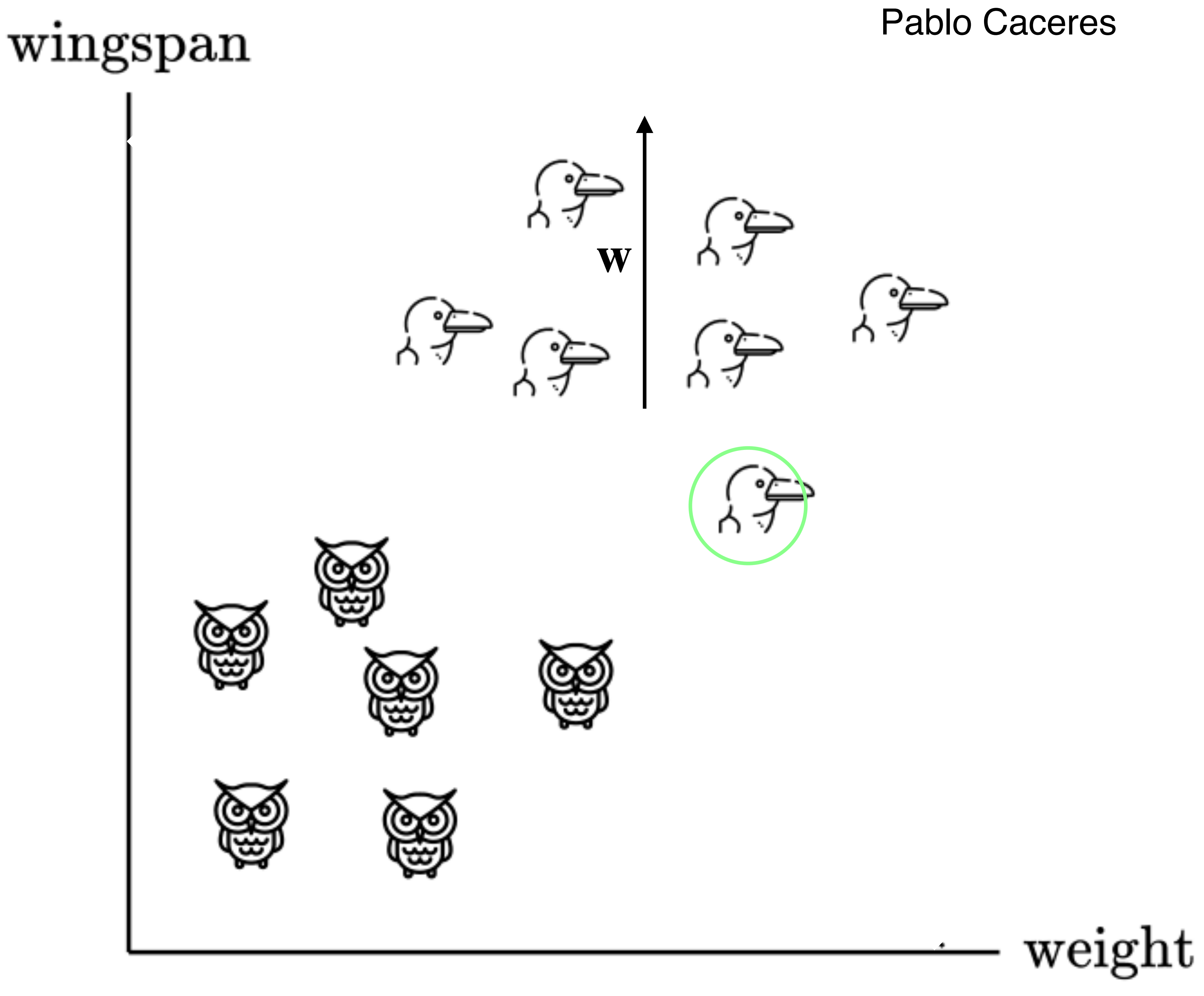
        ### If the model wrongly predicts the class, we update the weights and bias.
        if *error != 0* **then**

            ### Update the weights.
            $\mathbf{w} = \mathbf{w} + error \times x_j$

            ### Update the bias.
            $b = b + error$

    Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.



Pablo Caceres

Perceptron learning rule

Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^{m}$. (weight, wingspan) Owl=0 vs. Albatross=1

Initialize $\mathbf{w}$ and b randomly.

while *not converged* **do**

  # # # Loop through the examples.
  for $j = 1, m$ **do**

    # # # Compare the true label and the prediction.
    $error = y_j - \sigma(\mathbf{w}^T \mathbf{x}_j + b)$

    ### If the model wrongly predicts the class, we update the weights and bias.
    if *error != 0* **then**

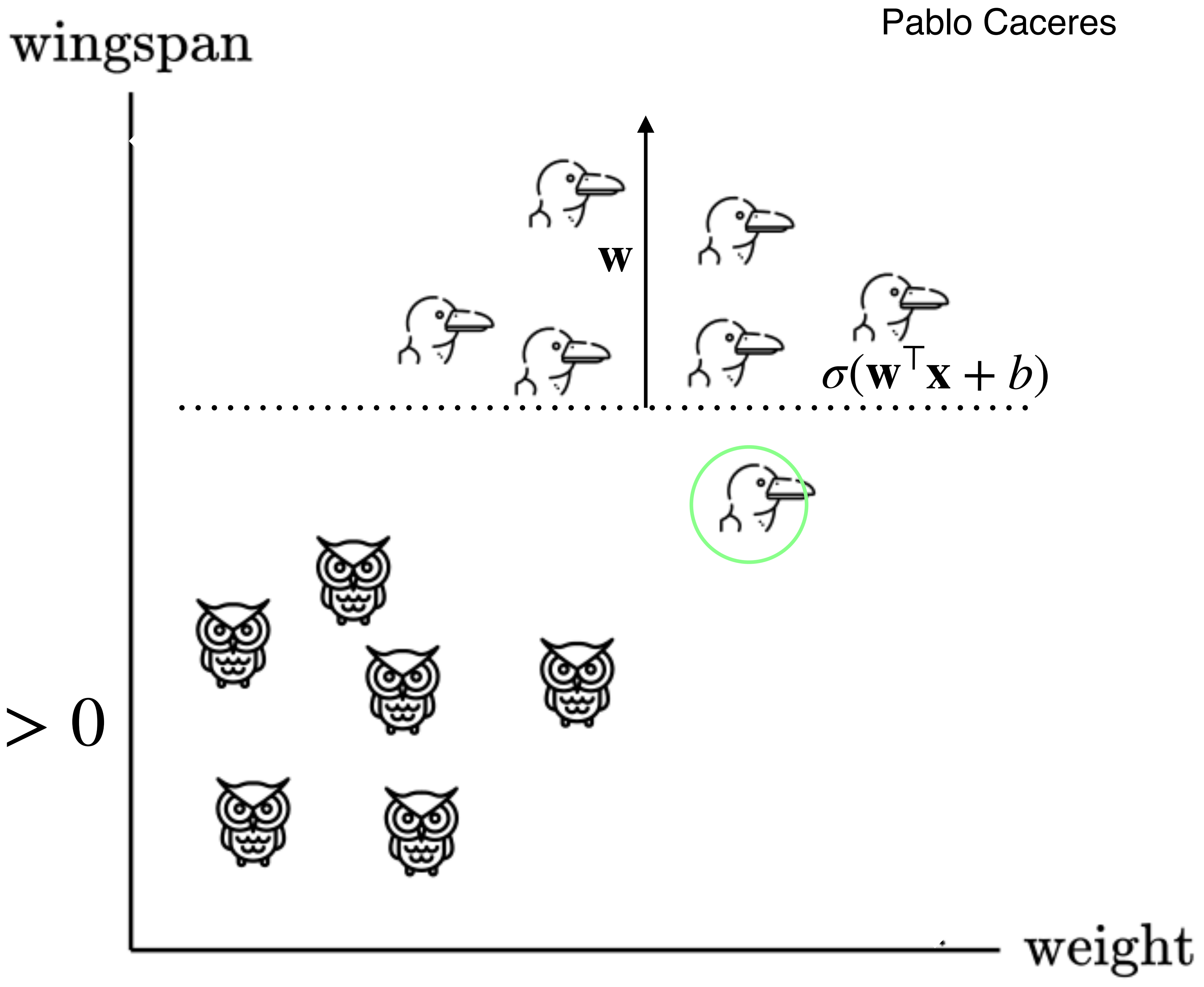
      ### Update the weights.
      $\mathbf{w} = \mathbf{w} + error \times x_j$

      ### Update the bias.
      $b = b + error$

  Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.

$$\sigma(\mathbf{w}^\top \mathbf{x} + b) = \begin{cases} 1 & \text{if} \quad \mathbf{w}^\top \mathbf{x} + b > 0 \\ 0 & \text{else} \end{cases}$$



Pablo Caceres

Perceptron learning rule

Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^{m}$. (weight, wingspan) Owl=0 vs. Albatross=1

Initialize $\mathbf{w}$ and b randomly.

while *not converged* **do**

  # # # Loop through the examples.

  for $j = 1, m$ **do**

    # # # Compare the true label and the prediction.

    $error = y_j - \sigma(\mathbf{w}^T\mathbf{x}_j + b)$

    ### If the model wrongly predicts the class, we update the weights and bias.

    if *error != 0* **then**

      ### Update the weights.

      $\mathbf{w} = \mathbf{w} + error \times x_j$

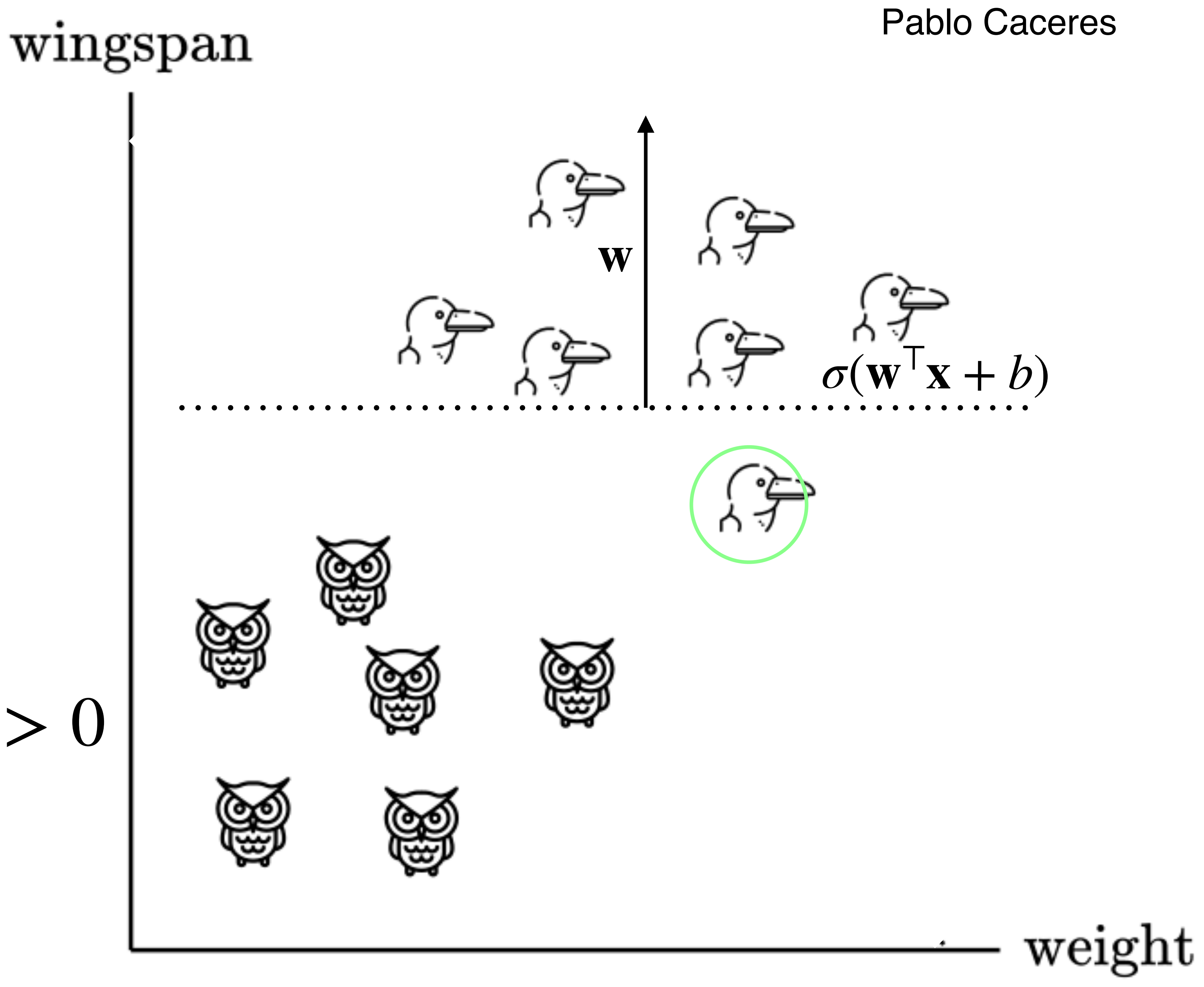
      ### Update the bias.

      $b = b + error$

  Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.

$$\sigma(\mathbf{w}^\top\mathbf{x} + b) = \begin{cases} 1 & \text{if} \quad \mathbf{w}^\top\mathbf{x} + b > 0 \\ 0 & \text{else} \end{cases}$$



Pablo Caceres

Perceptron learning rule

Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^{m}$. (weight, wingspan) Owl=0 vs. Albatross=1

Initialize $\mathbf{w}$ and b randomly.

while *not converged* **do**

  # # # Loop through the examples.

  for $j = 1, m$ **do**

    # # # Compare the true label and the prediction.

    $error = y_j - \sigma(\mathbf{w}^T \mathbf{x}_j + b)$

    ### If the model wrongly predicts the class, we update the weights and bias.

    if *error != 0* **then**

      ### Update the weights.

      $\mathbf{w} = \mathbf{w} + error \times x_j$

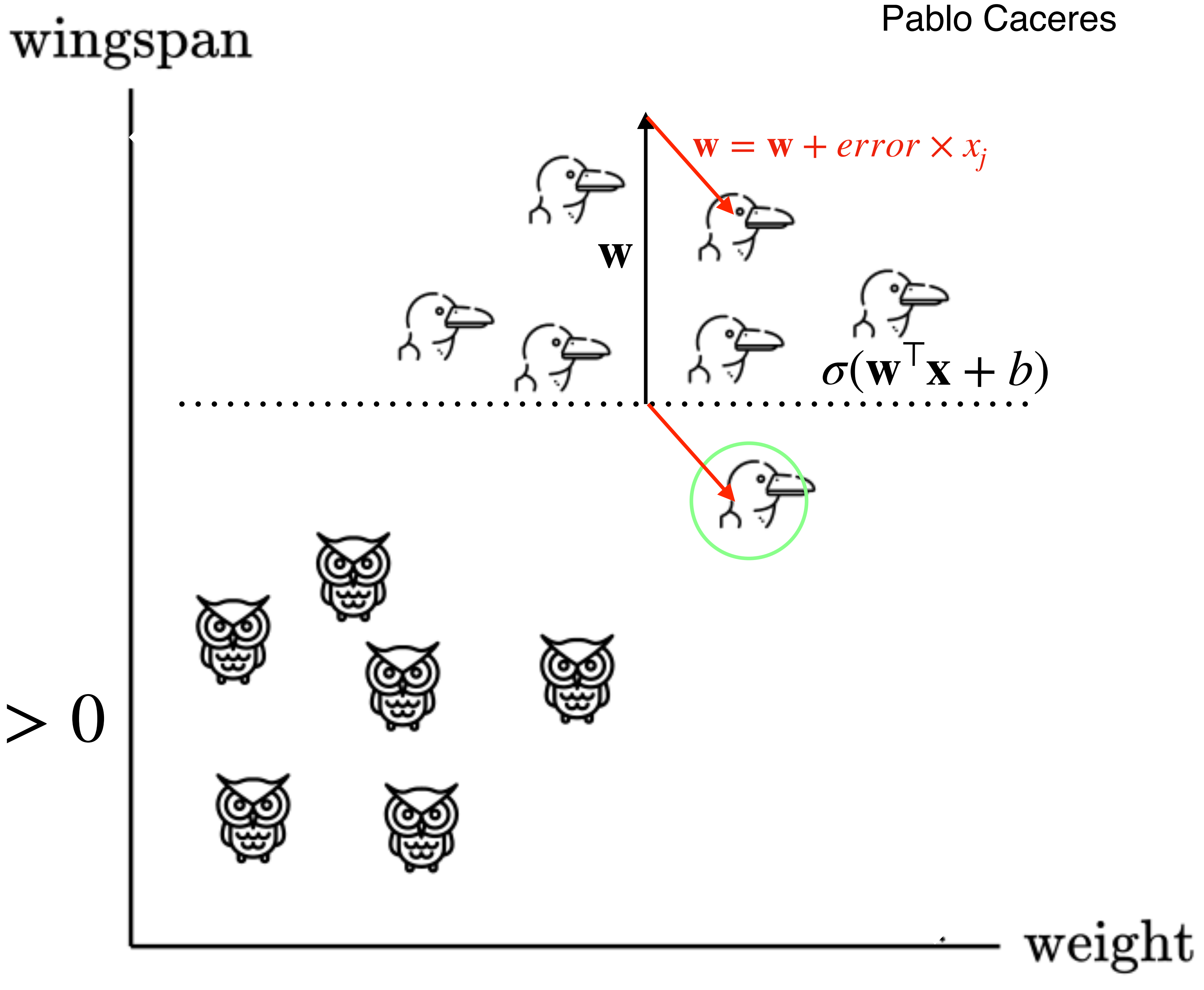
      ### Update the bias.

      $b = b + error$

  Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.

$$\sigma(\mathbf{w}^\top \mathbf{x} + b) = \begin{cases} 1 & \text{if} \quad \mathbf{w}^\top \mathbf{x} + b > 0 \\ 0 & \text{else} \end{cases}$$



Perceptron learning rule

Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^{m}$. (weight, wingspan) Owl=0 vs. Albatross=1

Initialize $\mathbf{w}$ and b randomly.

while *not converged* **do**

    # # # Loop through the examples.

    for $j = 1, m$ **do**

        # # # Compare the true label and the prediction.

        $error = y_j - \sigma(\mathbf{w}^T \mathbf{x}_j + b)$

        ### If the model wrongly predicts the class, we update the weights and bias.

        if *error != 0* **then**

            ### Update the weights.

            $\mathbf{w} = \mathbf{w} + error \times x_j$

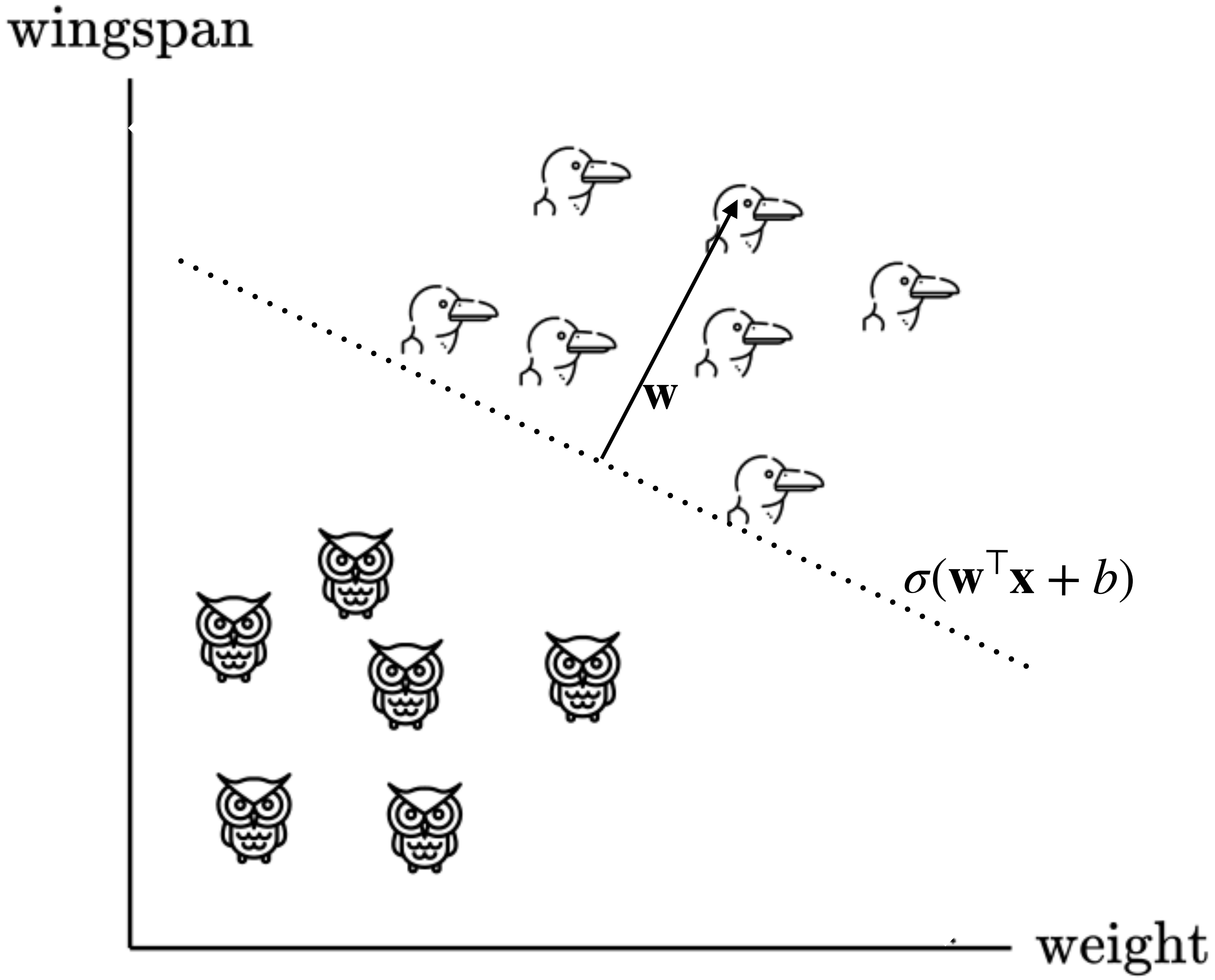
            ### Update the bias.

            $b = b + error$

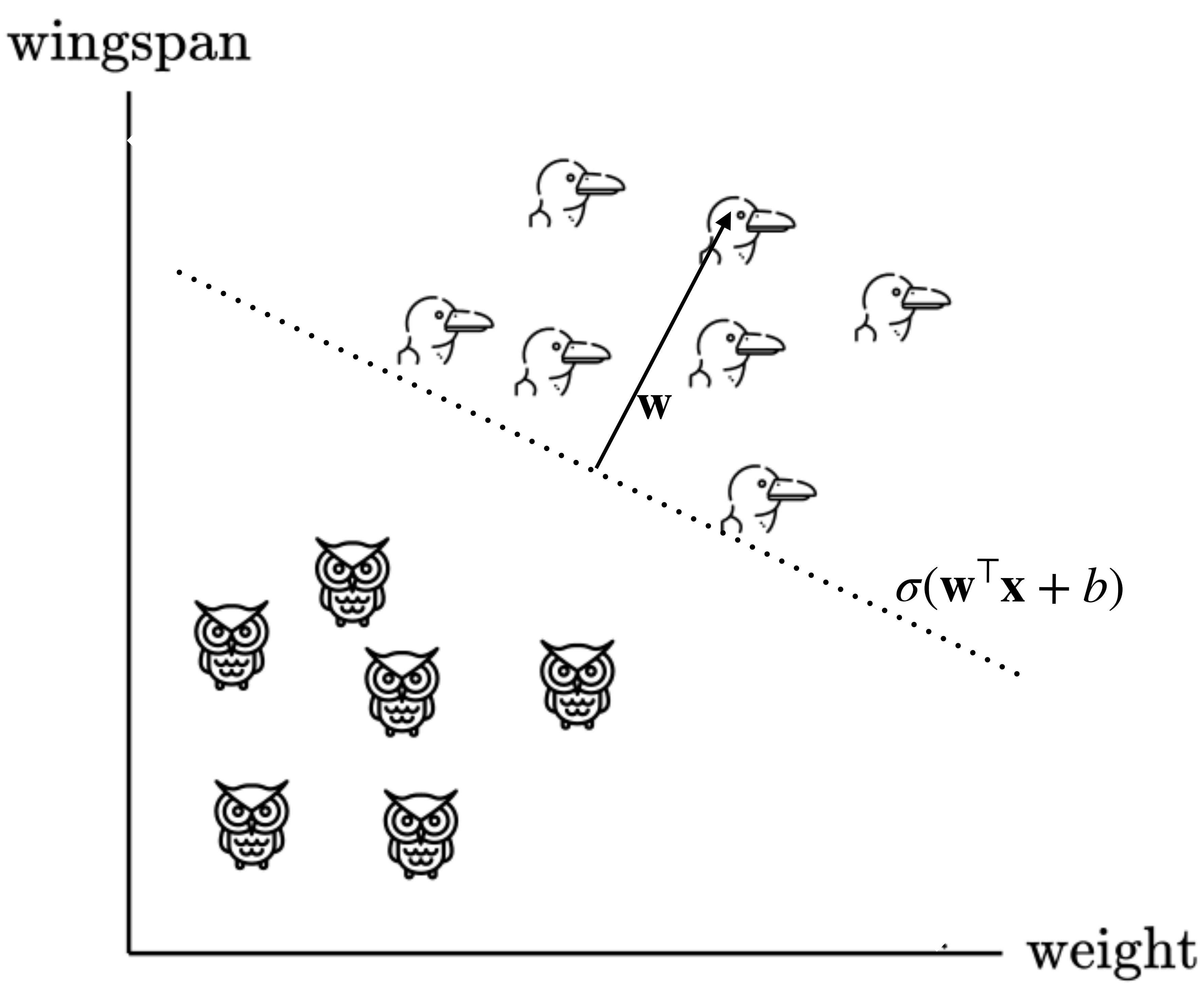
    Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.

$$\sigma(\mathbf{w}^\top \mathbf{x} + b) = \begin{cases} 1 & \text{if} \quad \mathbf{w}^\top \mathbf{x} + b > 0 \\ 0 & \text{else} \end{cases}$$



Perceptron learning rule



Algorithm 1: Perceptron Learning Algorithm

Input: Training examples $\{\mathbf{x}_i, y_i\}_{i=1}^m$. (weight, wingspan) Owl=0 vs. Albatross=1

Initialize $\mathbf{w}$ and b randomly.

while *not converged* **do**

    # # # Loop through the examples.
    for $j = 1, m$ **do**

        # # # Compare the true label and the prediction.
        $error = y_j - \sigma(\mathbf{w}^T \mathbf{x}_j + b)$

        ### If the model wrongly predicts the class, we update the weights and bias.
        if *error != 0* **then**

            ### Update the weights.
            $\mathbf{w} = \mathbf{w} + error \times x_j$

            ### Update the bias.
            $b = b + error$

    Test for convergence

Output: Set of weights $\mathbf{w}$ and bias b for the perceptron.

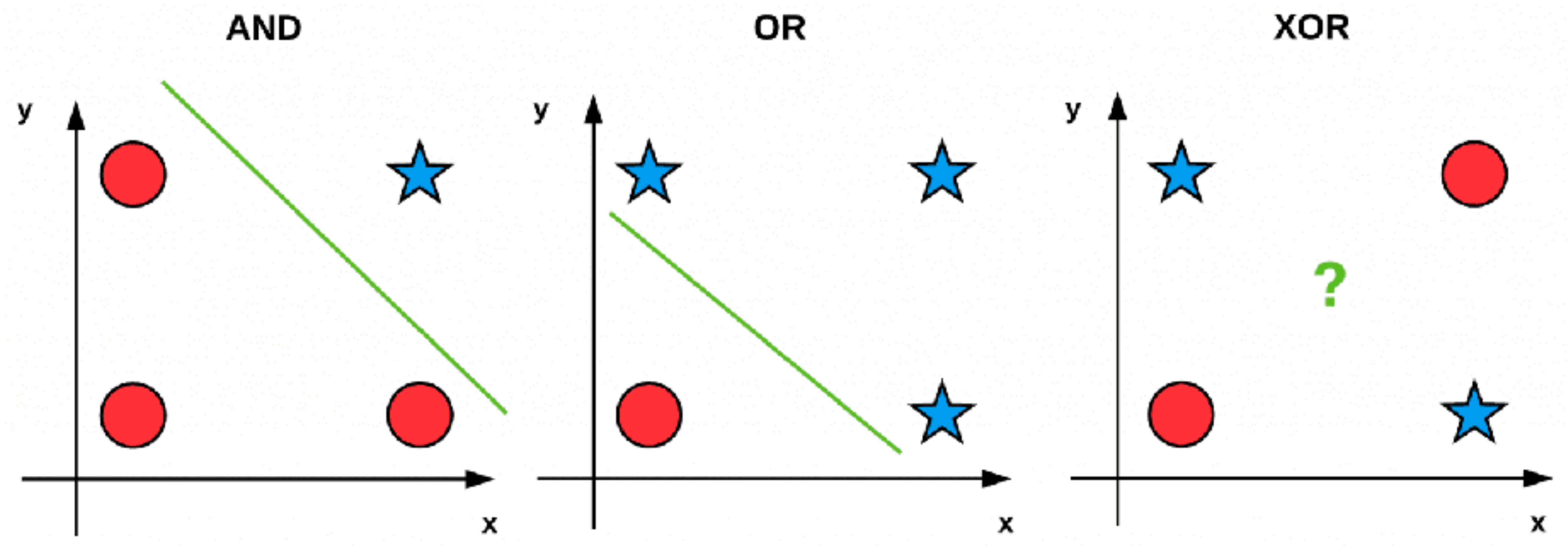
$$\sigma(\mathbf{w}^\top \mathbf{x} + b) = \begin{cases} 1 & \text{if} \quad \mathbf{w}^\top \mathbf{x} + b > 0 \\ 0 & \text{else} \end{cases}$$

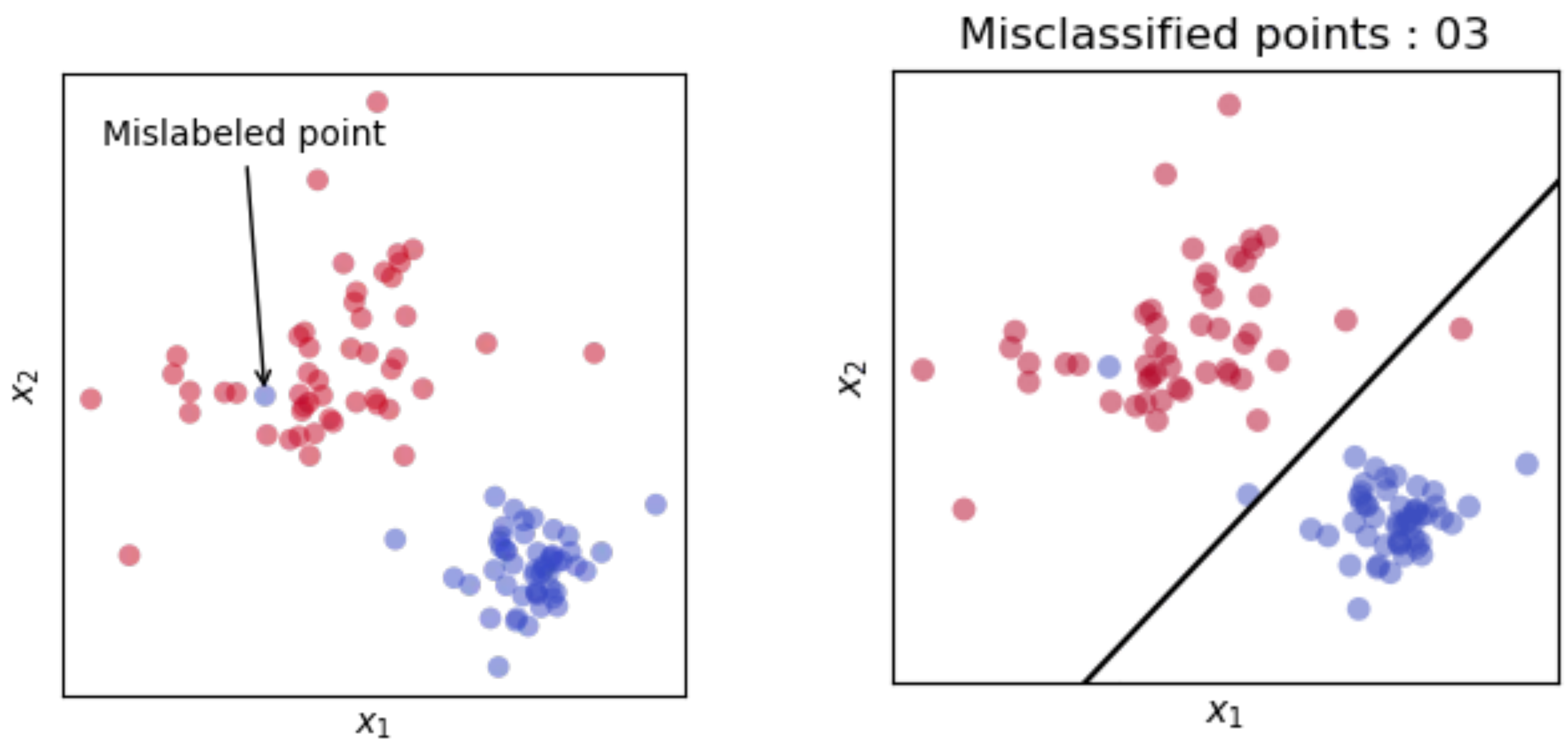
Guaranteed to converge if data is linearly separable

Limitations of linear separability

Adrian Rosebrock

- The perceptron can learn any linearly separable problem
  - But not all problems are lineary separable
- Even a single mislabeled data point in the data will throw the algorithm into chaos
- Enter the XOR problem and Minsky & Parpert (1969) critique
  - Argument: because a single neuron is unable to solve XOR, larger networks will also have similar problems
  - Therefore, the research program should be dropped

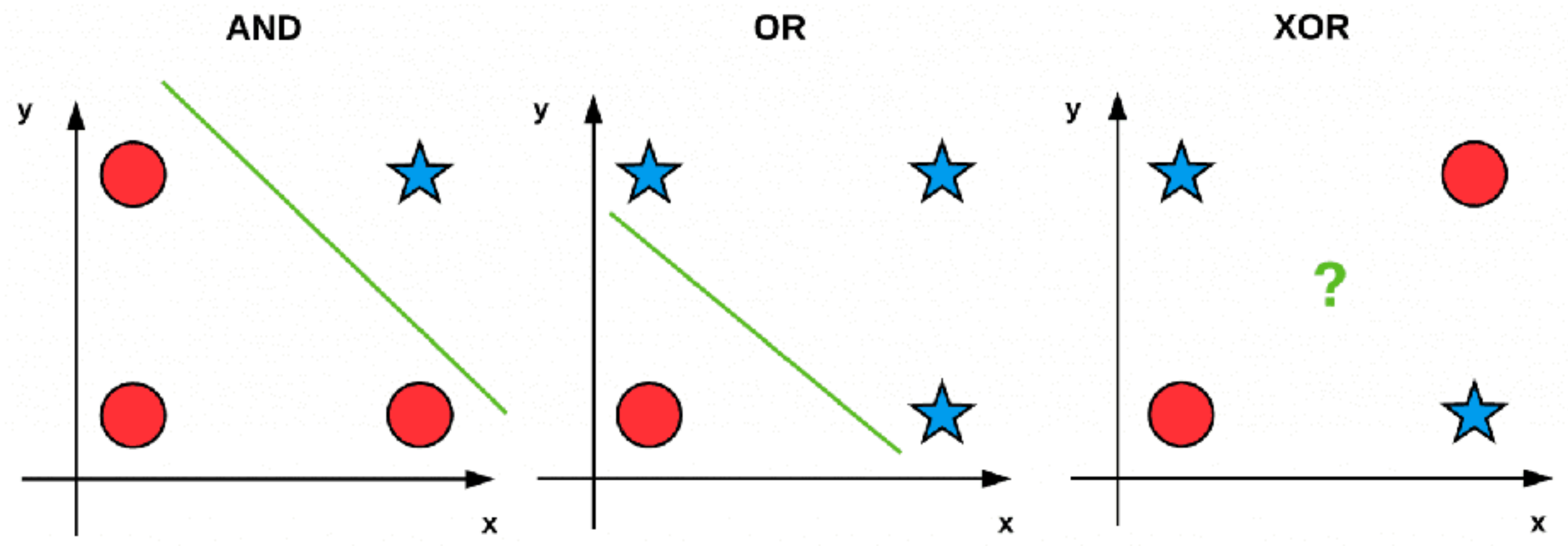


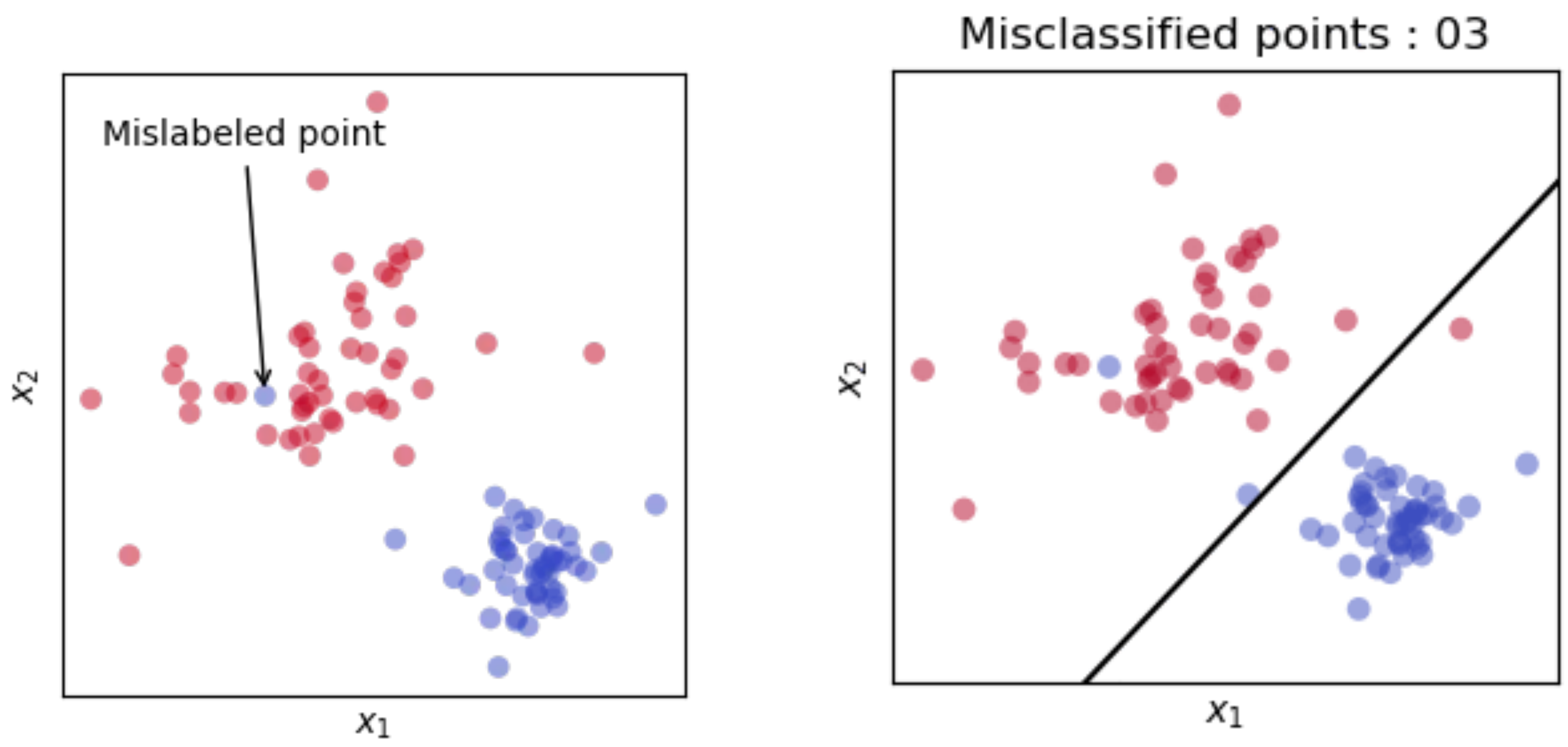


Limitations of linear separability

Adrian Rosebrock

- The perceptron can learn any linearly separable problem
  - But not all problems are lineary separable
- Even a single mislabeled data point in the data will throw the algorithm into chaos
- Enter the XOR problem and Minsky & Parpert (1969) critique
  - Argument: because a single neuron is unable to solve XOR, larger networks will also have similar problems
  - Therefore, the research program should be dropped





Addressing Minsky & Parpert's critiques

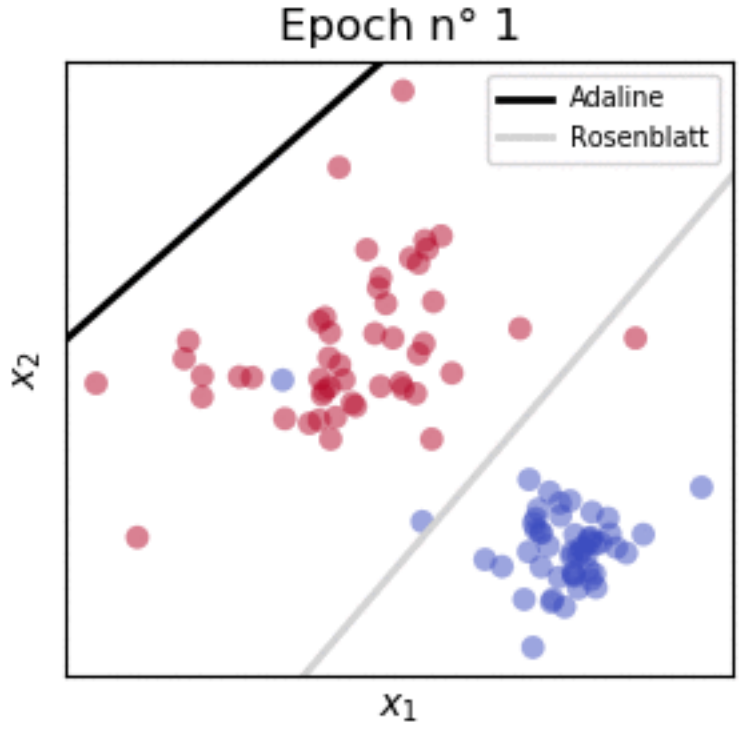
- Changing the learning rule
  - ADALINE adds robustness to training noise
- Adding more layers
  - While single neurons can only compute some logical predicates, *networks* of these neurons can compute any possible boolean function (Rosenblatt, 1962)
  - Multilayer Perceptron can solve XOR
- Changing the activation function
  - Beyond hard thresholds

Improving the Learning Rule

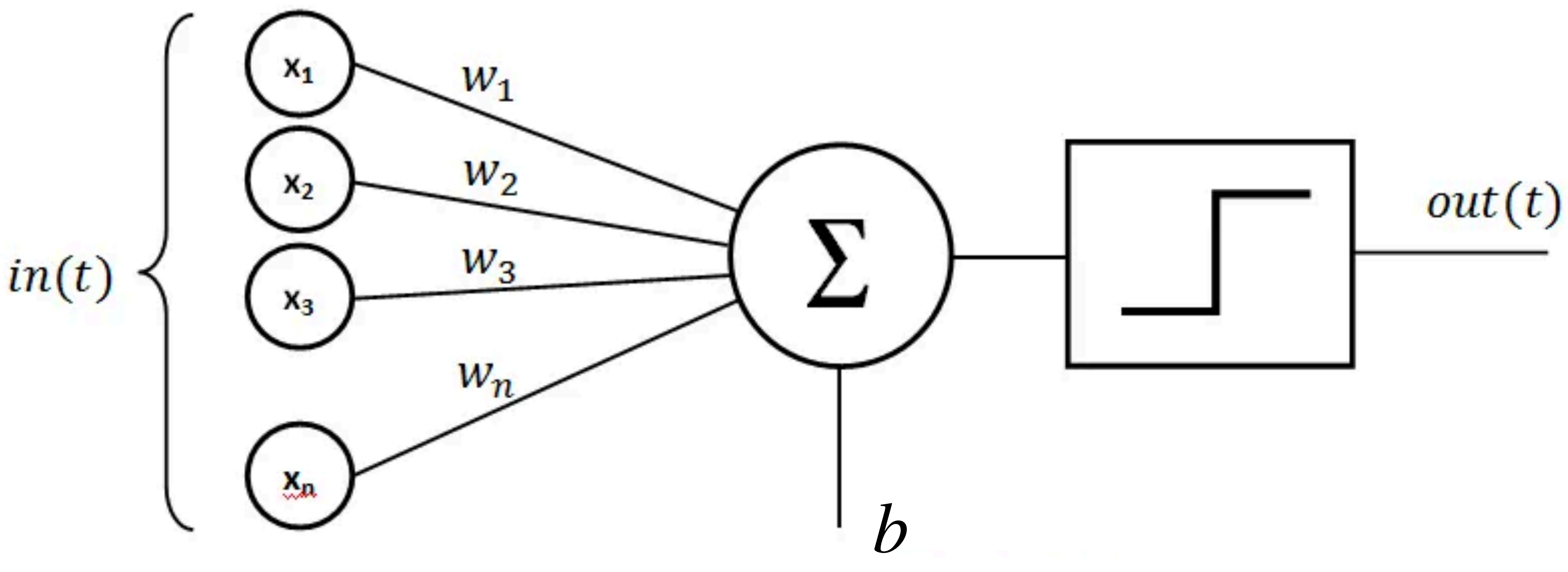
Adaptive Linear Element (ADALINE)

- Weight updates based on a *loss function* rather than the (binary) classification error
  - This uses the activation prior to the sigmoid step, allowing us to compute gradients
- We can use the Delta rule to minimize loss, which is equivalent to stochastic gradient descent for least-squares regression

ADALINE is more robust to training noise:



ADALINE



not on the exam

MSE

$$\mathscr{L}(\mathbf{w}, b) = \frac{1}{2} \sum_{i=1}^{m} \left((\mathbf{w}^{\top} \mathbf{x}_i + b) - y_i \right)^2$$

Weight update

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \Delta \mathbf{w}$$

$$\Delta \mathbf{w} = - \frac{\partial \mathscr{L}}{\partial \mathbf{w}}$$

$$= \sum_{i=1}^{m} \left((\mathbf{w}^{\top} \mathbf{x}_i + b) - y_i \right) \mathbf{x}_i$$

Bias update

$$b \leftarrow b + \alpha \Delta b$$

$$\Delta b = - \frac{\partial \mathscr{L}}{\partial b}$$

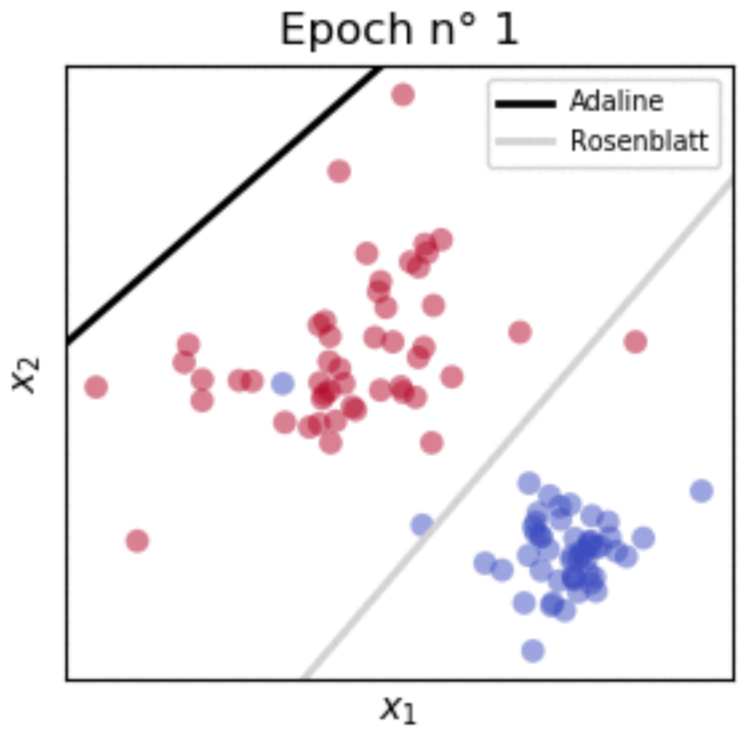
$$= \sum_{i=1}^{m} (\mathbf{w}^{\top} \mathbf{x}_i + b) - y_i$$

Improving the Learning Rule

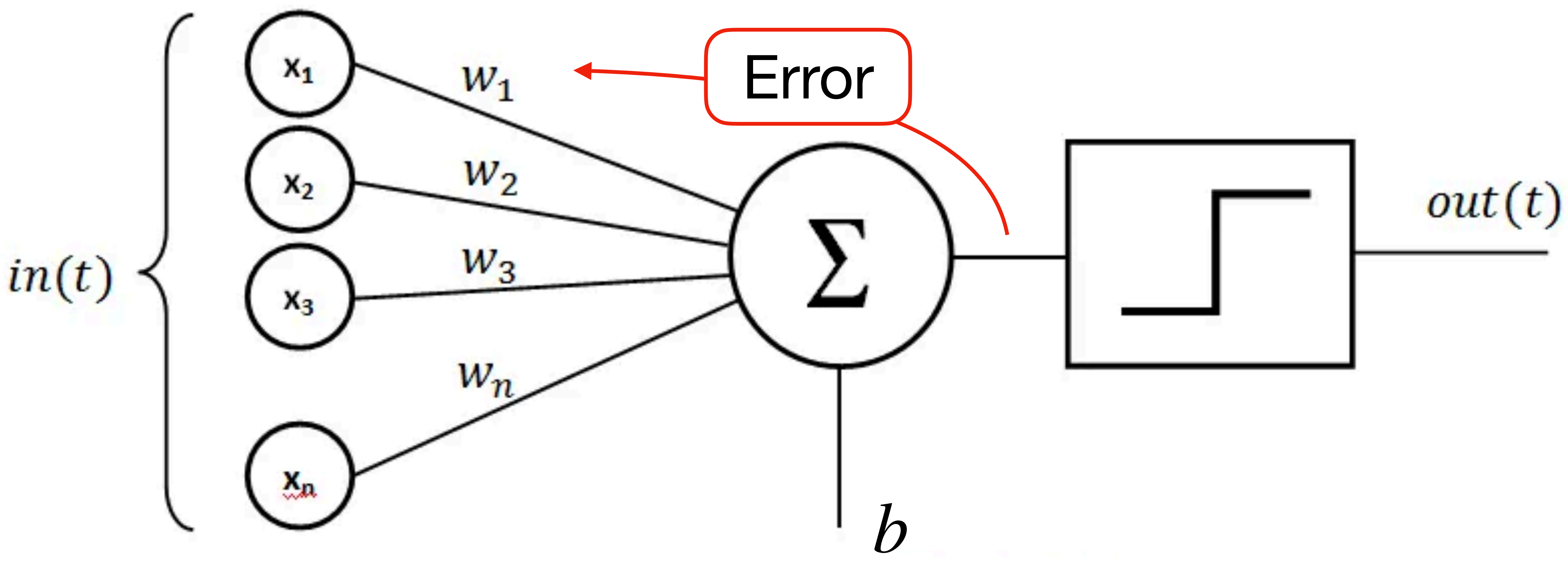
Adaptive Linear Element (ADALINE)

- Weight updates based on a *loss function* rather than the (binary) classification error
  - This uses the activation prior to the sigmoid step, allowing us to compute gradients
- We can use the Delta rule to minimize loss, which is equivalent to stochastic gradient descent for least-squares regression

ADALINE is more robust to training noise:



ADALINE



not on the exam

MSE

$$\mathscr{L}(\mathbf{w}, b) = \frac{1}{2}\sum_{i=1}^{m}\left((\mathbf{w}^\top \mathbf{x}_i + b) - y_i\right)^2$$

Weight update

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \Delta \mathbf{w}$$

$$\Delta \mathbf{w} = -\frac{\partial \mathscr{L}}{\partial \mathbf{w}}$$

$$= \sum_{i=1}^{m}\left((\mathbf{w}^\top \mathbf{x}_i + b) - y_i\right)\mathbf{x}_i$$

Bias update

$$b \leftarrow b + \alpha \Delta b$$

$$\Delta b = -\frac{\partial \mathscr{L}}{\partial b}$$

$$= \sum_{i=1}^{m}(\mathbf{w}^\top \mathbf{x}_i + b) - y_i$$

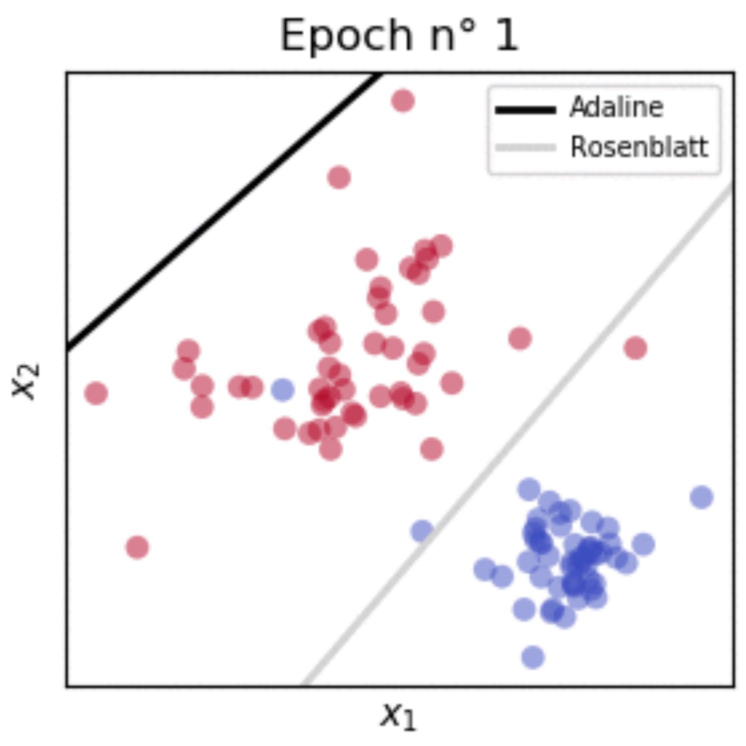
Widrow & Hoff, 1960

Improving the Learning Rule

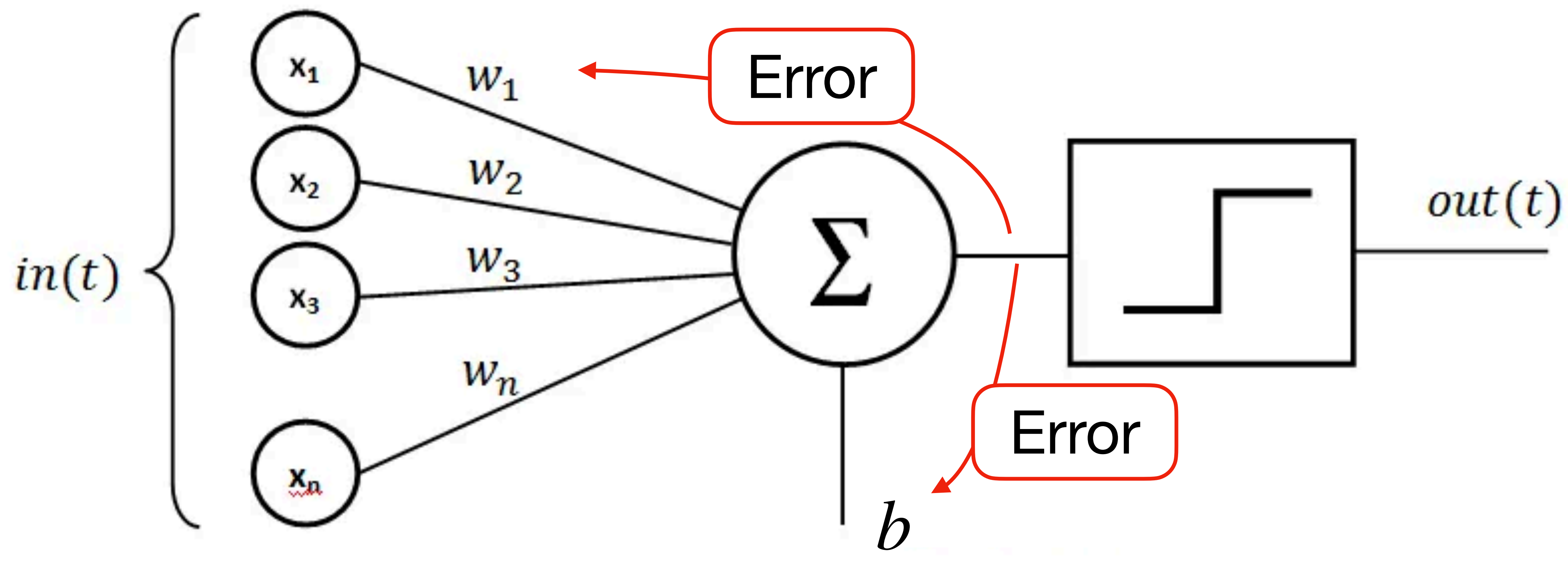
Adaptive Linear Element (ADALINE)

- Weight updates based on a *loss function* rather than the (binary) classification error
  - This uses the activation prior to the sigmoid step, allowing us to compute gradients
- We can use the Delta rule to minimize loss, which is equivalent to stochastic gradient descent for least-squares regression

ADALINE is more robust to training noise:



ADALINE



not on the exam

MSE

$$\mathscr{L}(\mathbf{w}, b) = \frac{1}{2}\sum_{i=1}^{m}\left((\mathbf{w}^\top\mathbf{x}_i + b) - y_i\right)^2$$

Weight update

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\Delta\mathbf{w}$$

$$\Delta\mathbf{w} = -\frac{\partial\mathscr{L}}{\partial\mathbf{w}}$$

$$= \sum_{i=1}^{m}\left((\mathbf{w}^\top\mathbf{x}_i + b) - y_i\right)\mathbf{x}_i$$

Bias update

$$b \leftarrow b + \alpha\Delta b$$

$$\Delta b = -\frac{\partial\mathscr{L}}{\partial b}$$

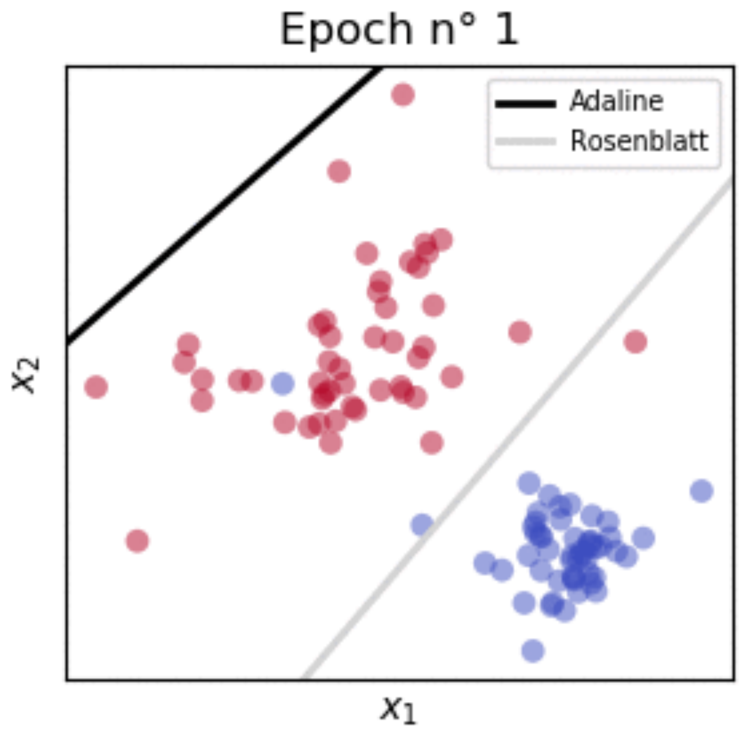
$$= \sum_{i=1}^{m}(\mathbf{w}^\top\mathbf{x}_i + b) - y_i$$

Improving the Learning Rule

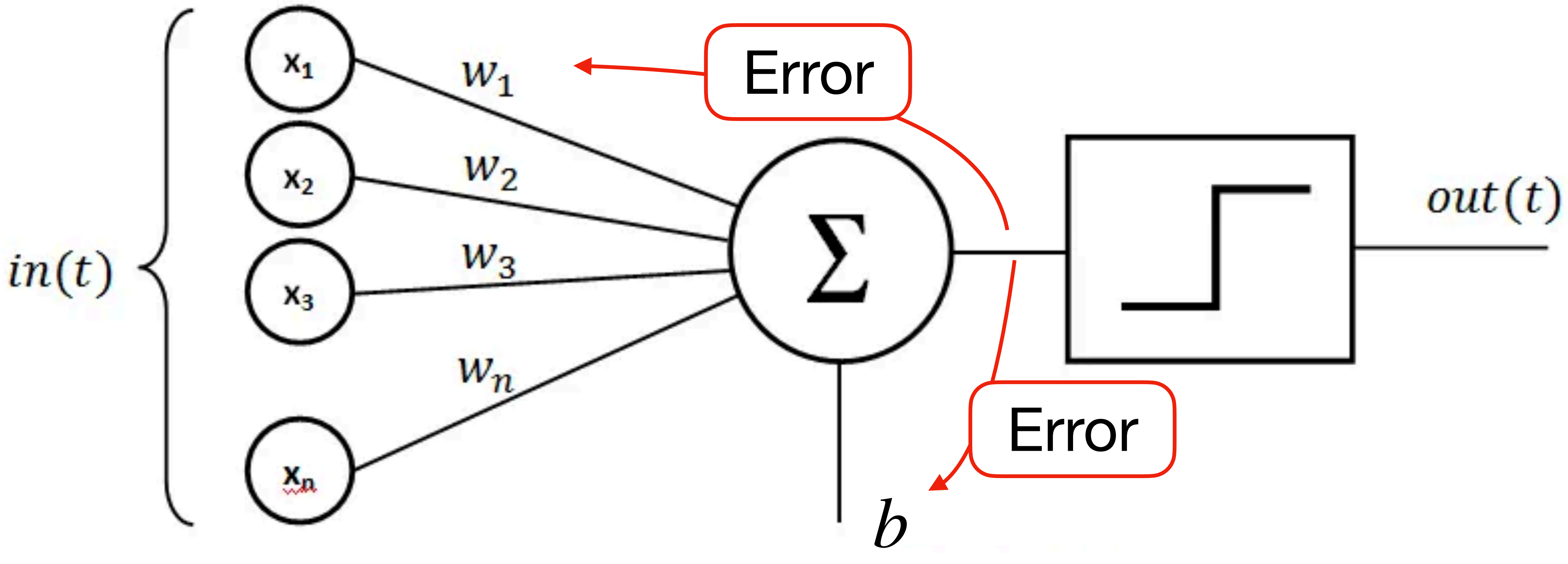
Adaptive Linear Element (ADALINE)

- Weight updates based on a *loss function* rather than the (binary) classification error
  - This uses the activation prior to the sigmoid step, allowing us to compute gradients
- We can use the Delta rule to minimize loss, which is equivalent to stochastic gradient descent for least-squares regression

ADALINE is more robust to training noise:



ADALINE



not on the exam

MSE

$$\mathscr{L}(\mathbf{w}, b) = \frac{1}{2}\sum_{i=1}^{m}\left((\mathbf{w}^\top \mathbf{x}_i + b) - y_i\right)^2$$

Weight update

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \Delta \mathbf{w}$$

$$\Delta \mathbf{w} = -\frac{\partial \mathscr{L}}{\partial \mathbf{w}}$$

$$= \sum_{i=1}^{m}\left((\mathbf{w}^\top \mathbf{x}_i + b) - y_i\right)\mathbf{x}_i$$

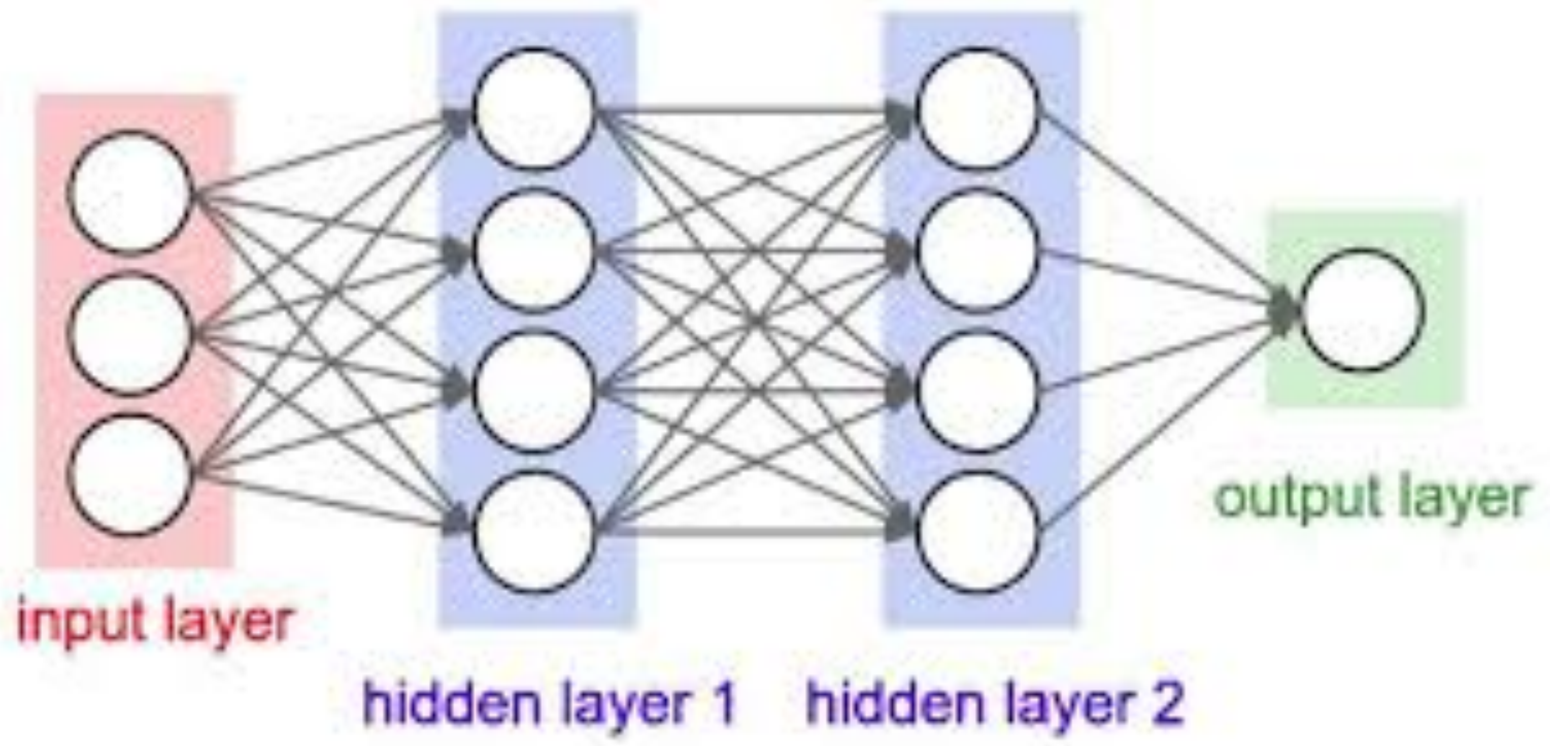
Bias update

$$b \leftarrow b + \alpha \Delta b$$

$$\Delta b = -\frac{\partial \mathscr{L}}{\partial b}$$

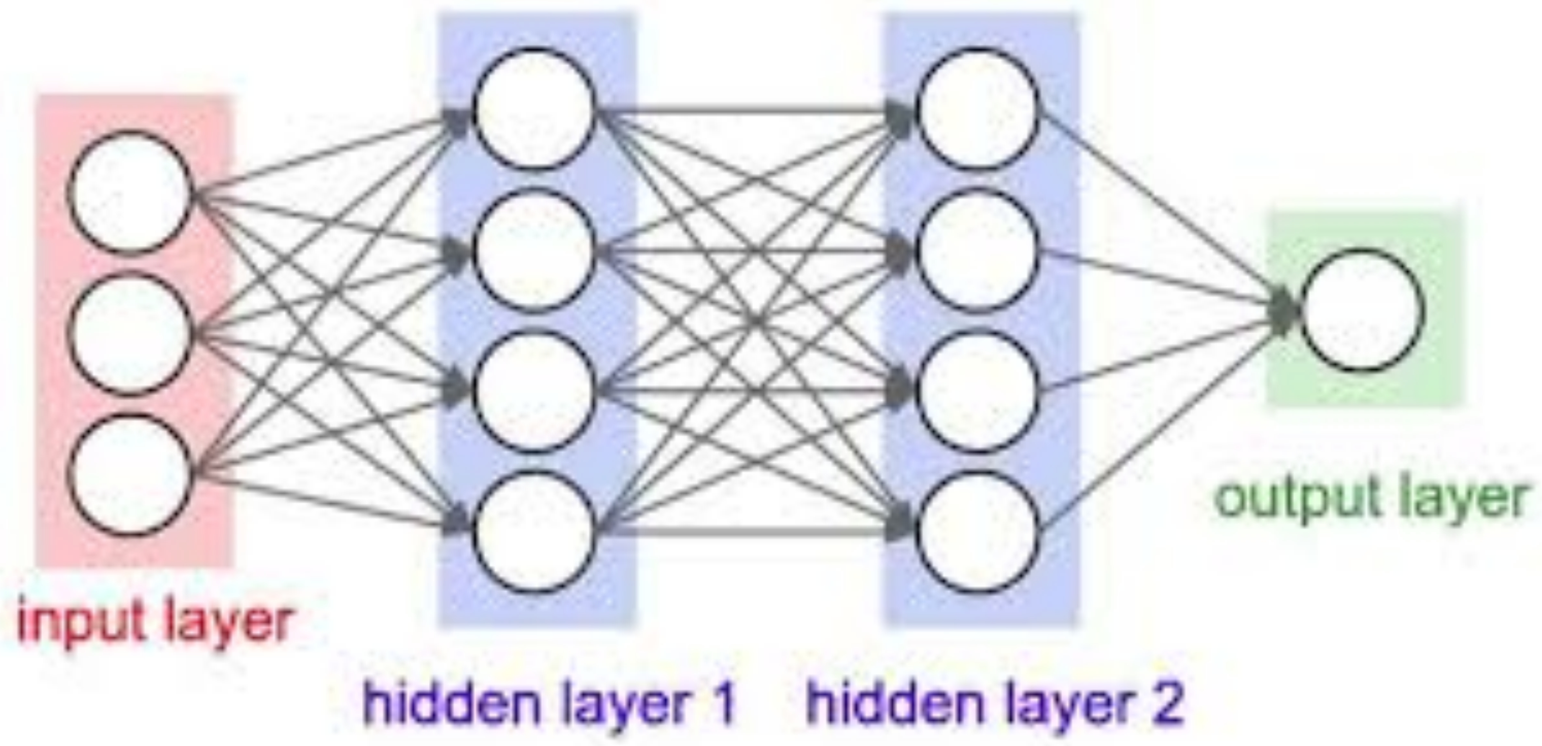
$$= \sum_{i=1}^{m}(\mathbf{w}^\top \mathbf{x}_i + b) - y_i$$

Multilayer Perceptron

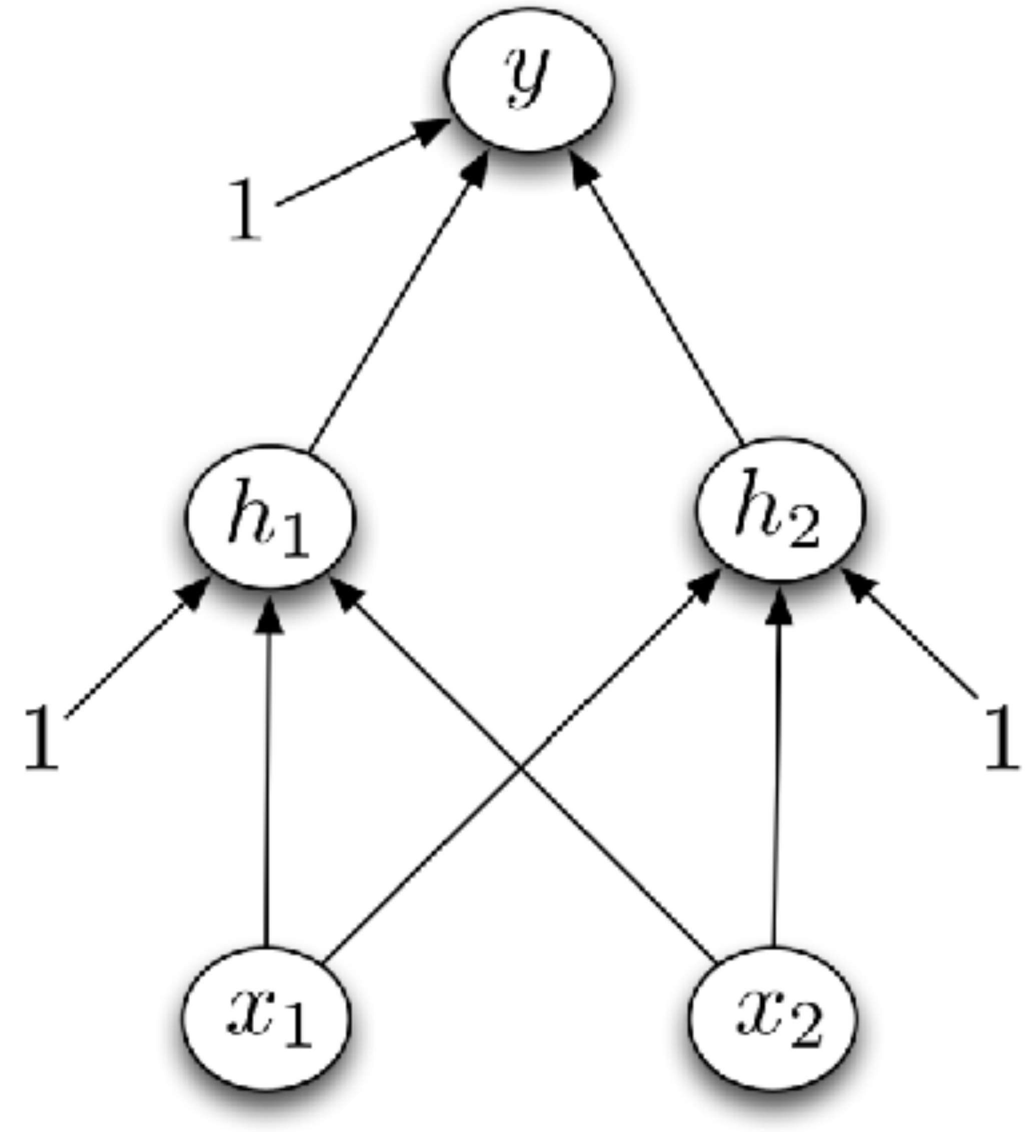


- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR

Multilayer Perceptron

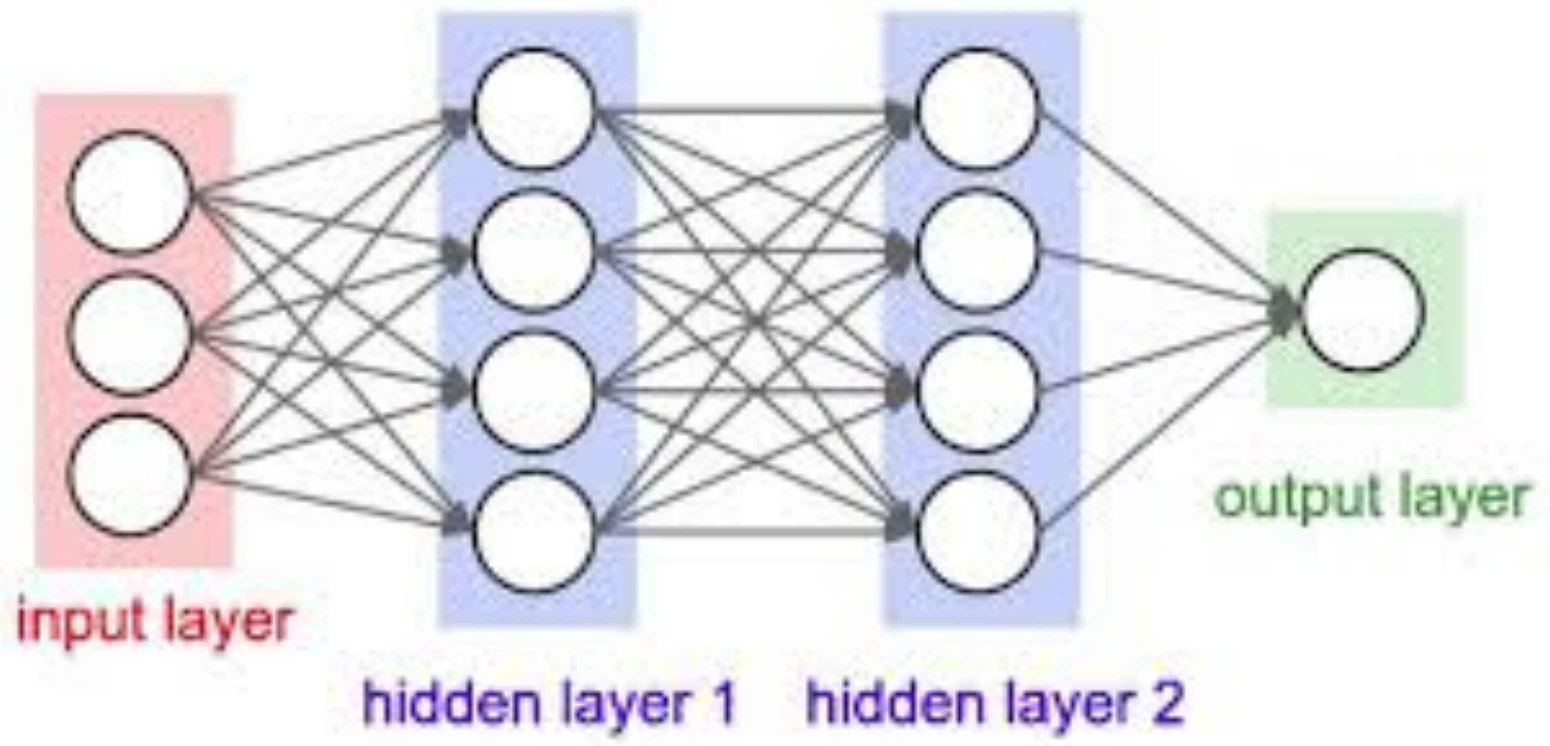


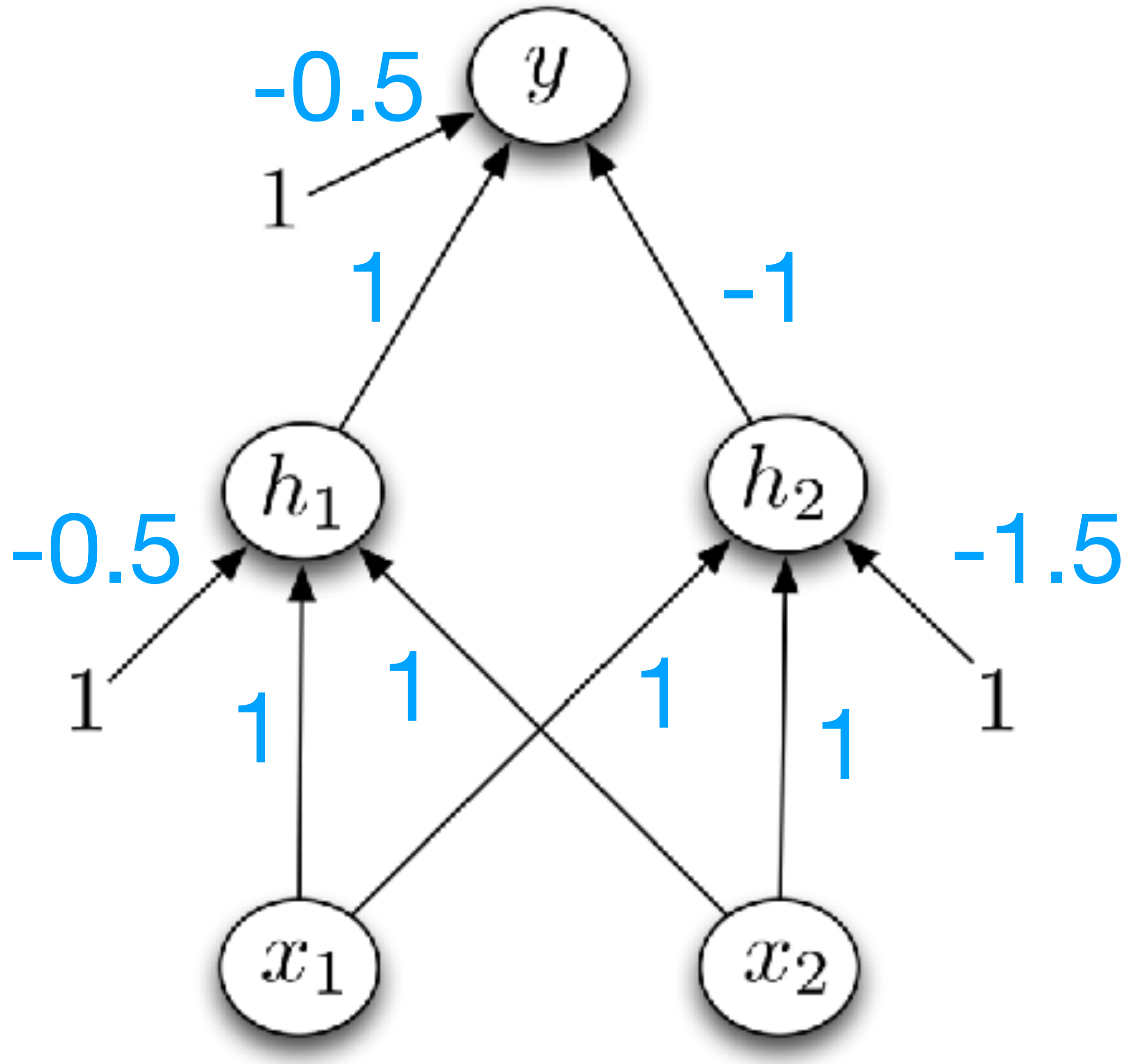
- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR



Multilayer Perceptron

- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR

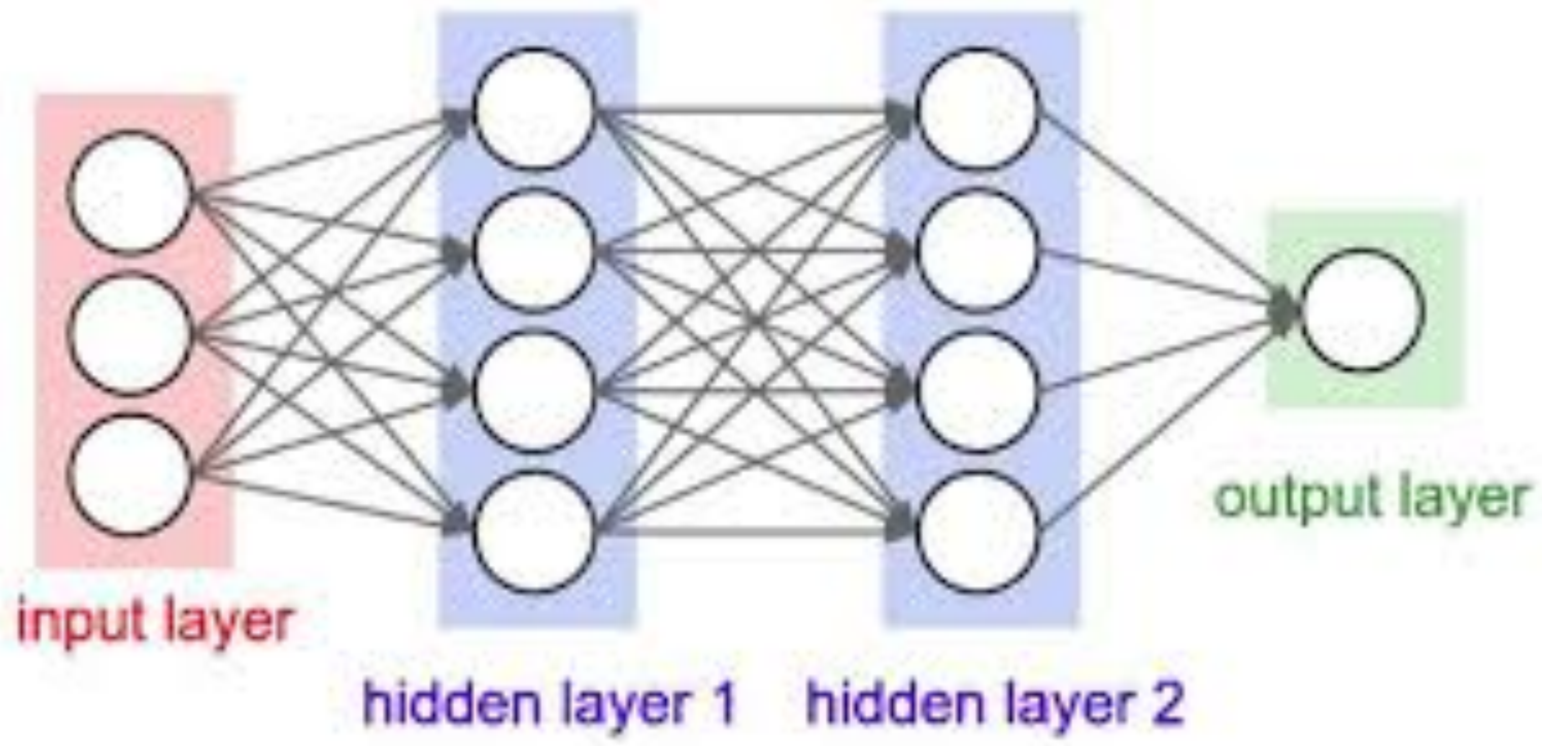




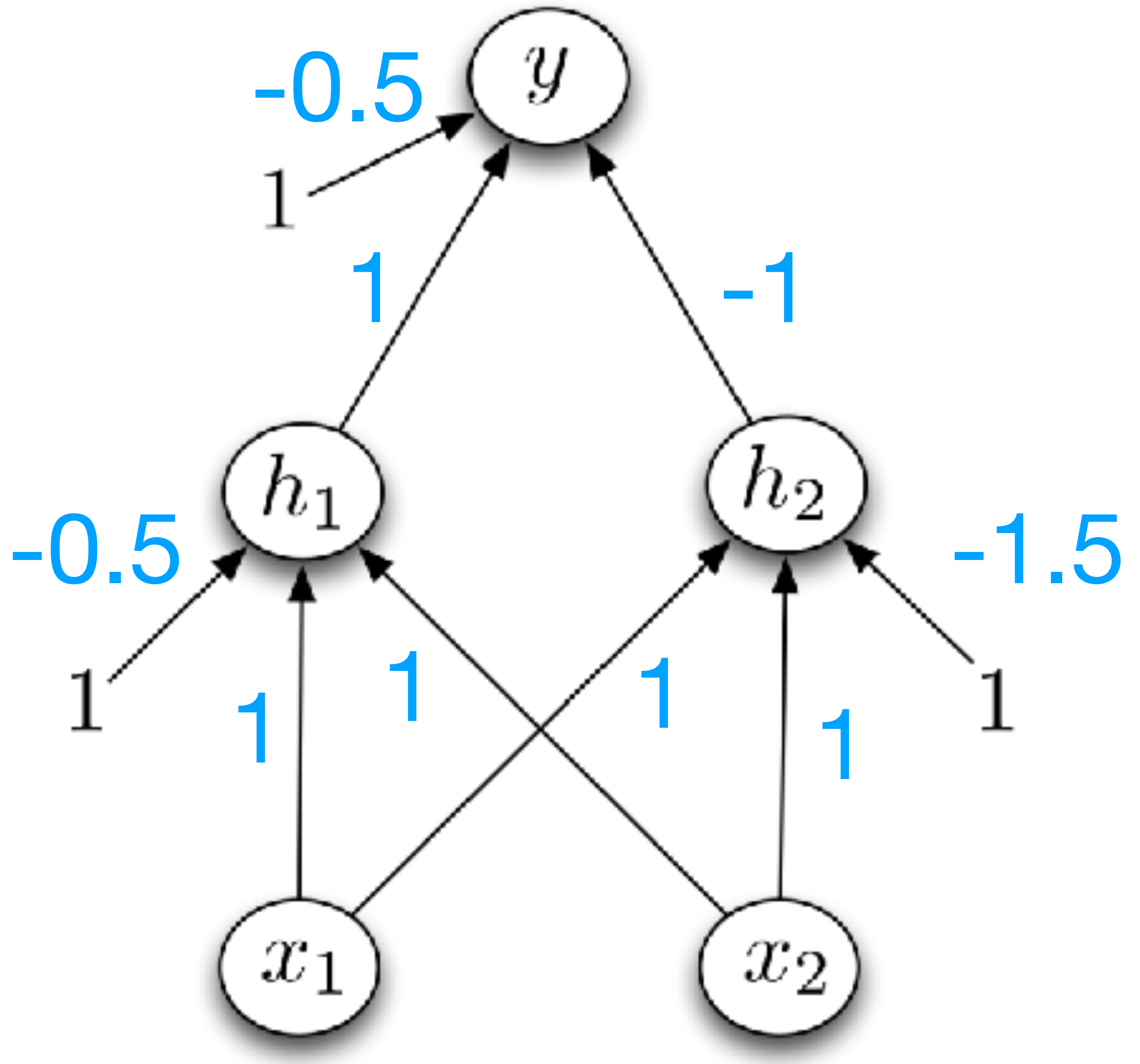
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0			
1	1			
1	0			
0	1			

Multilayer Perceptron



- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR

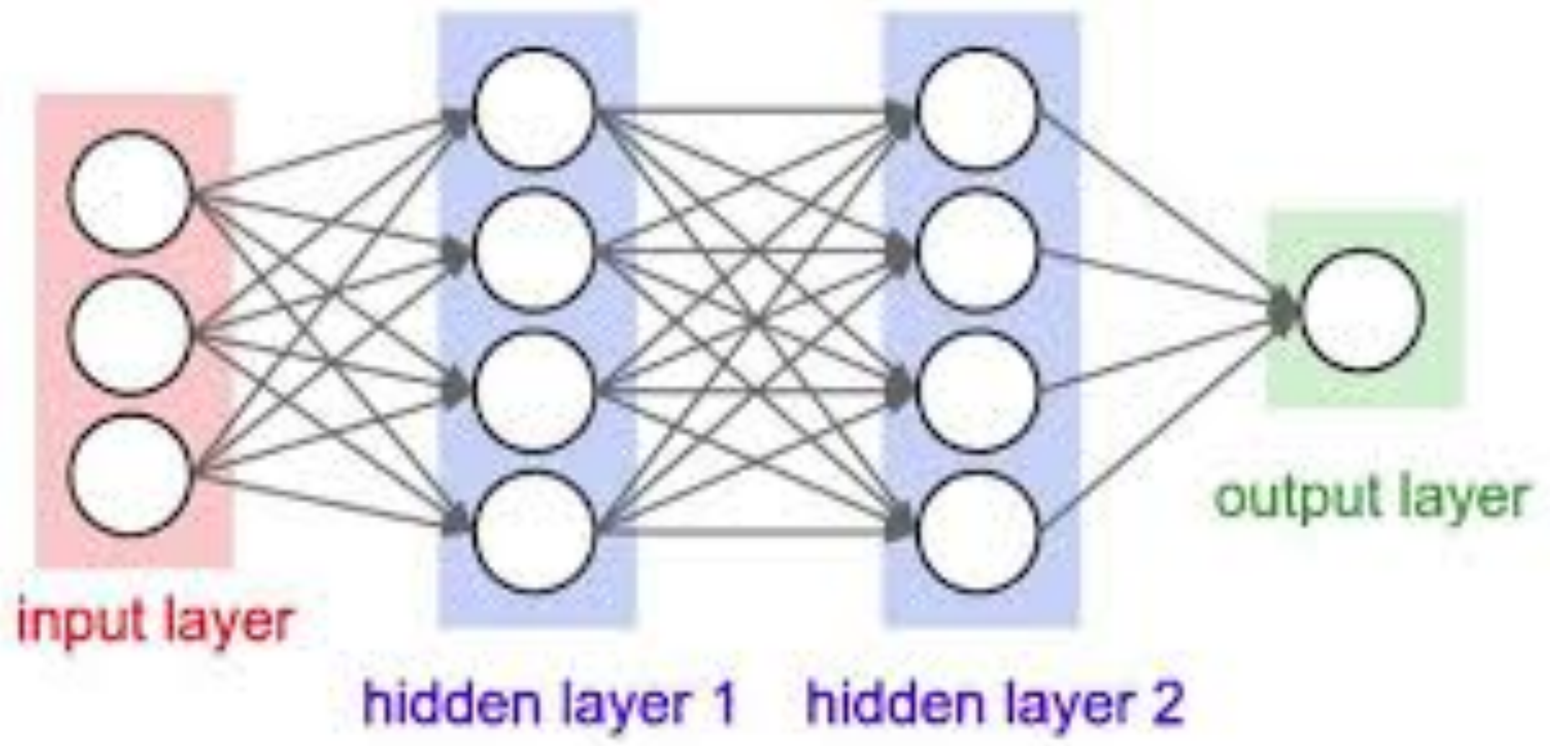


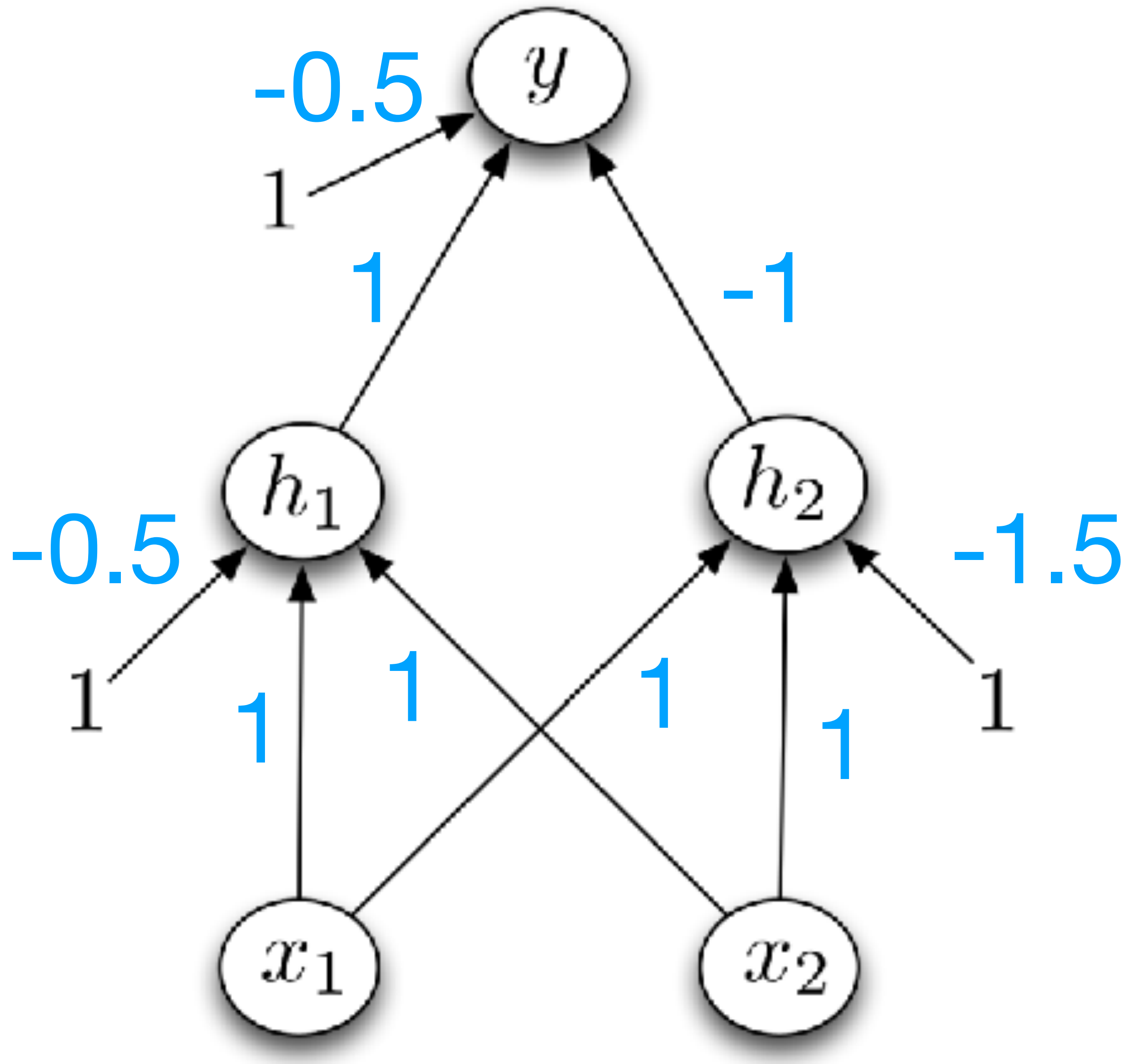
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$		
1	1			
1	0			
0	1			

Multilayer Perceptron

- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR

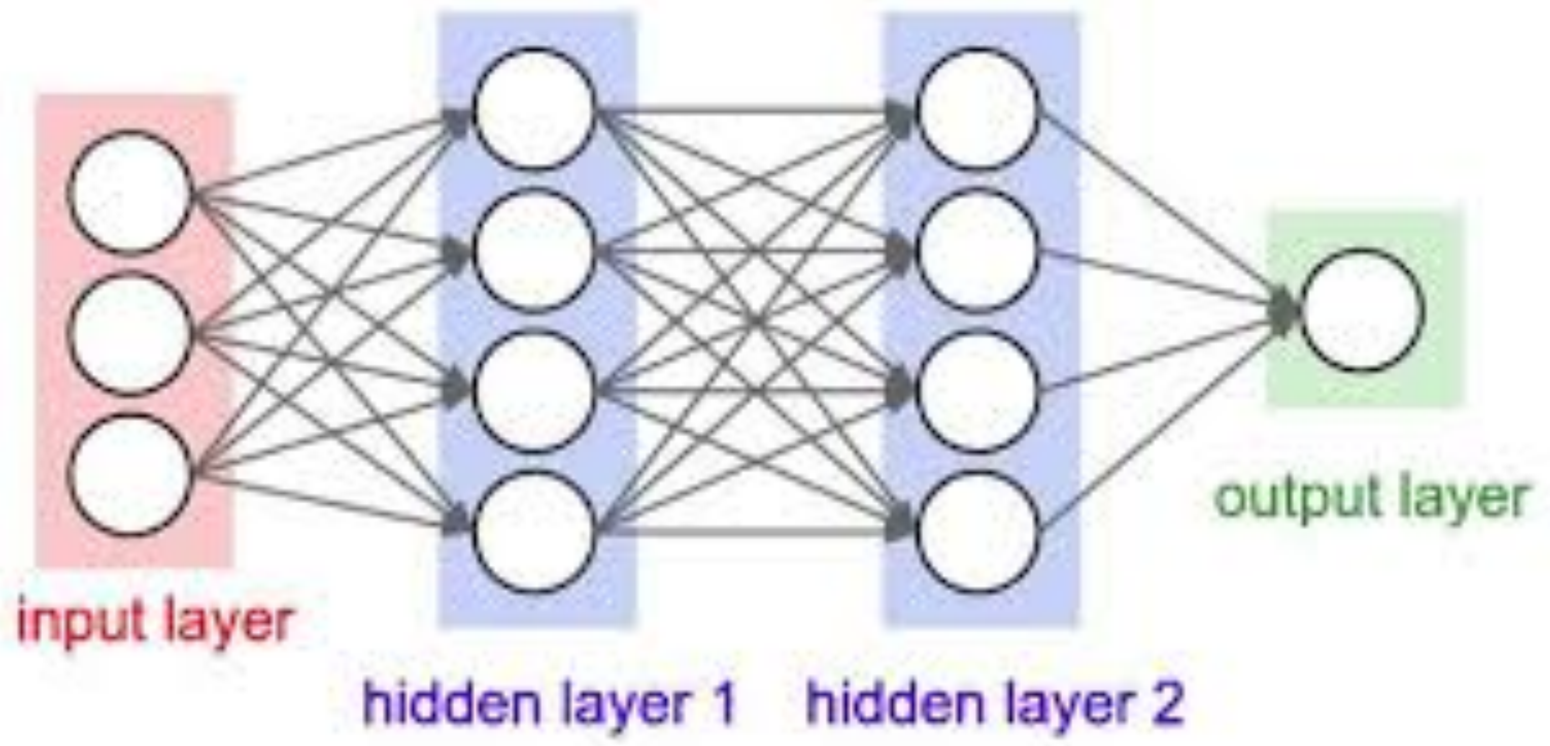




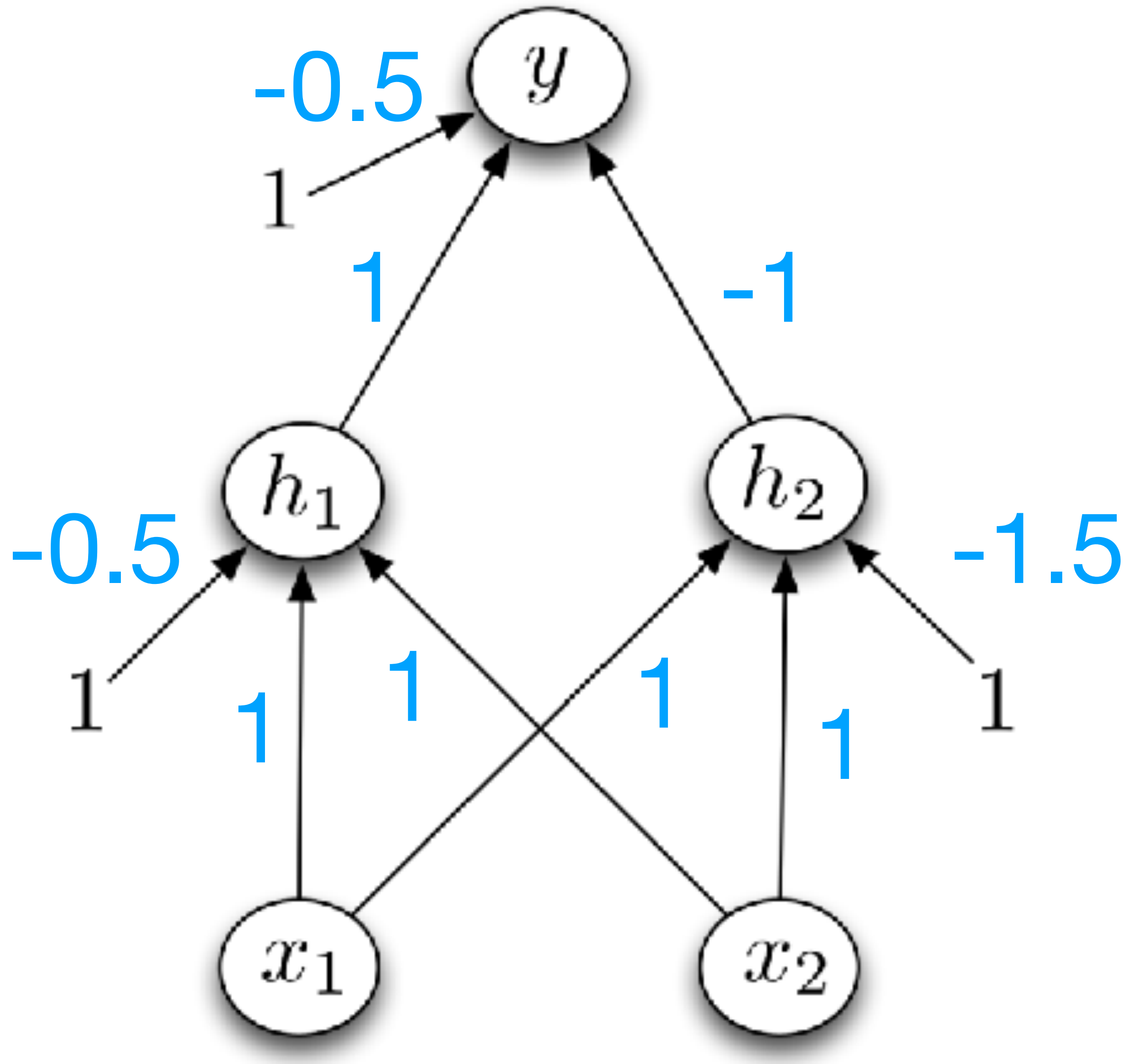
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	
1	1			
1	0			
0	1			

Multilayer Perceptron



- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR

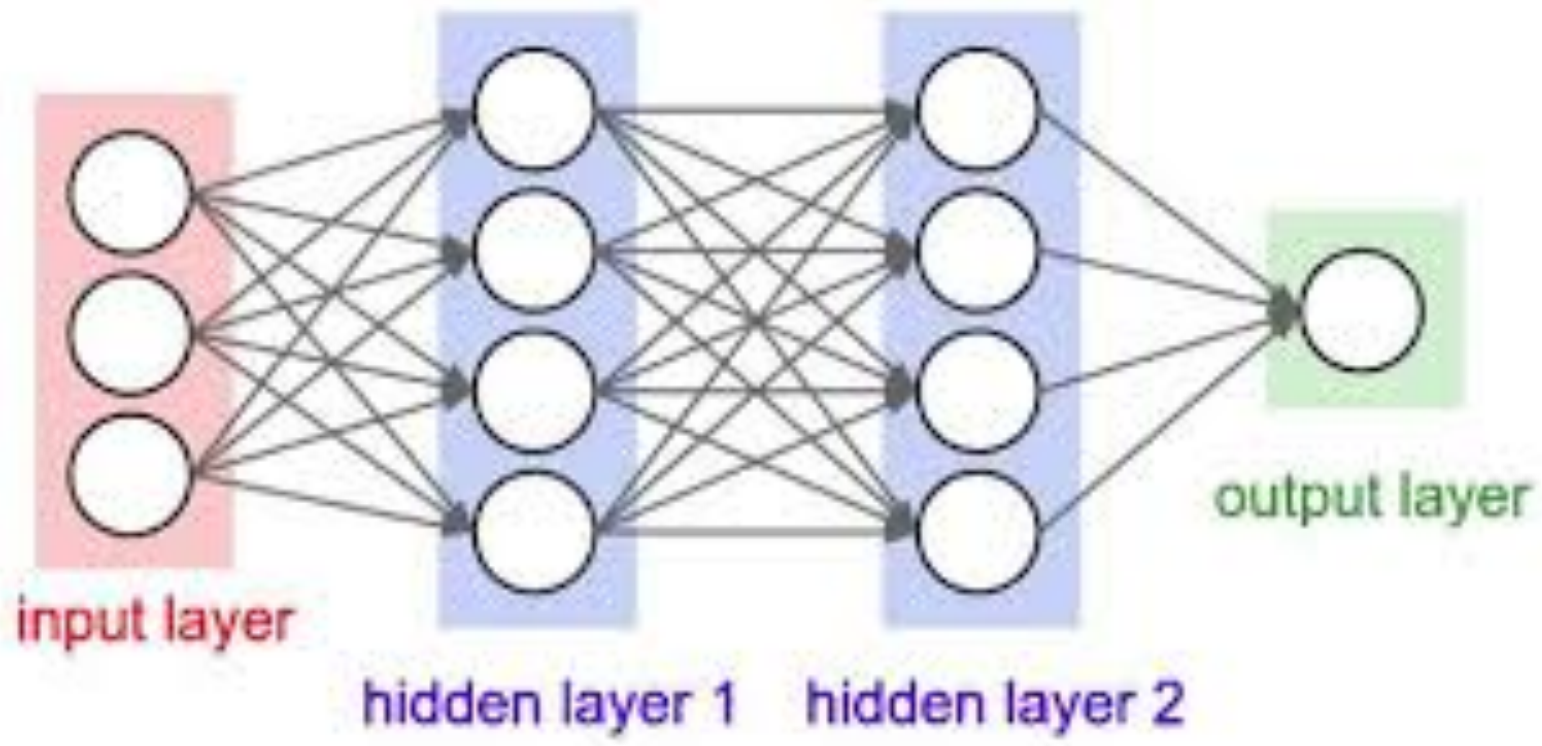


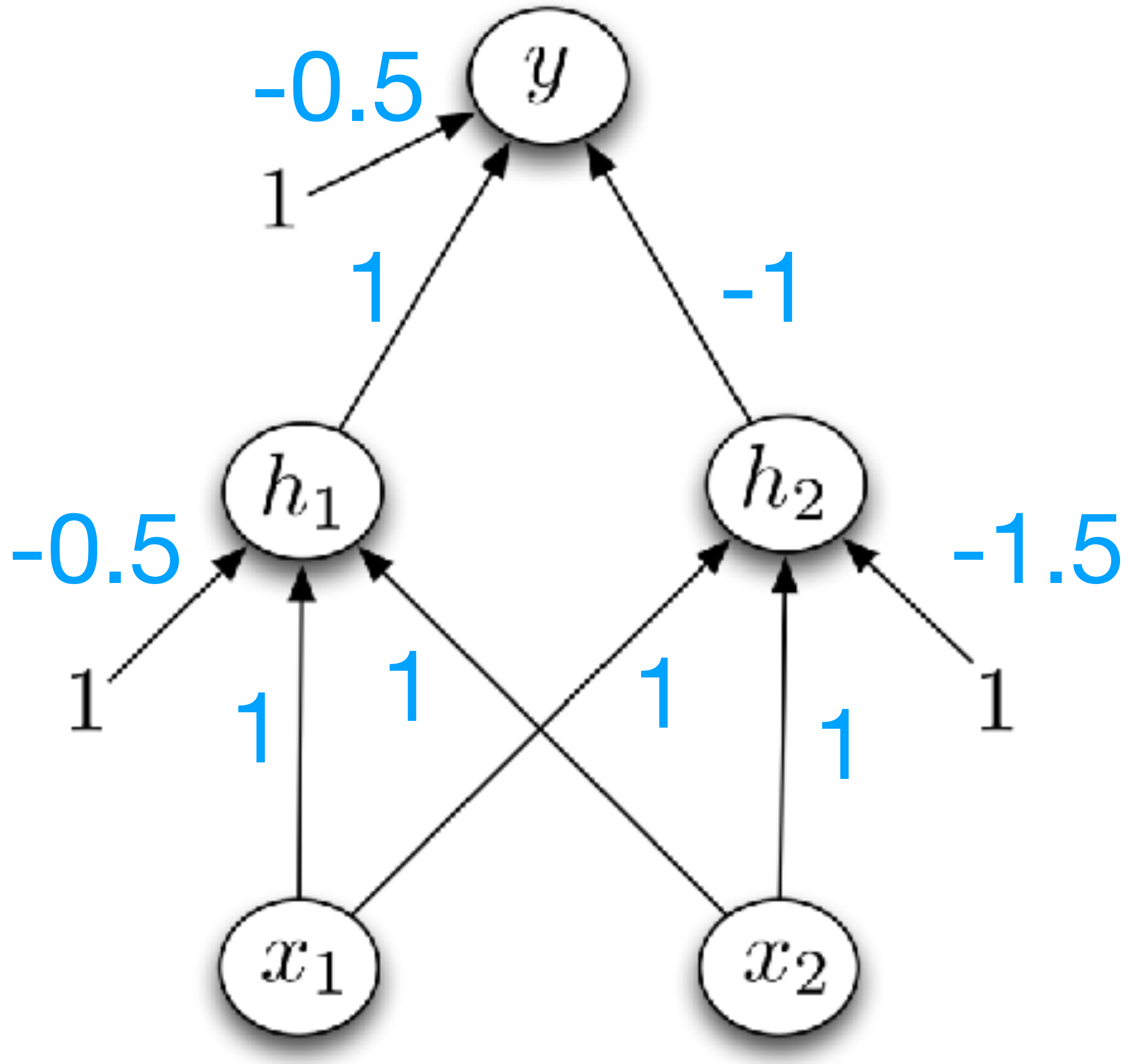
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	$\sigma(-.5) = 0$
1	1			
1	0			
0	1			

Multilayer Perceptron

- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR



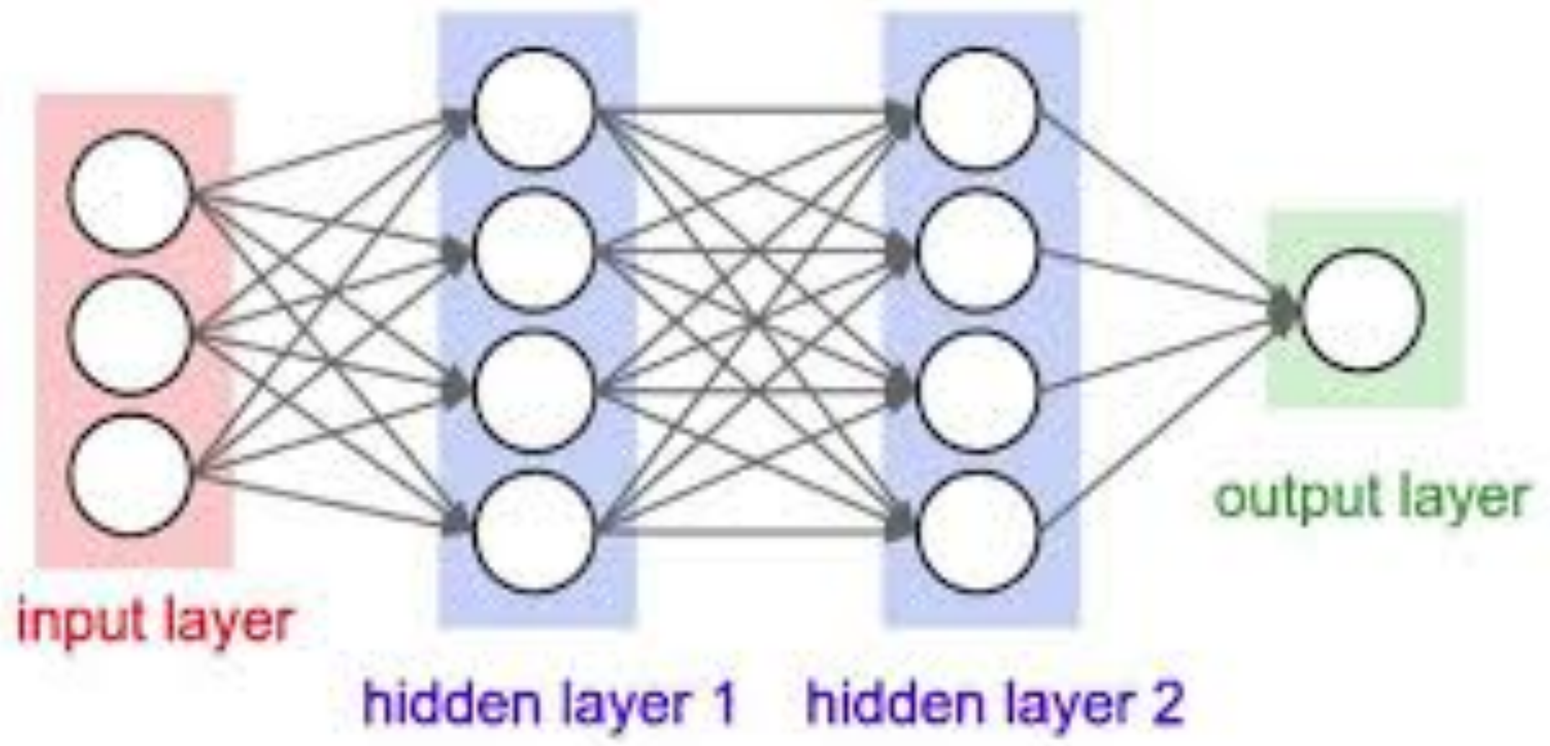


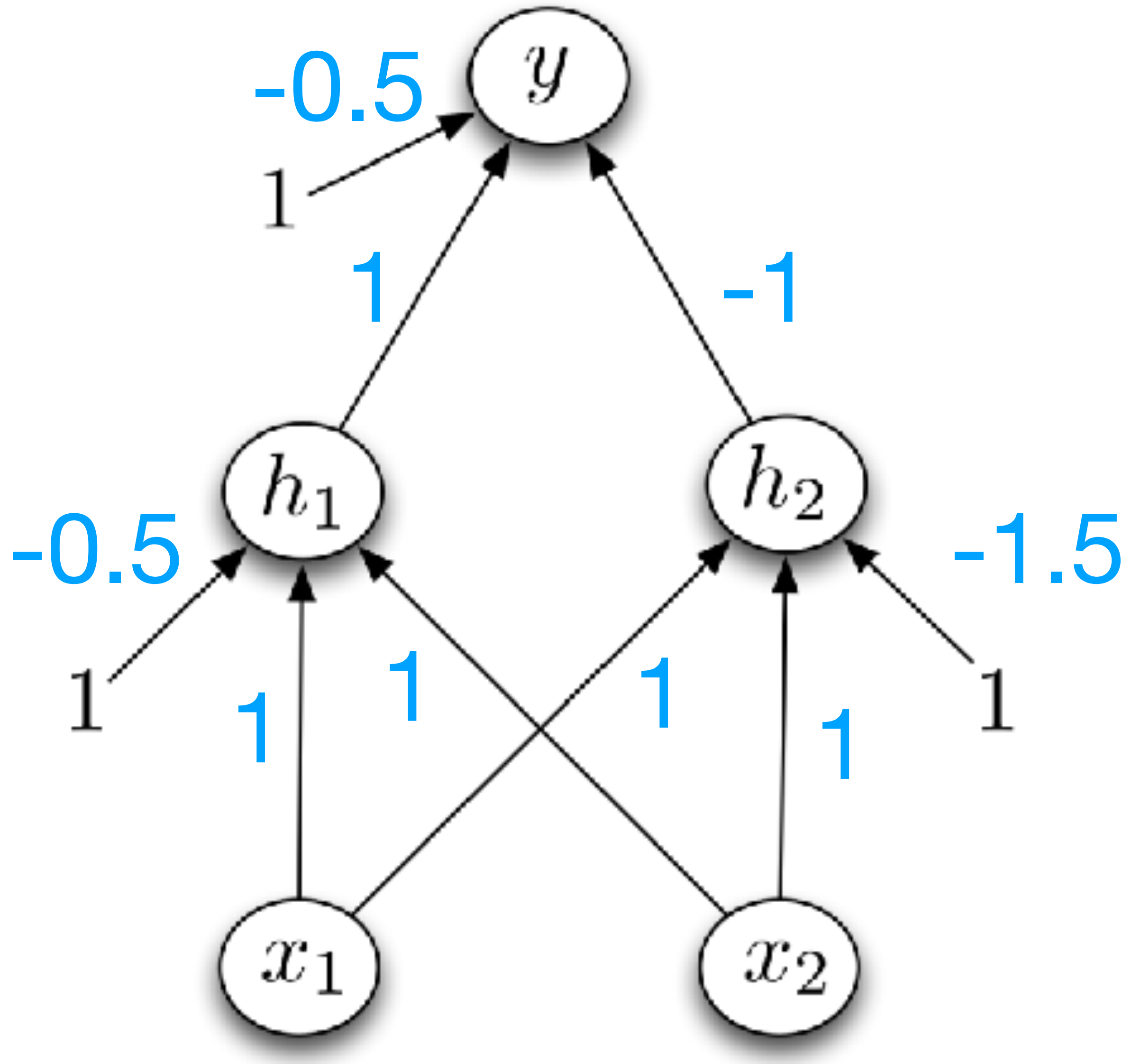
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	$\sigma(-.5) = 0$
1	1	$\sigma(1.5) = 1$		
1	0			
0	1			

Multilayer Perceptron

- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR



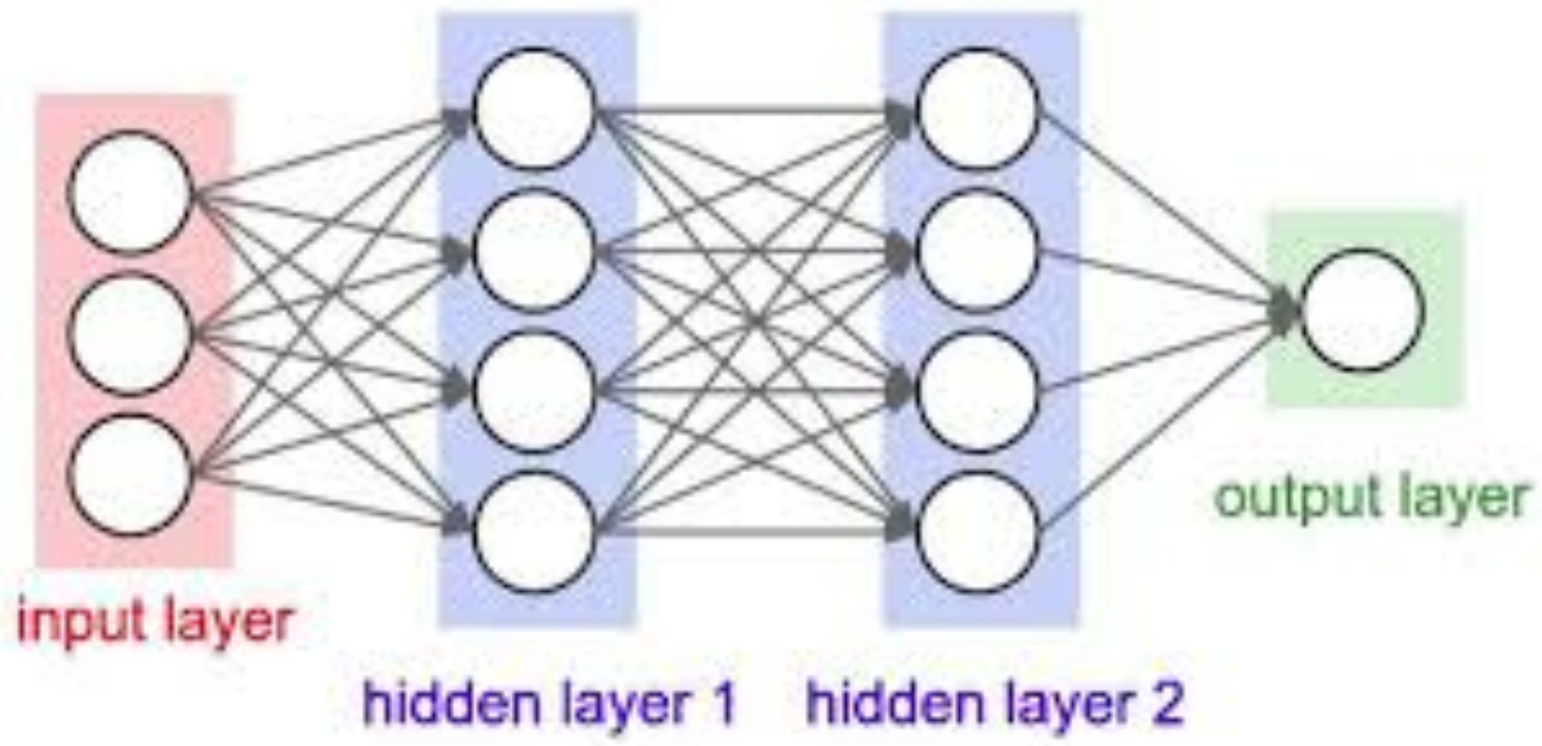


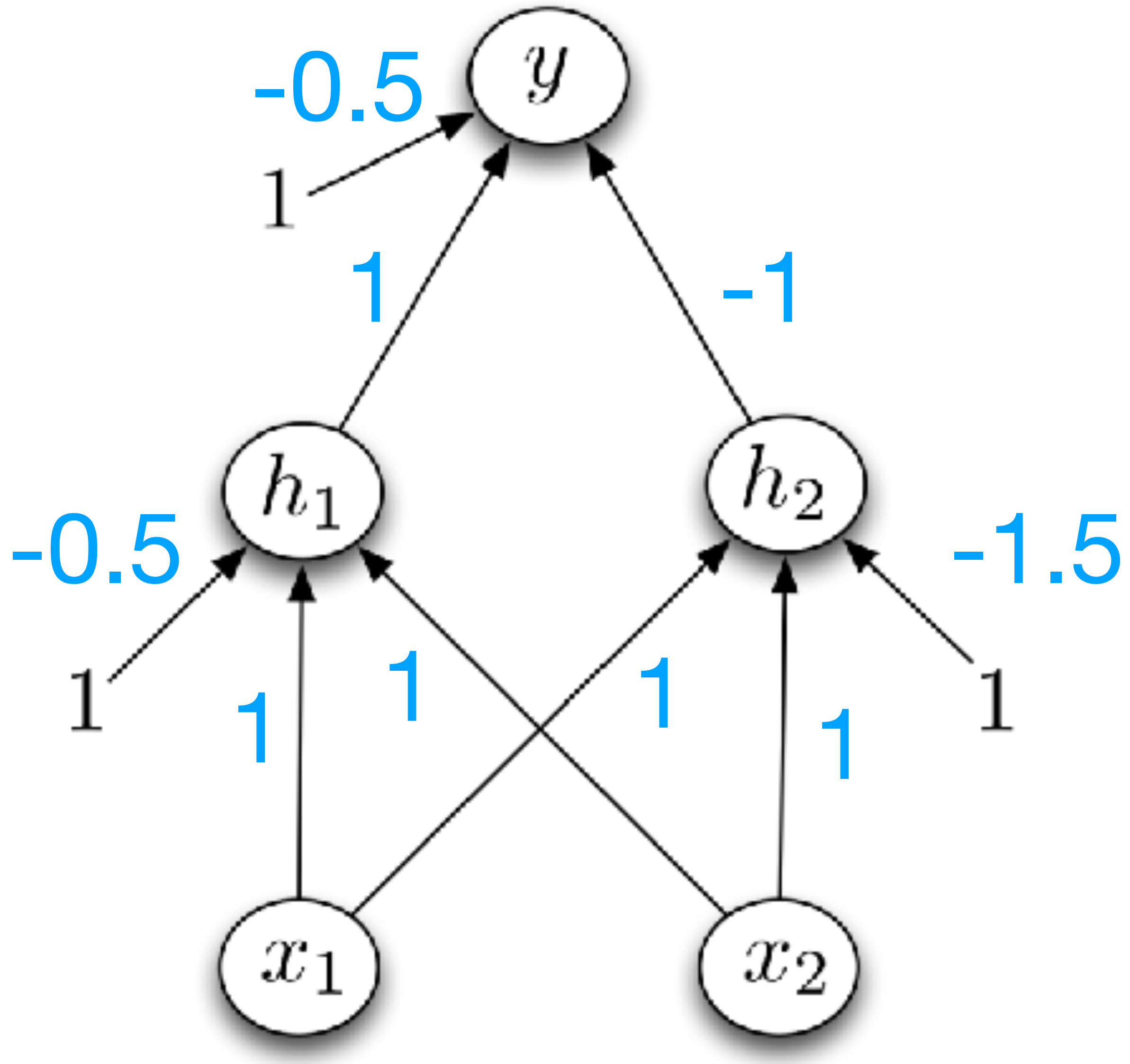
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	$\sigma(-.5) = 0$
1	1	$\sigma(1.5) = 1$	$\sigma(.5) = 1$	
1	0			
0	1			

Multilayer Perceptron

- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^{\top}\mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^{\top}\mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR



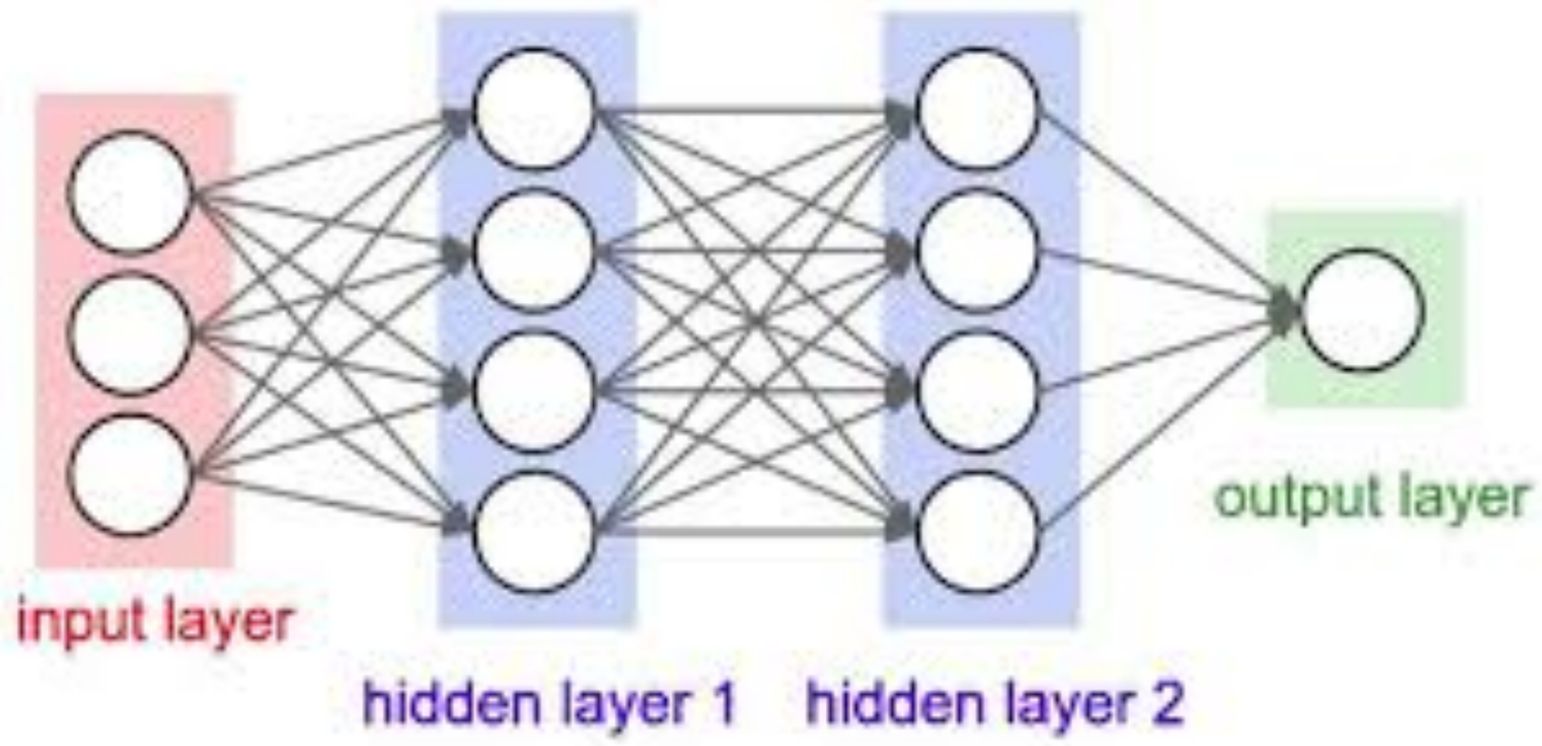


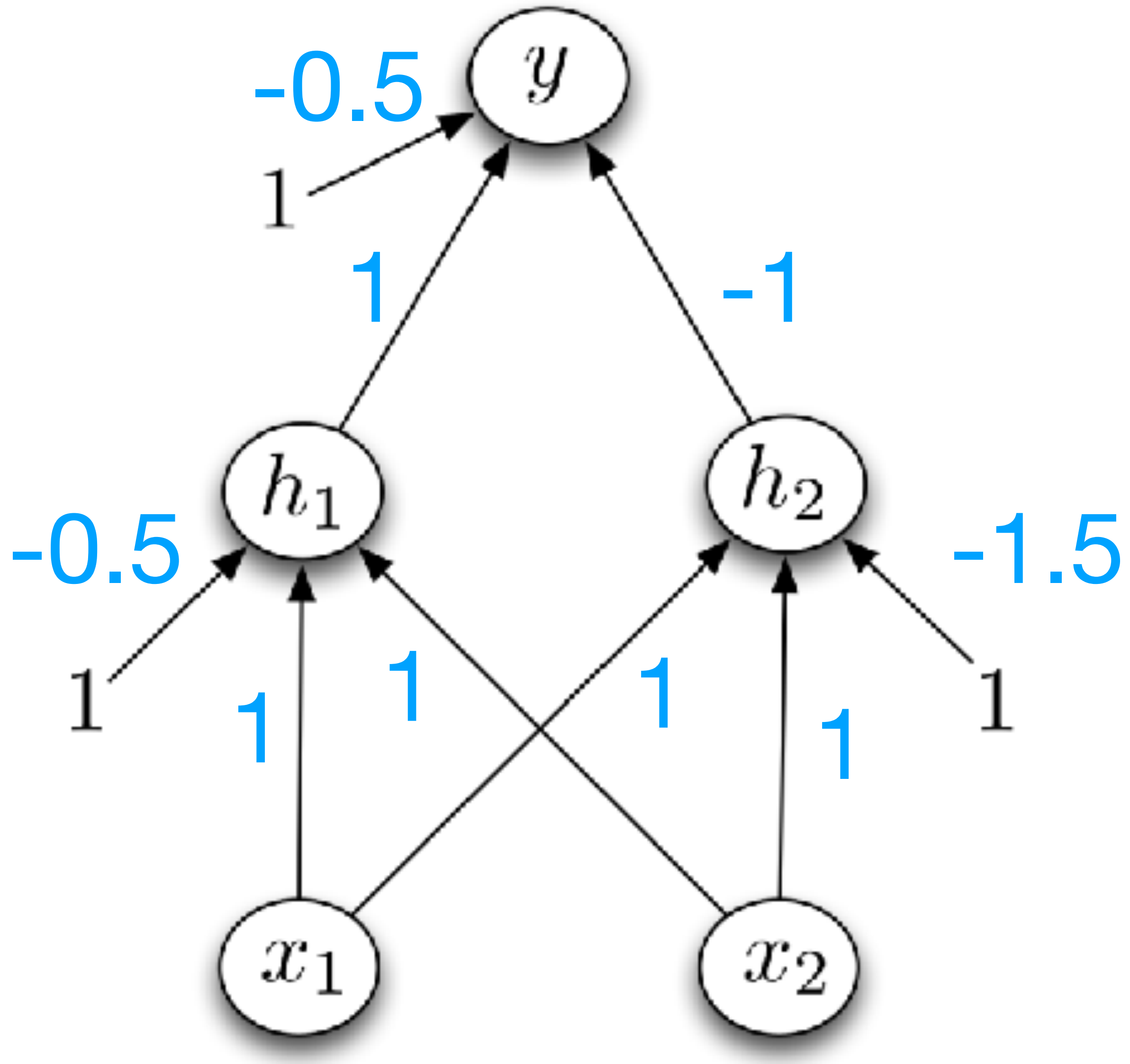
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	$\sigma(-.5) = 0$
1	1	$\sigma(1.5) = 1$	$\sigma(.5) = 1$	$\sigma(-.5) = 0$
1	0			
0	1			

Multilayer Perceptron

- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR



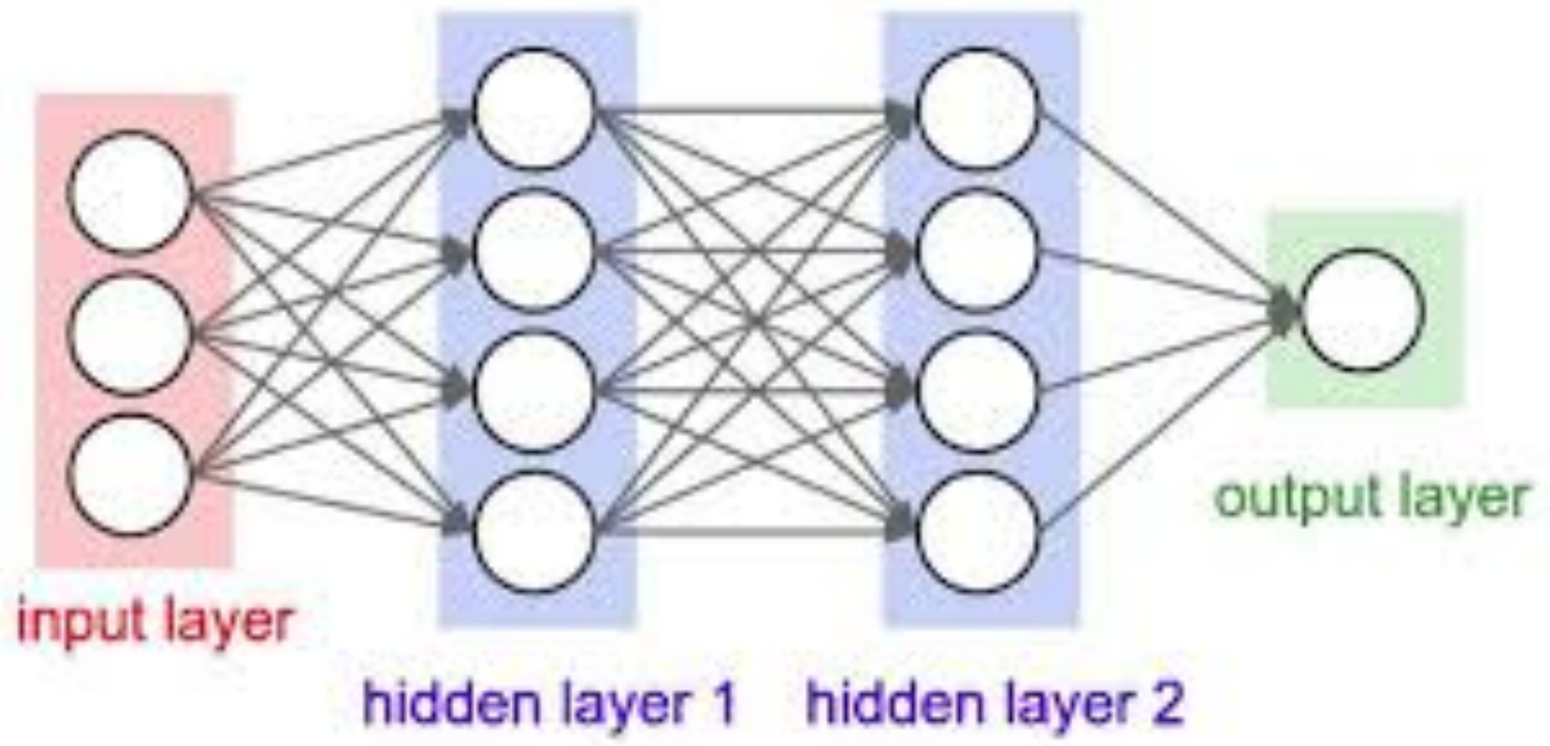


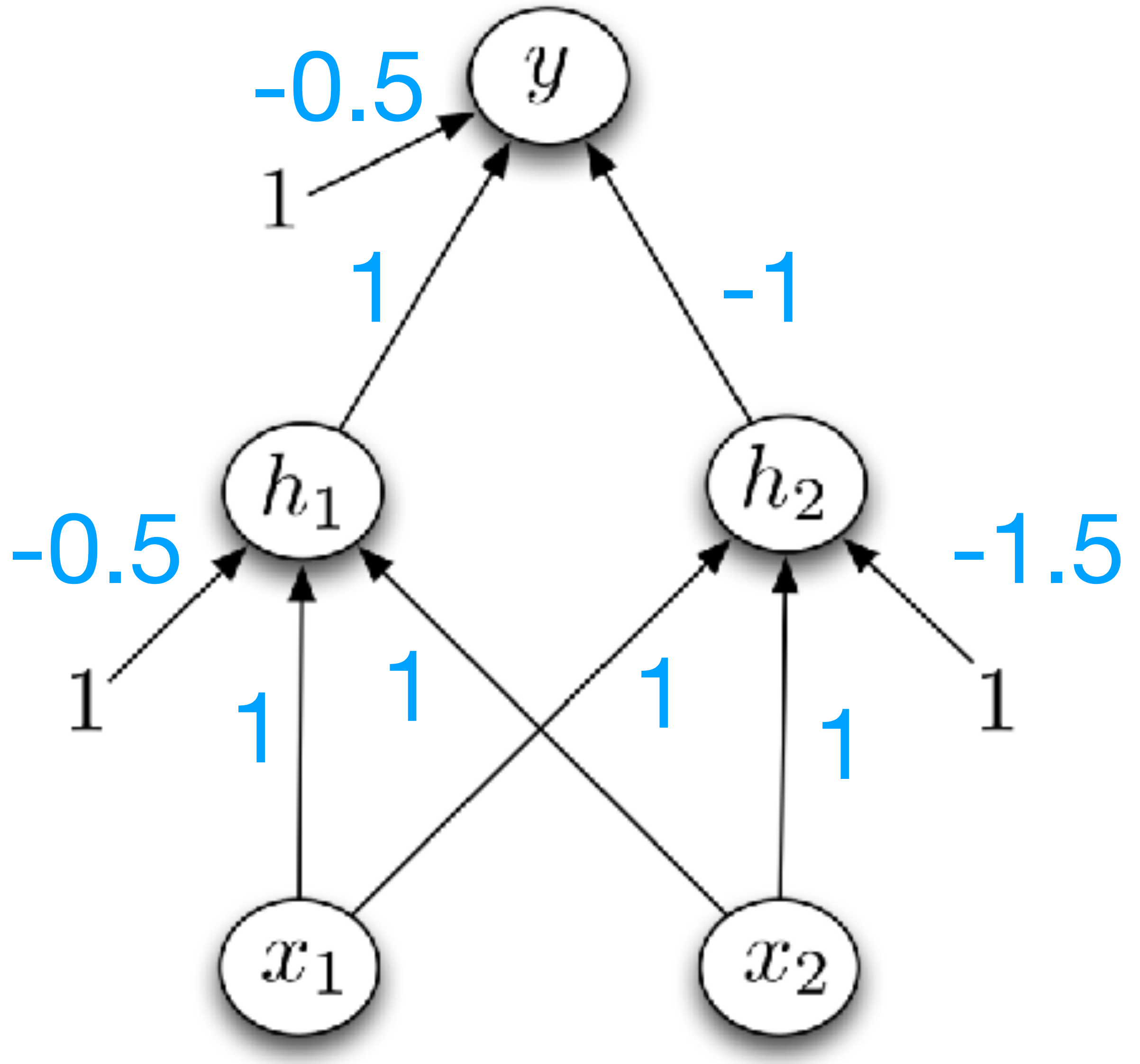
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	$\sigma(-.5) = 0$
1	1	$\sigma(1.5) = 1$	$\sigma(.5) = 1$	$\sigma(-.5) = 0$
1	0	$\sigma(.5) = 1$		
0	1			

Multilayer Perceptron

- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR



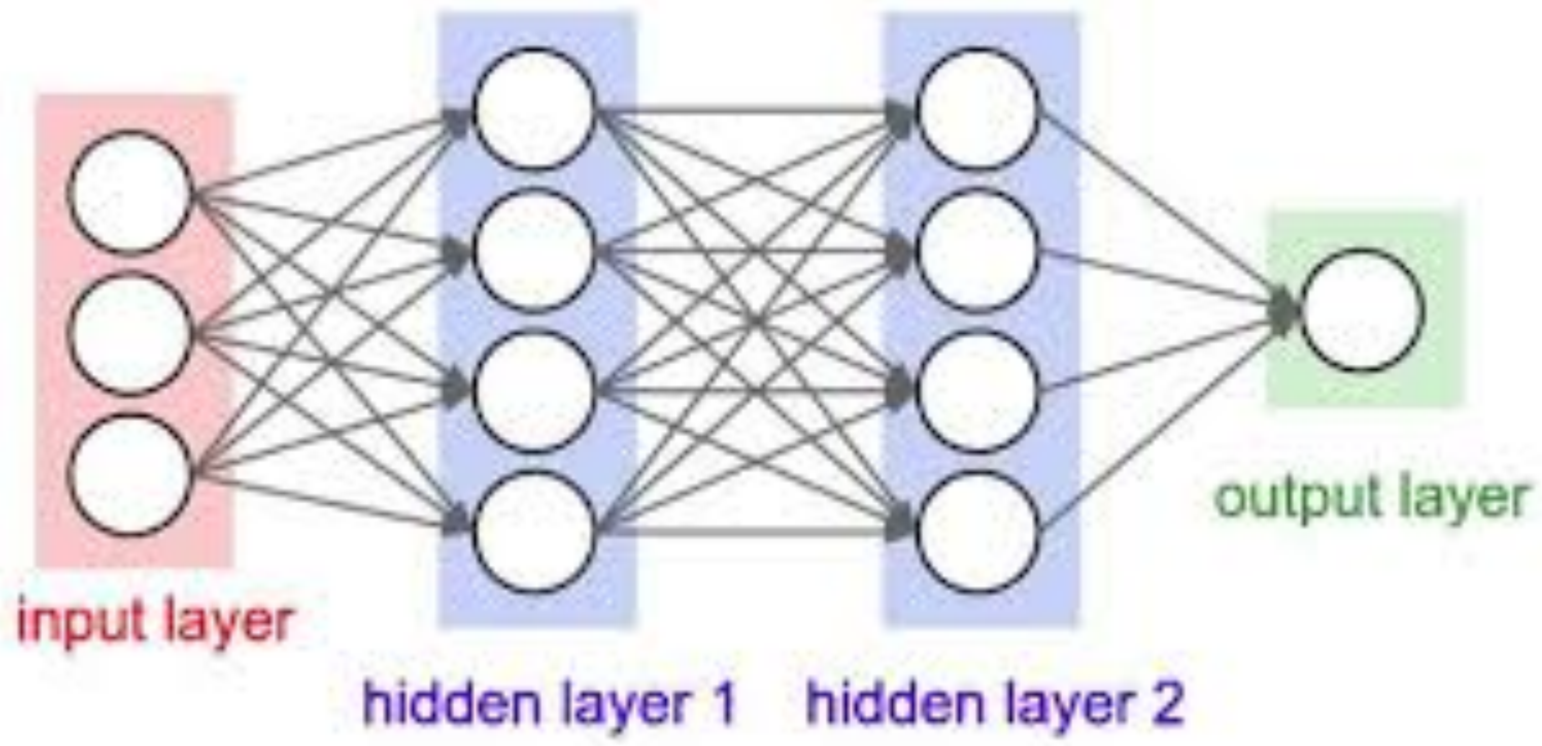


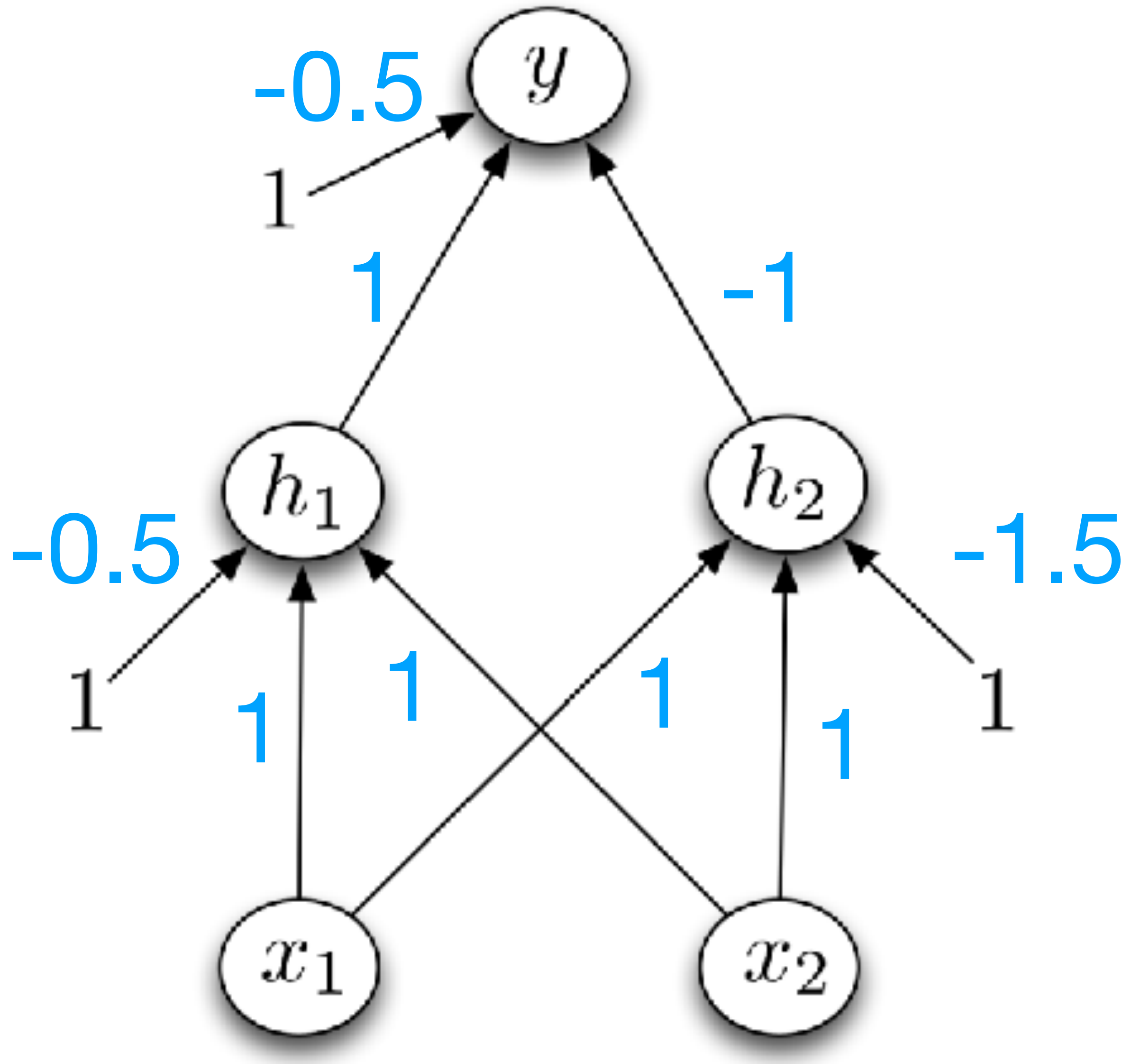
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	$\sigma(-.5) = 0$
1	1	$\sigma(1.5) = 1$	$\sigma(.5) = 1$	$\sigma(-.5) = 0$
1	0	$\sigma(.5) = 1$	$\sigma(-.5) = 0$	
0	1			

Multilayer Perceptron

- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR



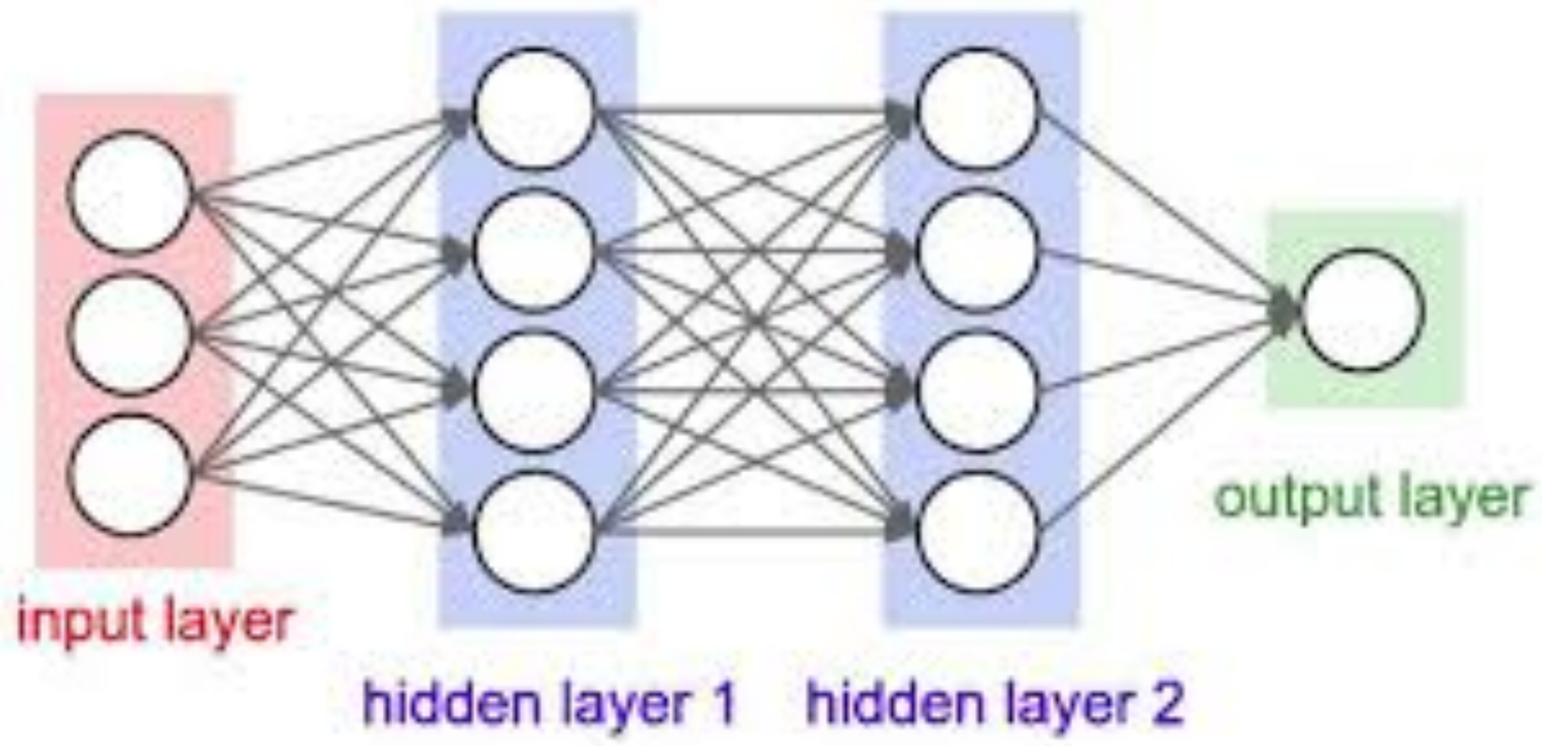


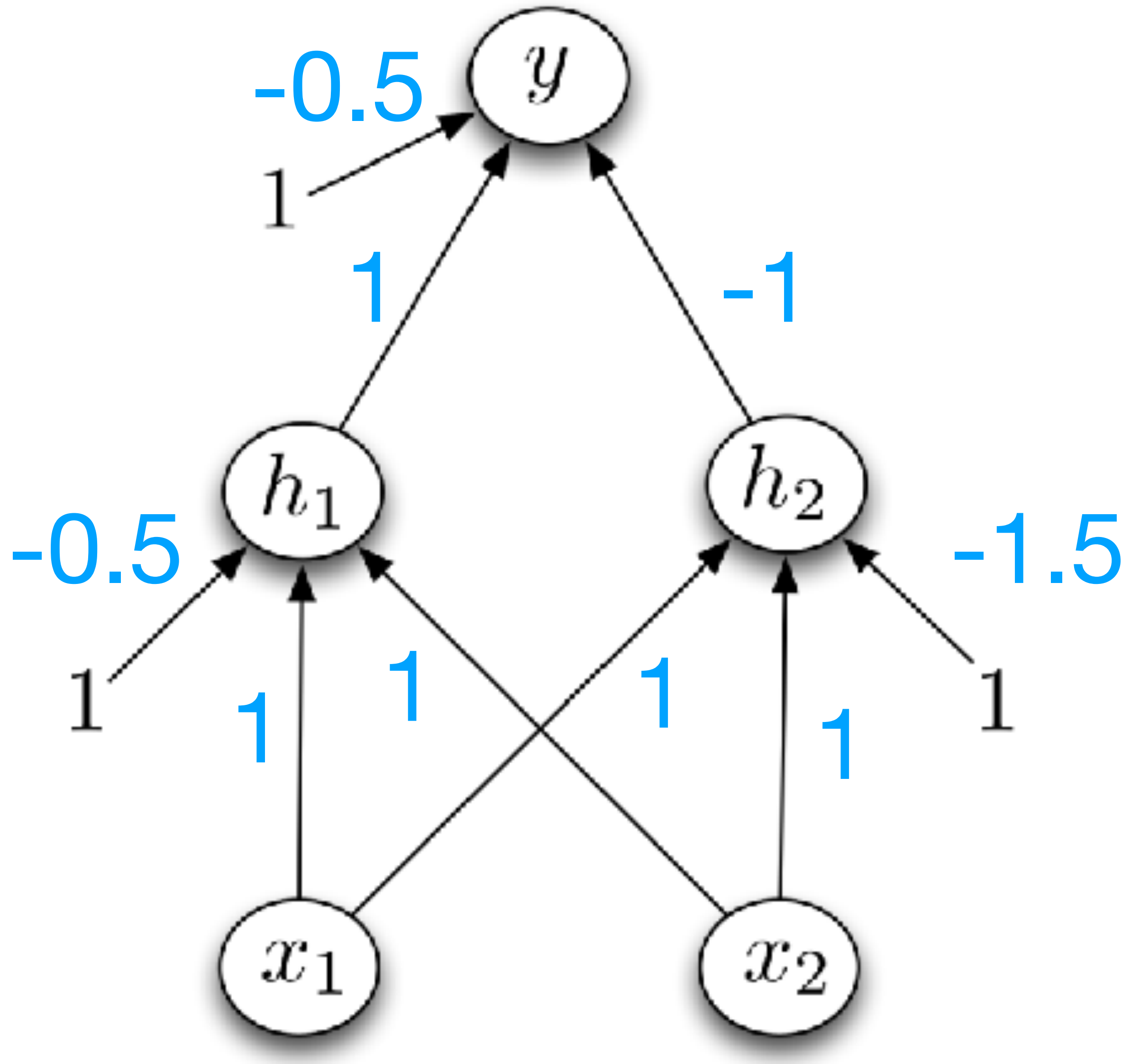
What are h_1, h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	$\sigma(-.5) = 0$
1	1	$\sigma(1.5) = 1$	$\sigma(.5) = 1$	$\sigma(-.5) = 0$
1	0	$\sigma(.5) = 1$	$\sigma(-.5) = 0$	$\sigma(.5) = 1$
0	1			

Multilayer Perceptron

- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR



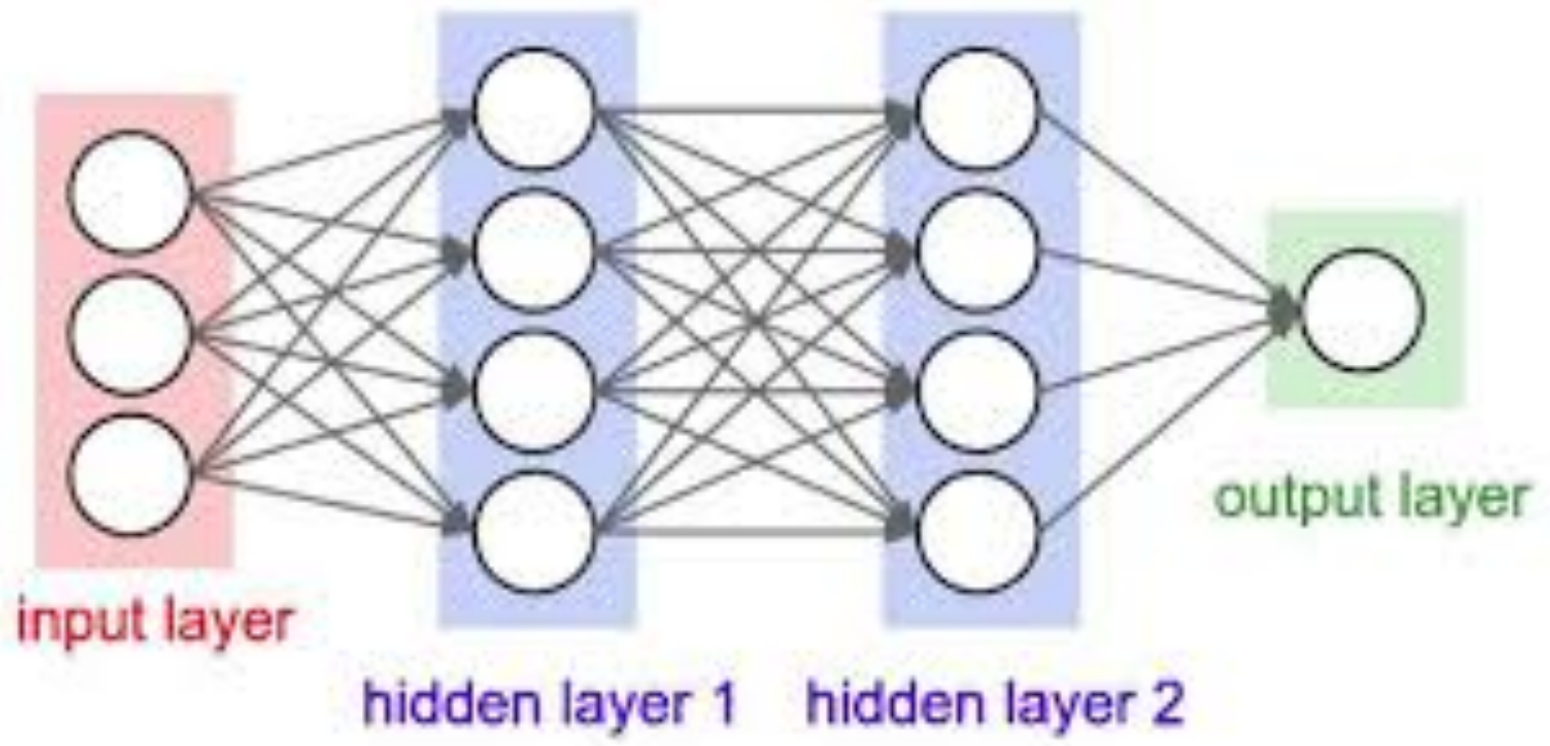


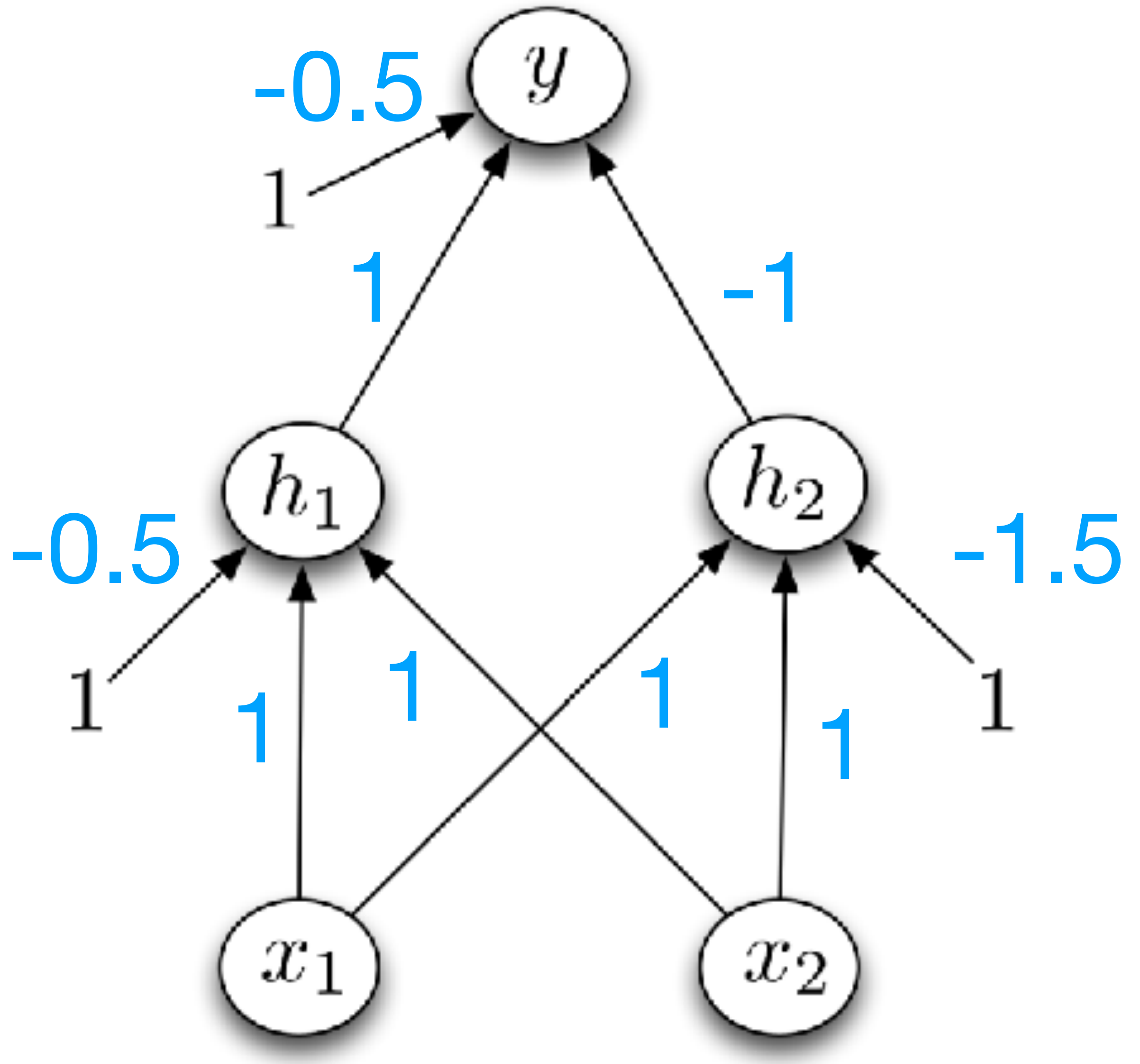
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	$\sigma(-.5) = 0$
1	1	$\sigma(1.5) = 1$	$\sigma(.5) = 1$	$\sigma(-.5) = 0$
1	0	$\sigma(.5) = 1$	$\sigma(-.5) = 0$	$\sigma(.5) = 1$
0	1	$\sigma(.5) = 1$		

Multilayer Perceptron

- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR



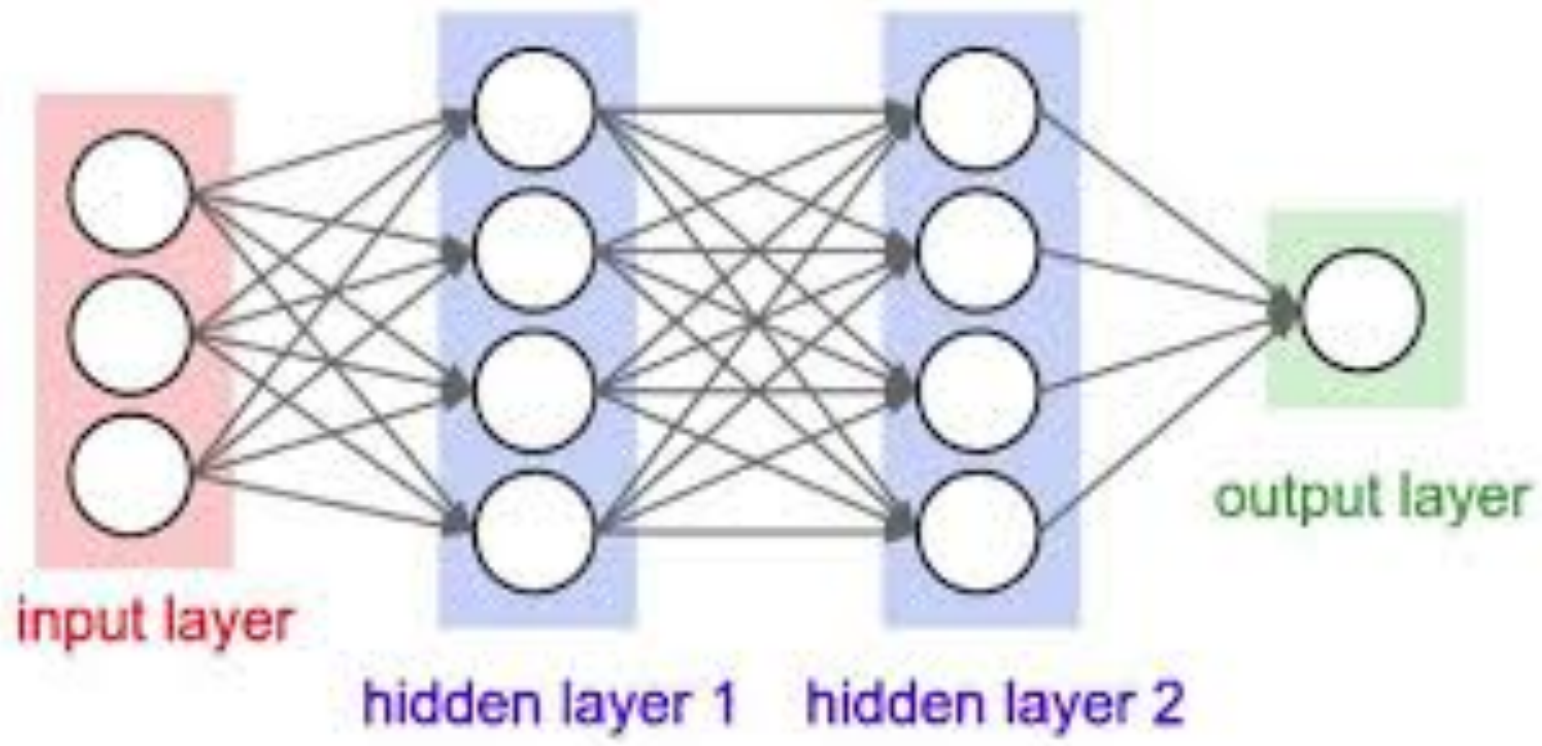


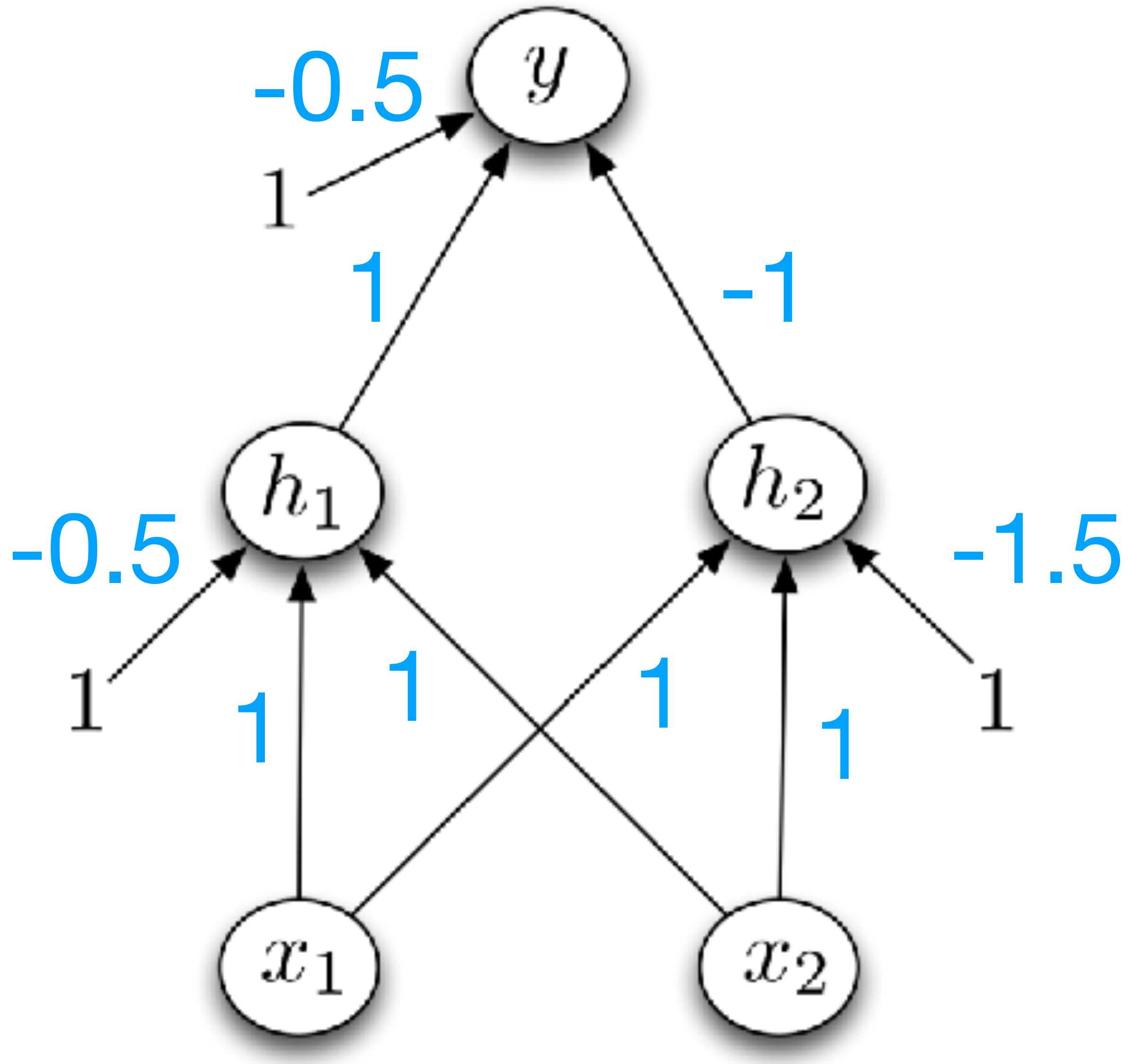
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	$\sigma(-.5) = 0$
1	1	$\sigma(1.5) = 1$	$\sigma(.5) = 1$	$\sigma(-.5) = 0$
1	0	$\sigma(.5) = 1$	$\sigma(-.5) = 0$	$\sigma(.5) = 1$
0	1	$\sigma(.5) = 1$	$\sigma(-.5) = 0$	

Multilayer Perceptron

- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR

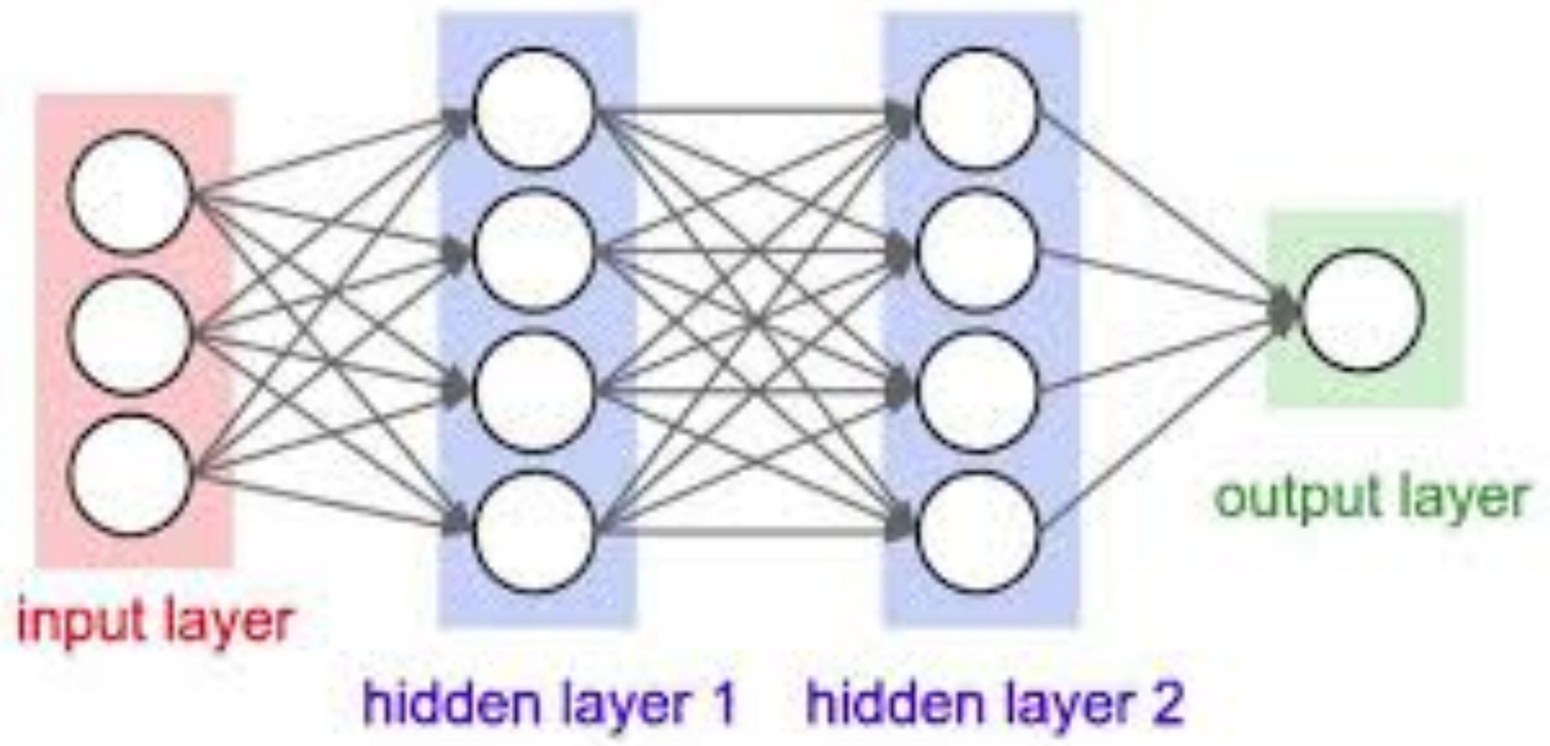




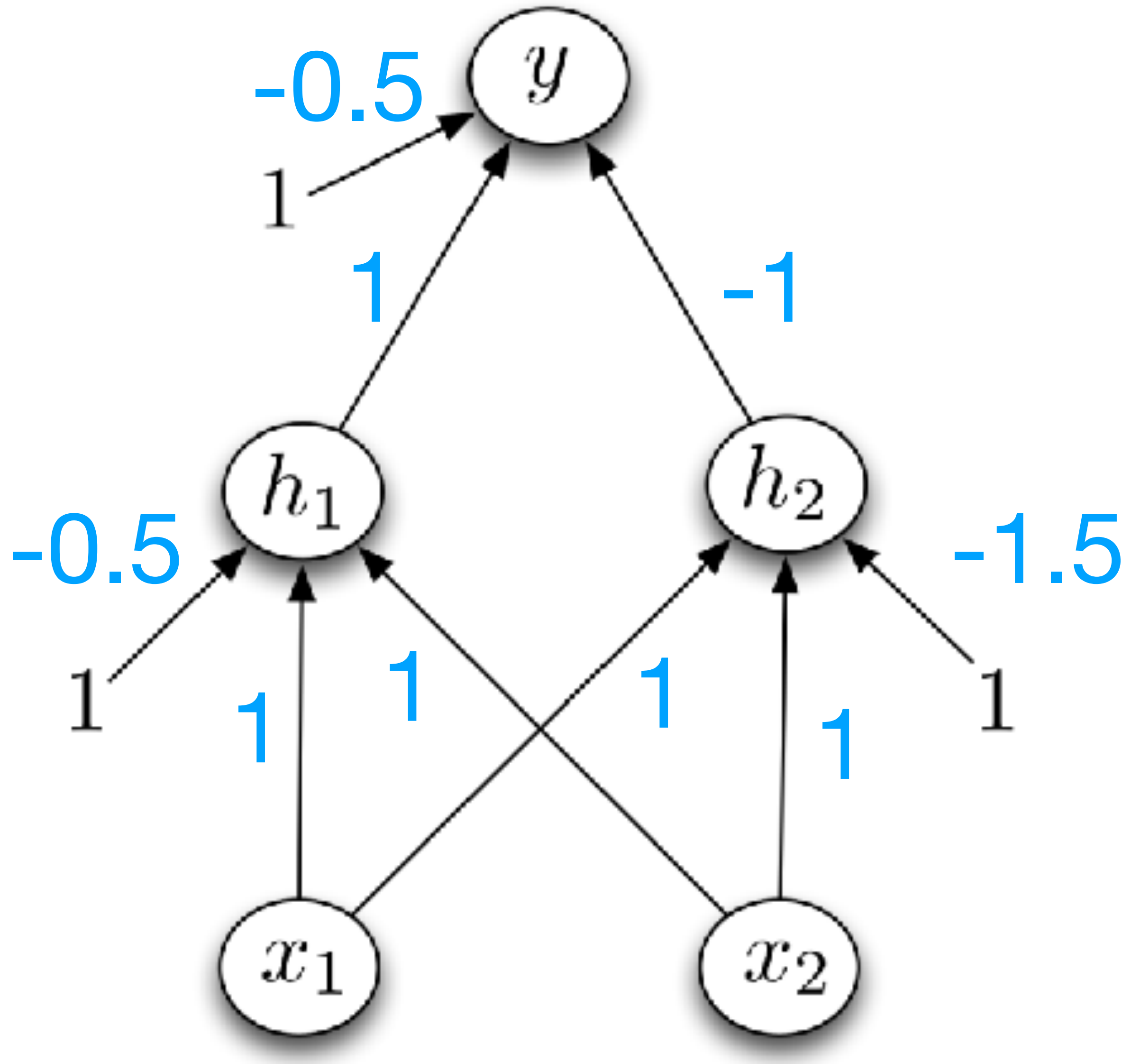
What are h_1,h_2, and y when:

x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	$\sigma(-.5) = 0$
1	1	$\sigma(1.5) = 1$	$\sigma(.5) = 1$	$\sigma(-.5) = 0$
1	0	$\sigma(.5) = 1$	$\sigma(-.5) = 0$	$\sigma(.5) = 1$
0	1	$\sigma(.5) = 1$	$\sigma(-.5) = 0$	$\sigma(.5) = 1$

Multilayer Perceptron



- MLPs are feedforward networks with multiple hidden layers, where we apply the same activation function at each layer
  - $h^{(1)} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
  - $h^{(i+1)} = \sigma(\mathbf{w}^\top \mathbf{h}^{(i)} + b)$
- A single hidden layer allows us to solve XOR



What are h_1,h_2, and y when:

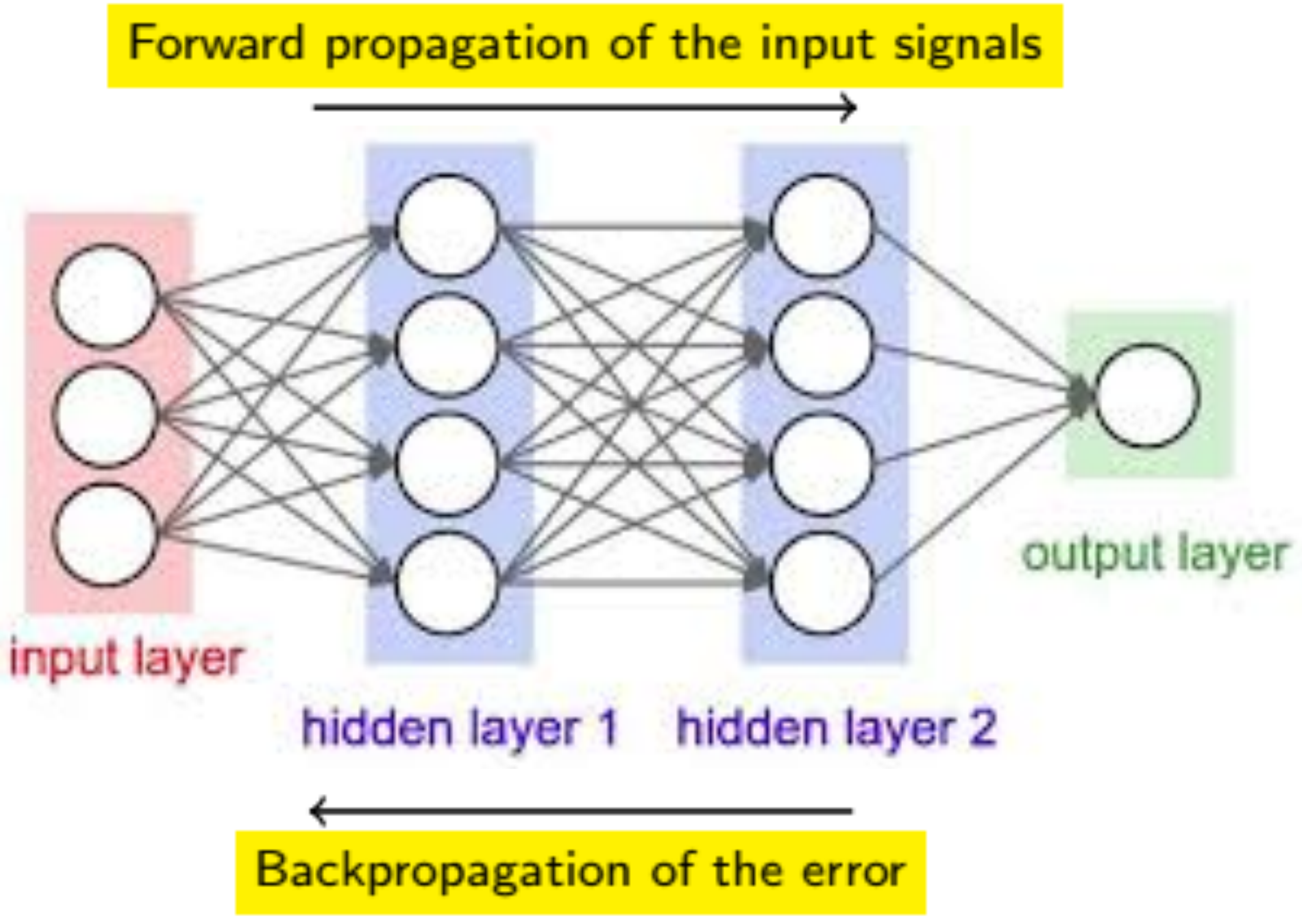
x_1	x_2	h_1	h_2	y
0	0	$\sigma(-.5) = 0$	$\sigma(-1.5) = 0$	$\sigma(-.5) = 0$
1	1	$\sigma(1.5) = 1$	$\sigma(.5) = 1$	$\sigma(-.5) = 0$
1	0	$\sigma(.5) = 1$	$\sigma(-.5) = 0$	$\sigma(.5) = 1$
0	1	$\sigma(.5) = 1$	$\sigma(-.5) = 0$	$\sigma(.5) = 1$

Historical note

- Rosenblatt introduced an MLP with 3 layers in 1962, but only the final layer had learning connections
- First deep learning MLP by Ivakhenko & Lapa (1965), with stochastic gradient descent added in 1967 by Shun'ichi Amari

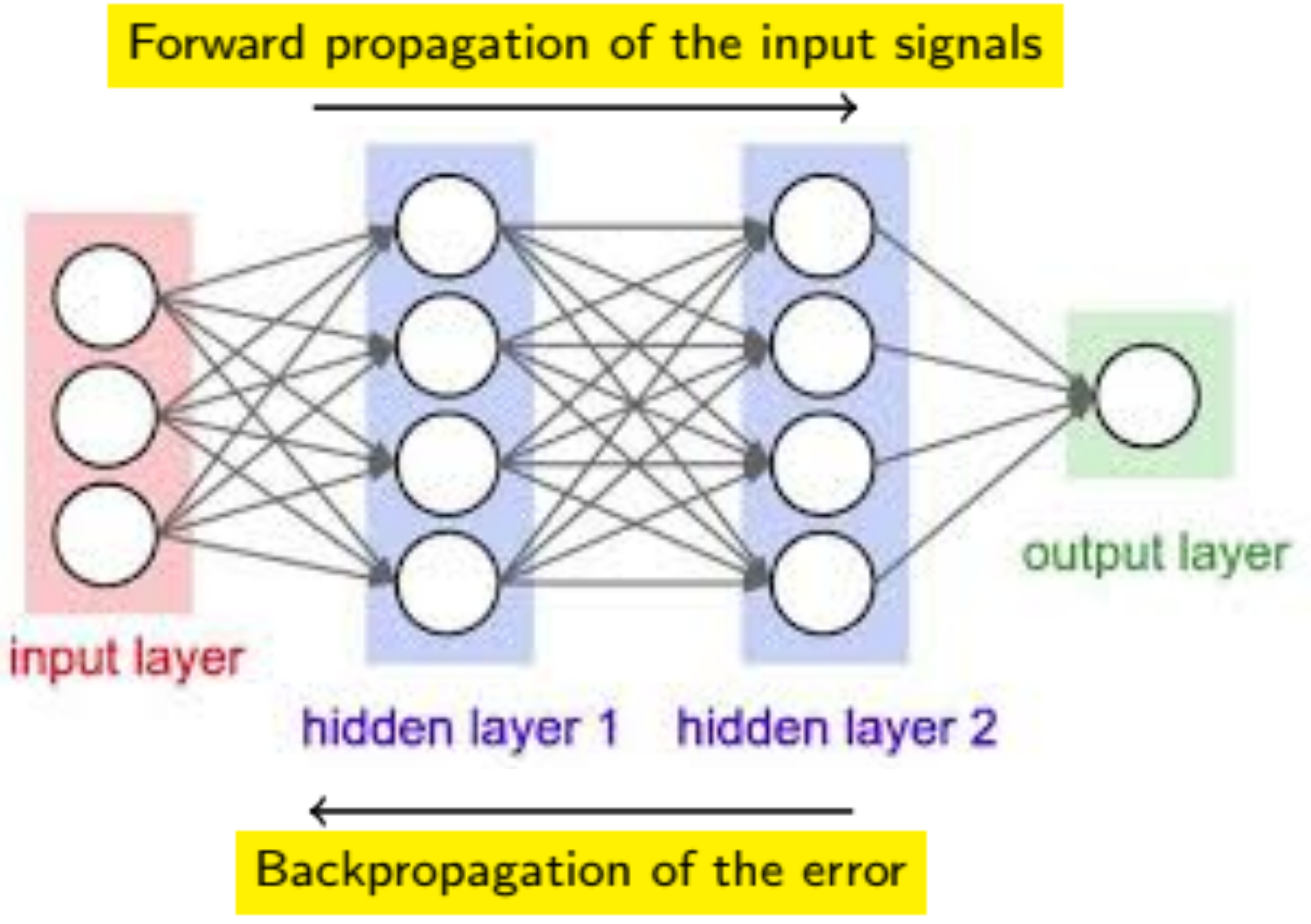
Backpropagation

- Introduced by Rosenblatt (1962), but he didn't know how to implement it*



Backpropagation

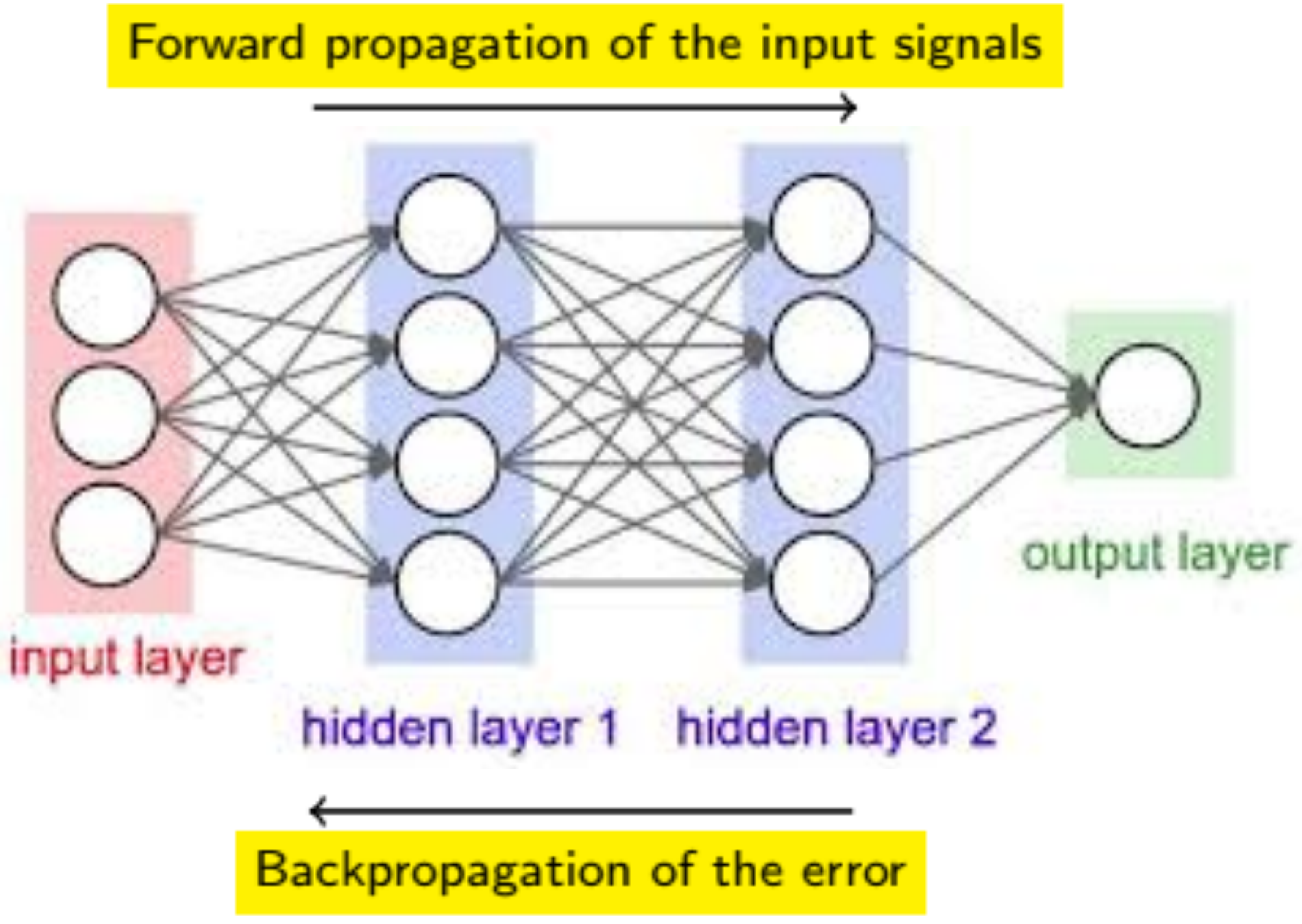
- Introduced by Rosenblatt (1962), but he didn't know how to implement it*



* his perceptrons had discrete outputs, thus no derivatives

Backpropagation

- Introduced by Rosenblatt (1962), but he didn't know how to implement it*
- **Goal:** update weights/bias to minimize **the loss**
  - direction of update is known as **the gradient**

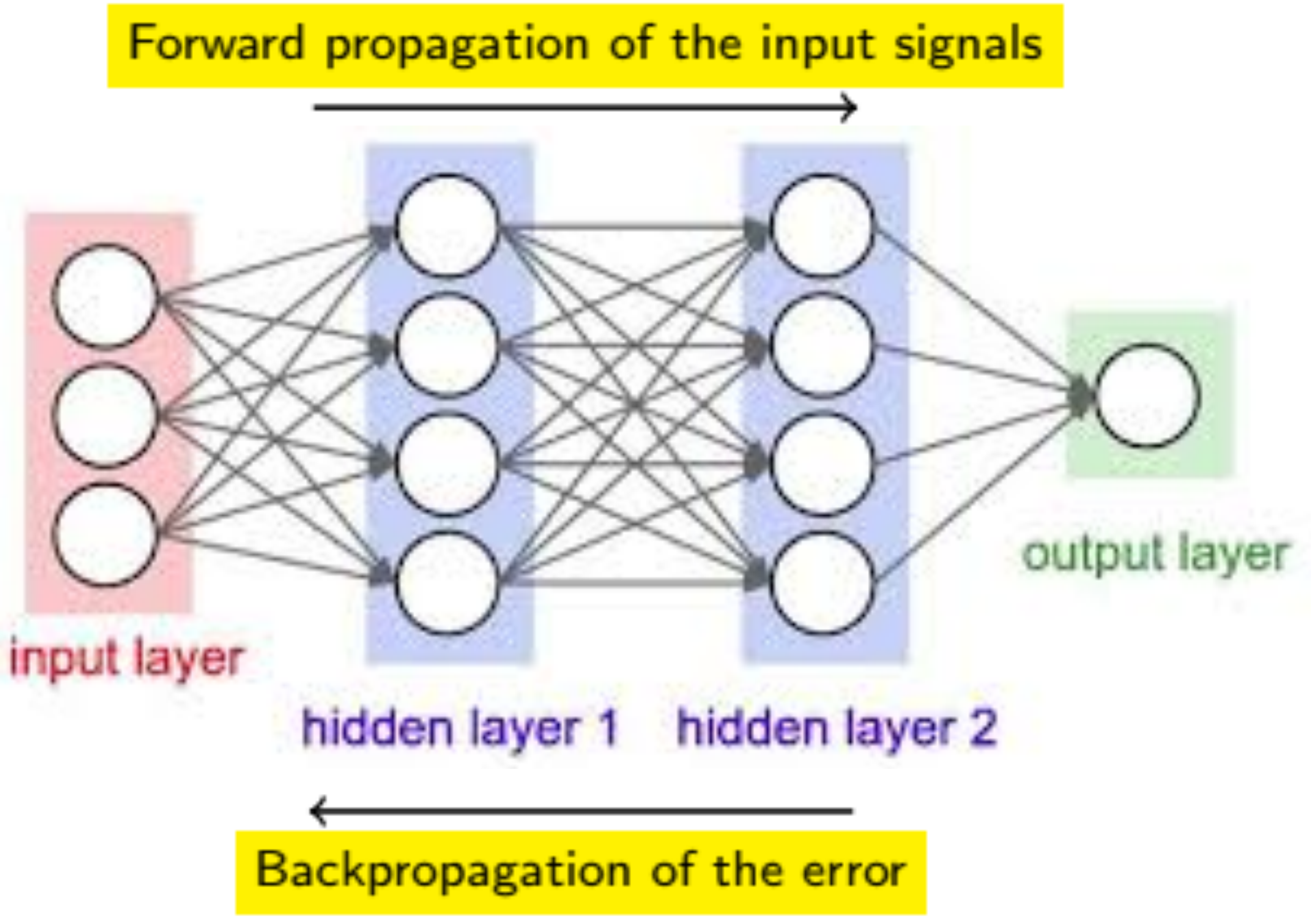


$$\mathscr{L} = \frac{1}{2}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

* his perceptrons had discrete outputs, thus no derivatives

Backpropagation

- Introduced by Rosenblatt (1962), but he didn't know how to implement it*
- **Goal:** update weights/bias to minimize **the loss**
  - direction of update is known as **the gradient**



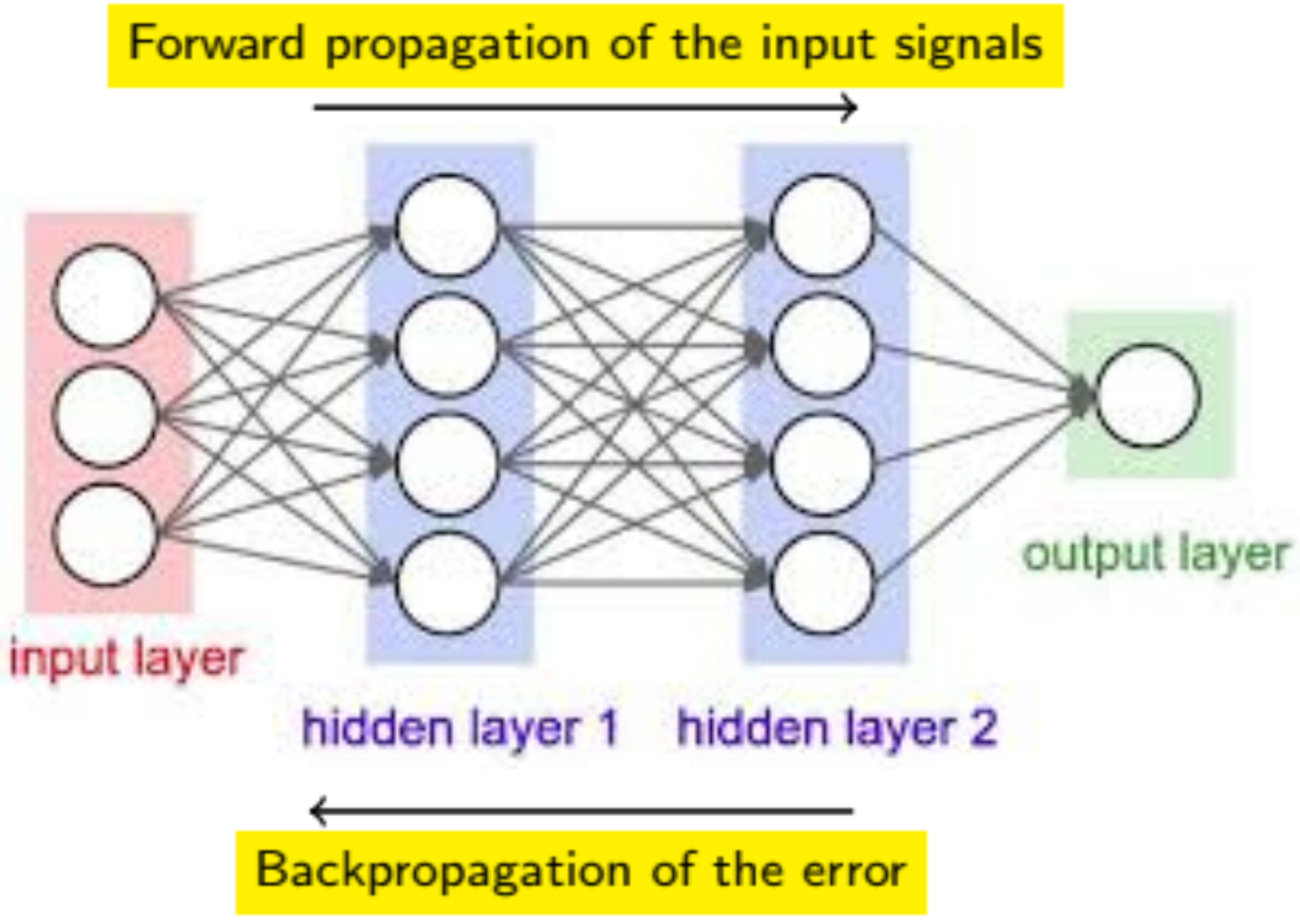
$$\mathscr{L} = \frac{1}{2}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

$$w \leftarrow w - \alpha \frac{\partial \mathscr{L}}{\partial w}$$

* his perceptrons had discrete outputs, thus no derivatives

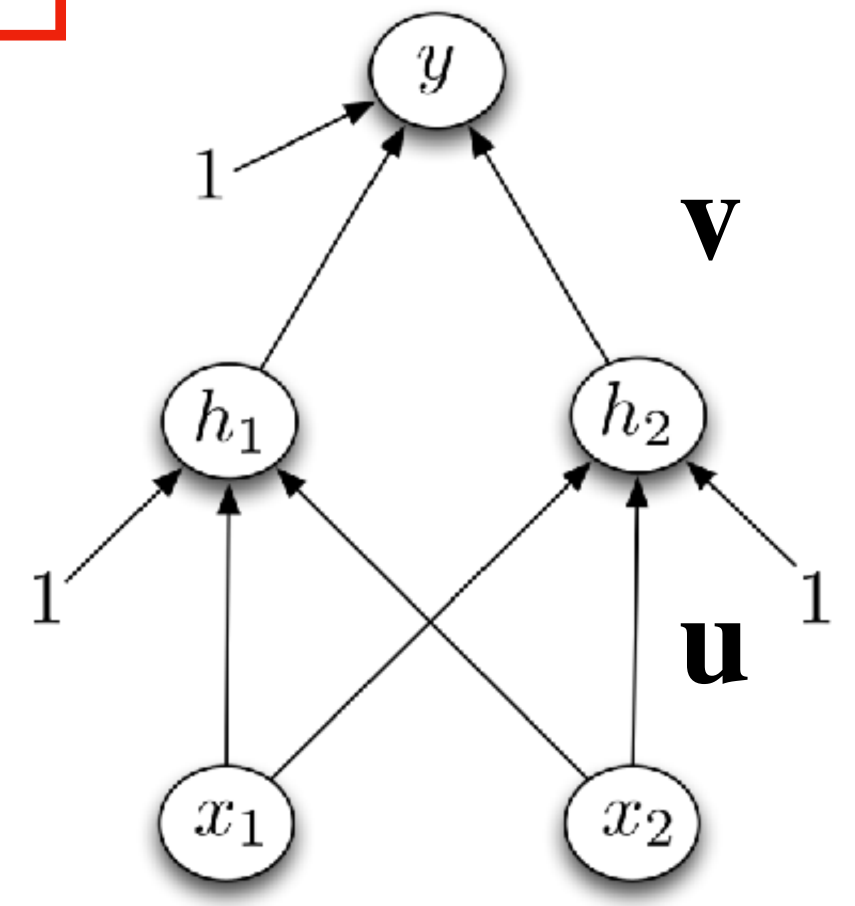
Backpropagation

- Introduced by Rosenblatt (1962), but he didn't know how to implement it*
- **Goal:** update weights/bias to minimize **the loss**
  - direction of update is known as **the gradient**
- Since MLPs are composed of recursive functions at each layer, we can apply the **chain rule** (Leibniz, 1676) to compute the gradient layer by layer, moving backwards through the network:
  - $\frac{\partial \mathscr{L}}{\partial \mathbf{u}} = \frac{\partial \mathscr{L}}{\partial \mathbf{v}} \frac{\partial \mathbf{v}}{\partial \mathbf{u}}$
  - We use the error to first update the $\mathbf{v}$ weights, and then update $\mathbf{u}$ weights w.r.t. how they change $\mathbf{v}$



$$\mathscr{L} = \frac{1}{2} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

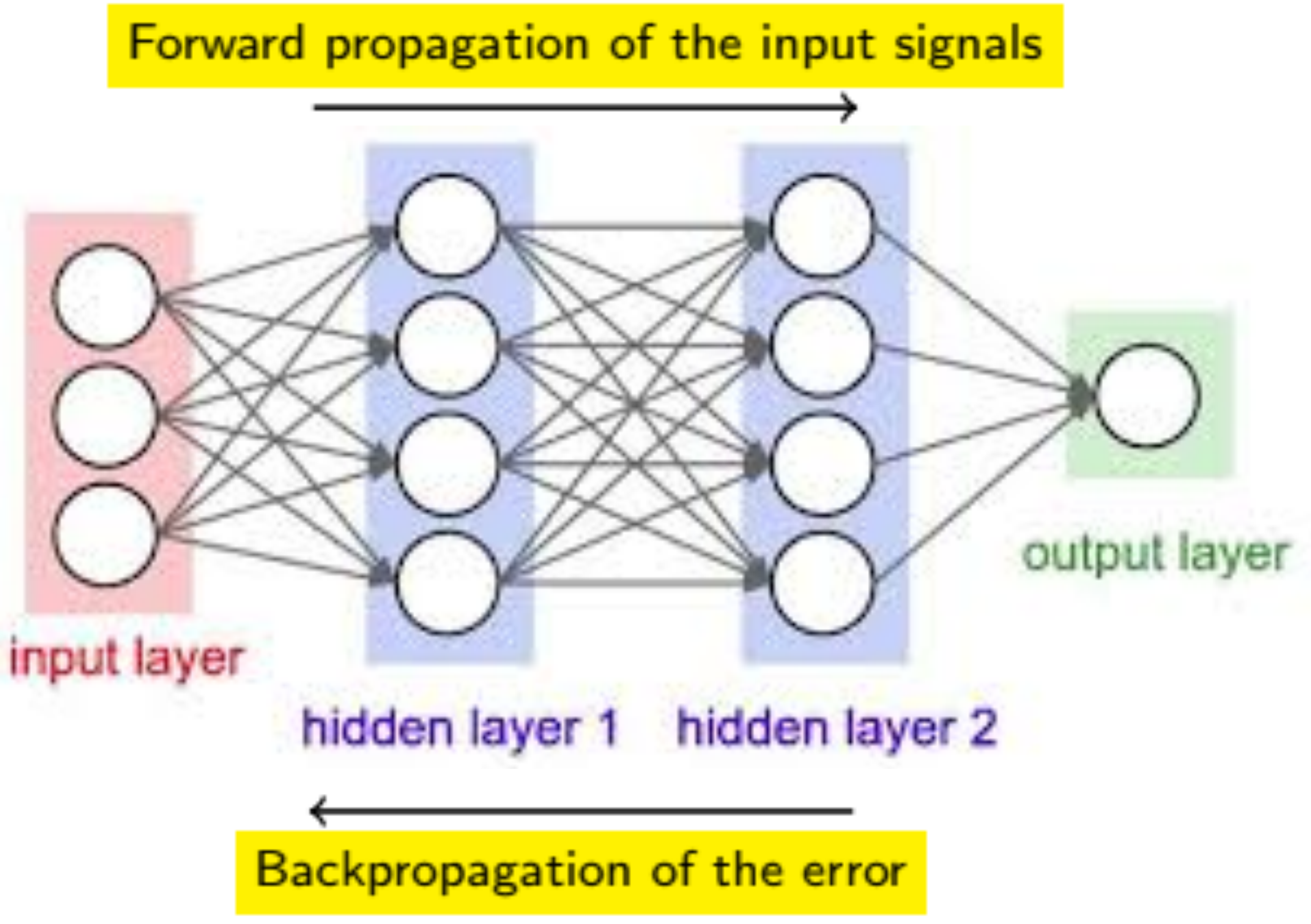
$$w \leftarrow w - \alpha \frac{\partial \mathscr{L}}{\partial w}$$



* his perceptrons had discrete outputs, thus no derivatives

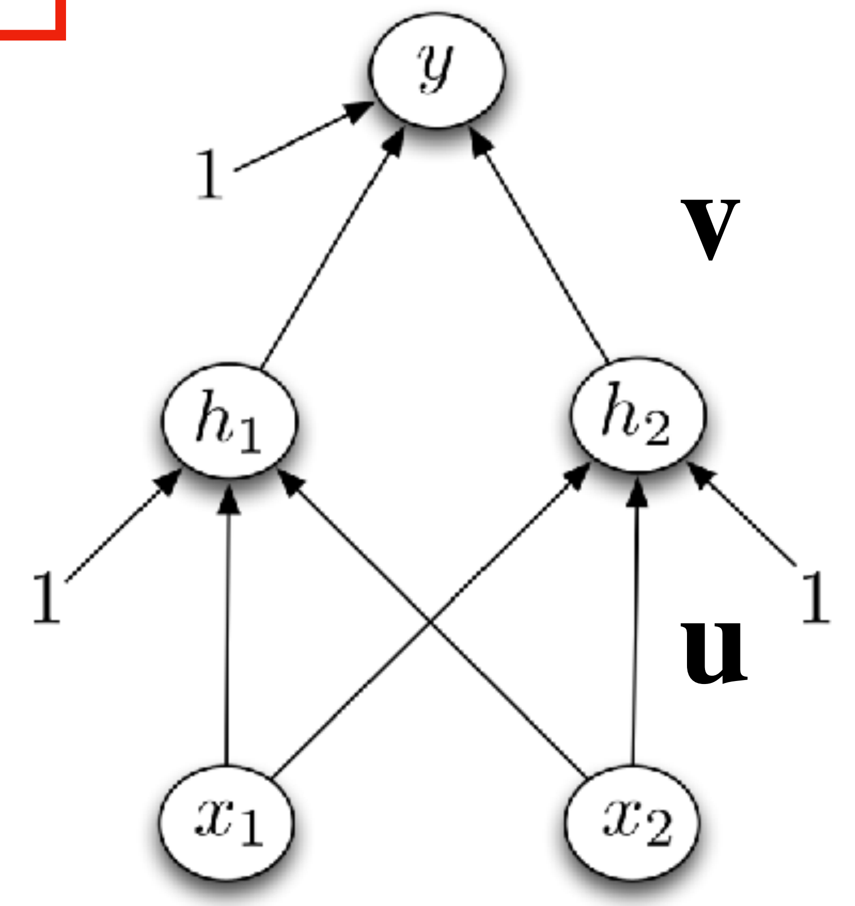
Backpropagation

- Introduced by Rosenblatt (1962), but he didn't know how to implement it*
- **Goal:** update weights/bias to minimize **the loss**
  - direction of update is known as **the gradient**
- Since MLPs are composed of recursive functions at each layer, we can apply the **chain rule** (Leibniz, 1676) to compute the gradient layer by layer, moving backwards through the network:
  - $\frac{\partial \mathscr{L}}{\partial \mathbf{u}} = \frac{\partial \mathscr{L}}{\partial \mathbf{v}} \frac{\partial \mathbf{v}}{\partial \mathbf{u}}$
  - We use the error to first update the $\mathbf{v}$ weights, and then update $\mathbf{u}$ weights w.r.t. how they change $\mathbf{v}$
- For further reading, see Grosse & Ba (CSC421)



$$\mathscr{L} = \frac{1}{2} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

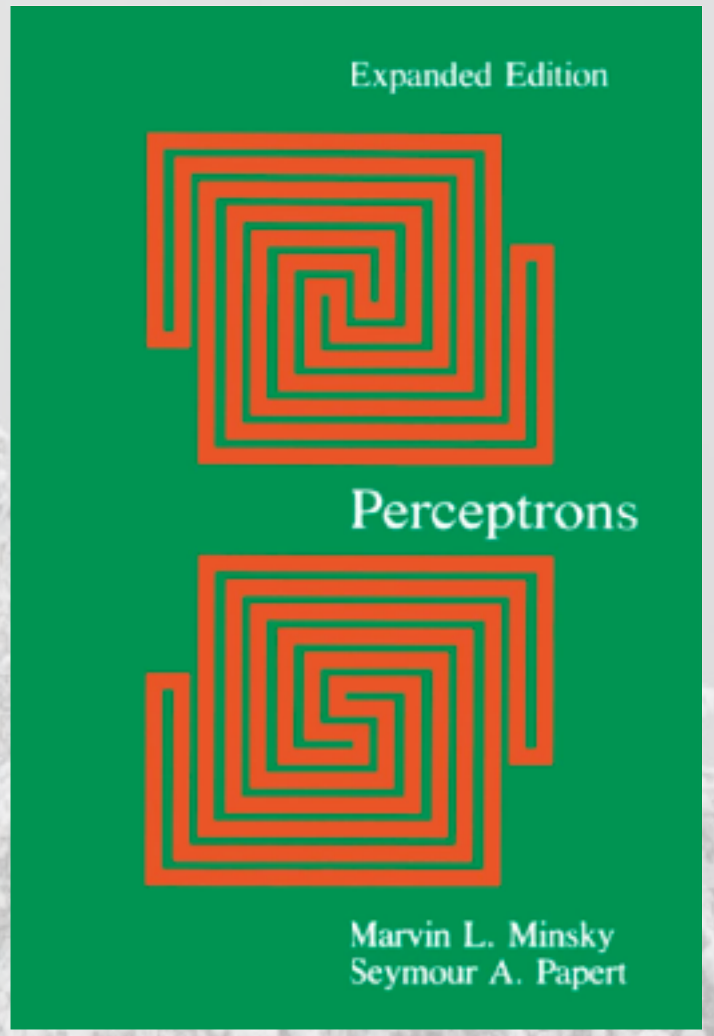
$$w \leftarrow w - \alpha \frac{\partial \mathscr{L}}{\partial w}$$

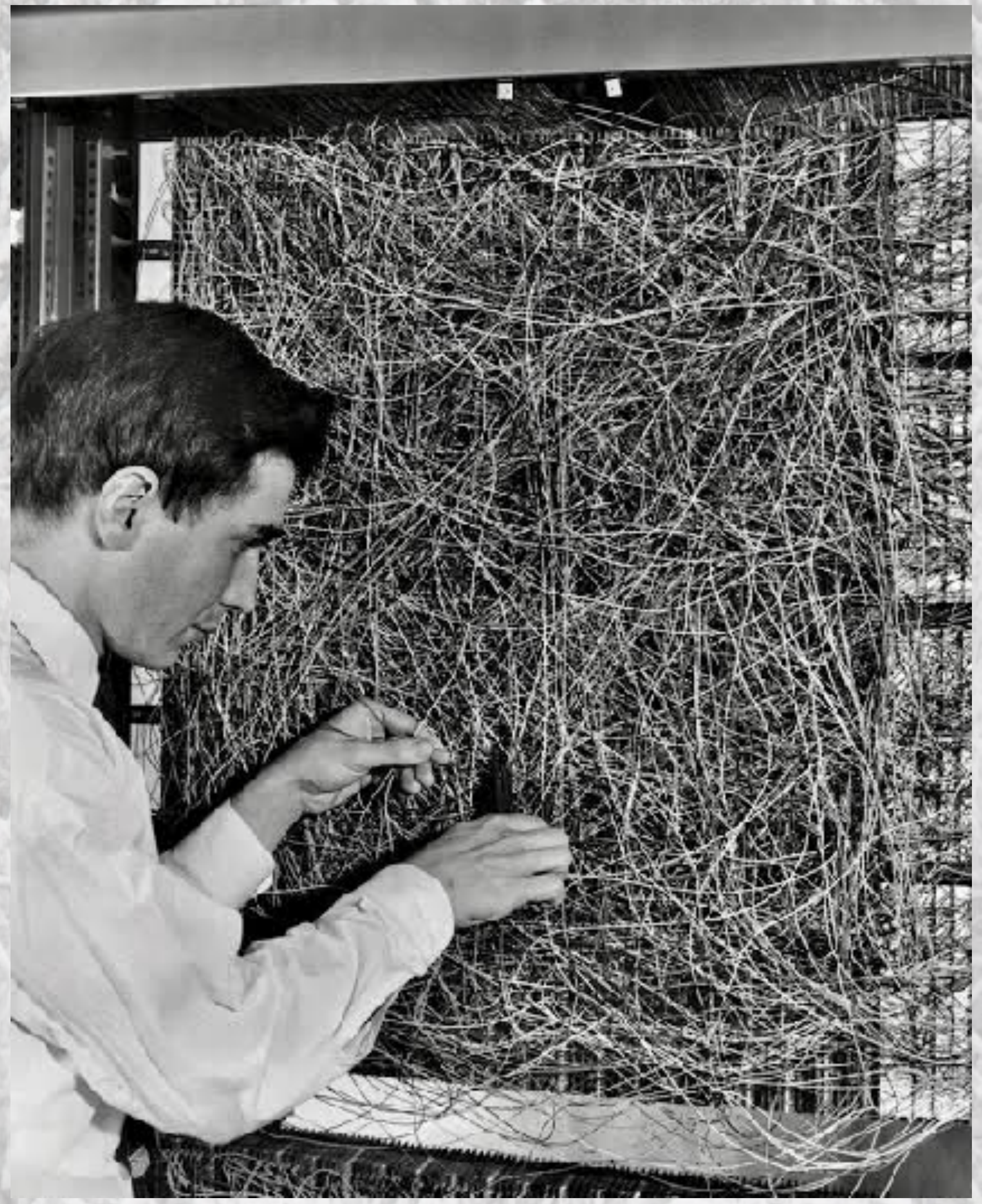


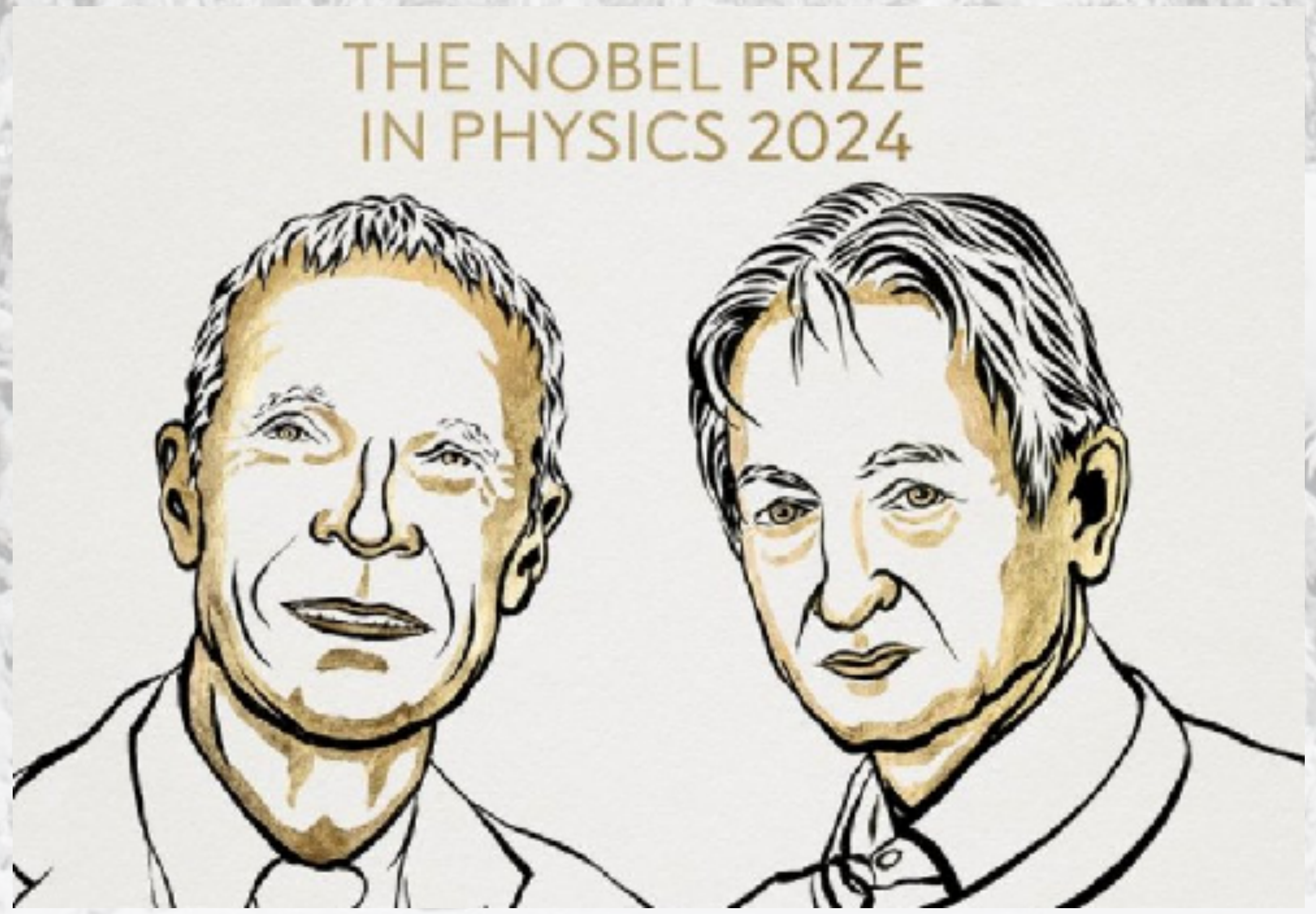
* his perceptrons had discrete outputs, thus no derivatives

The AI winter

- Minsky & Papert’s (1969) critique of Perceptrons being unable to solve XOR problems was taken as a fundamental limitation
- Funding and interest in AI research dried up
- In 1971 Frank Rosenblatt died in a trajic boating accident
- It wouldn’t be until the 1980s when people like John Hopfield and David Rumelhart would revive interest







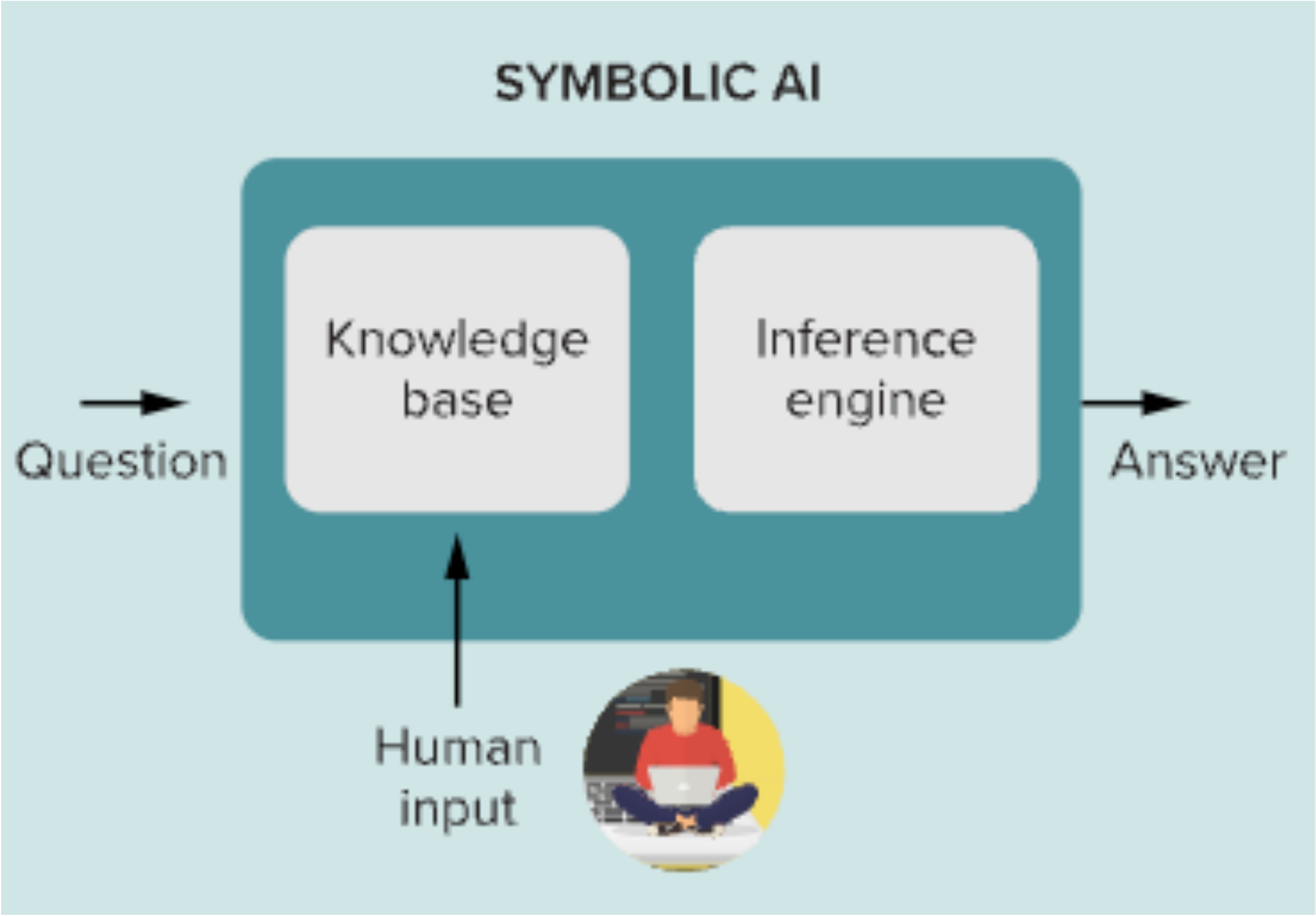
Connectionism: Summary

- **Perceptrons** can learn a number of logical operations, but fail at problems that are not linearly separable (e.g, XOR)
- **Rosenblatt's** learning rule is guaranteed to converge (for linearly separable problems), but is brittle with noisy training data
    - ADALINE offers a more robust learning rule, which is equivalent to stochastic gradient descent
- **Multilayer Perceptrons** are capable of solving XOR and other non-linearly separable problems
- **Backpropogation** is necessary for learning in MLPs, by passing the gradient across multiple layers using the chain rule

General Principles

- Incrementally improve predictions by reducing error
  - The unit of learning is the magnitude of the prediction error (Delta-rule)
    - Rescorla-Wagner model and ADALINE
    - But more generally, stochastic gradient descent, backpropogation, and all modern RL use this principle
- Incremental learning is not always guaranteed to succeed
  - Behavioral shaping can help guide learning towards desired outcomes
  - Single layer perceptrons are limited in which types of problems they can solve
    - Adding more layers helps, but it took a long time to develop learning rules
  - Gradient descent can get stuck in local optima
- What other principles have you picked up?

Next week we will look at what happened during the AI winter and explore the limits of stimulus-response learning



Symbolic AI

- What happened during the AI winter?
- Intelligence as manipulating symbols through rules and logical operations
- Learning as search

Cognitive Maps

- From Stimulus-Response learning to Stimulus-Stimulus learning
- Constructing a mental representation of the environment
- Neurological evidence for cognitive maps in the brain