STAT 992: Foundation Models for Biomedical Data

Ben Lengerich

Lecture 02: A Brief Introduction to Deep Learning

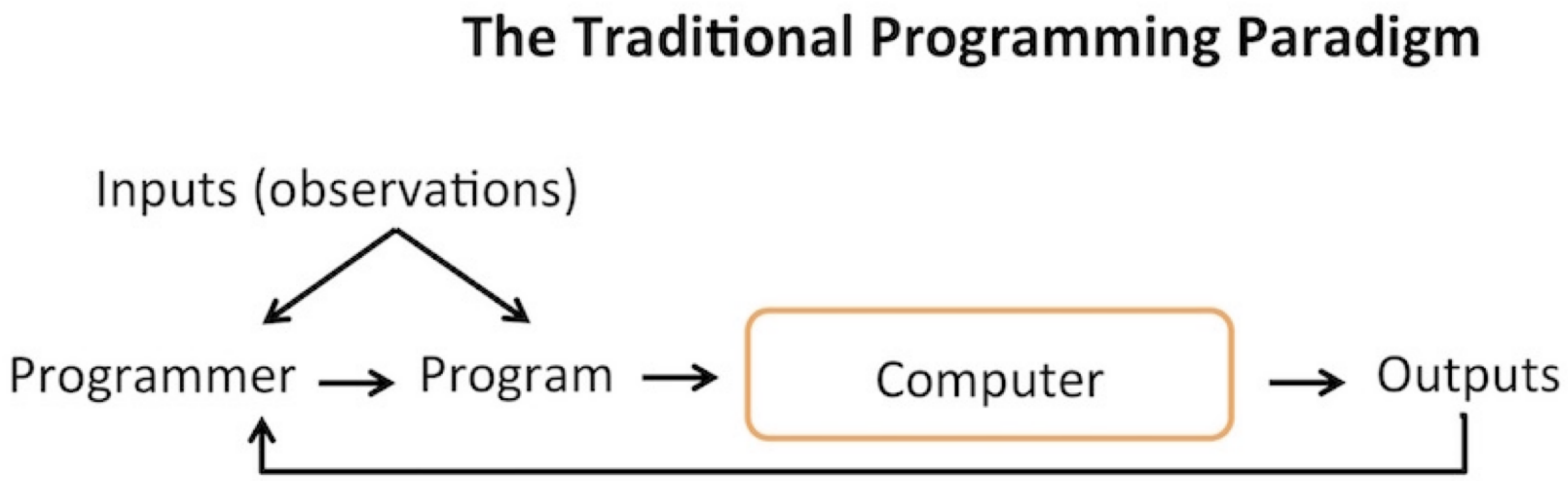
January 26, 2026

Today

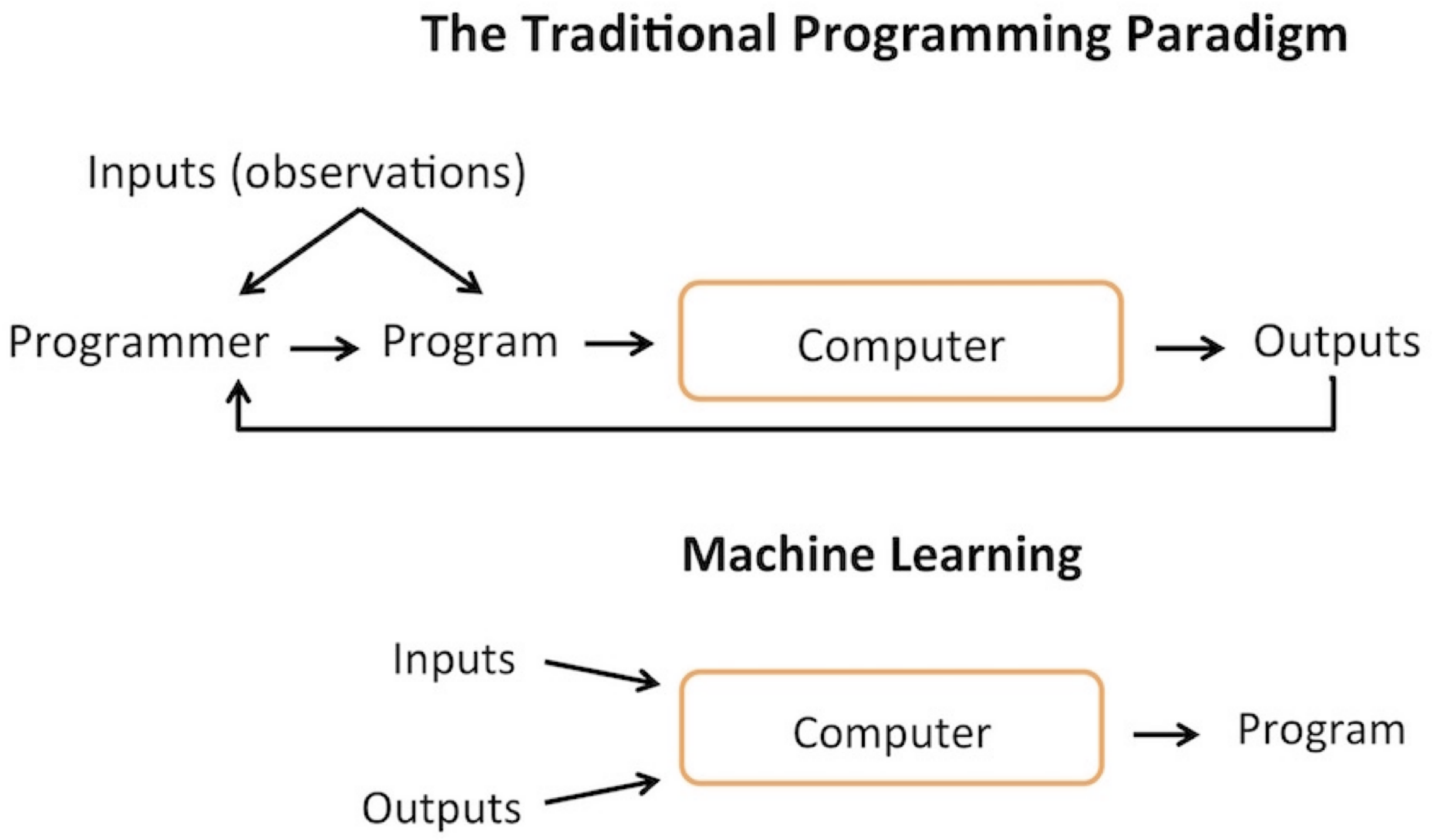
1. What is Machine Learning?
2. Three Types of Machine Learning
3. Some Necessary Jargon
4. The building blocks of Deep Learning Architectures
    1. Perceptron
    2. Logistic Regression
    3. Multilayer Perceptron
5. Training Deep Models
    1. Backpropagation
    2. Automated Differentiation

1. What is Machine Learning?

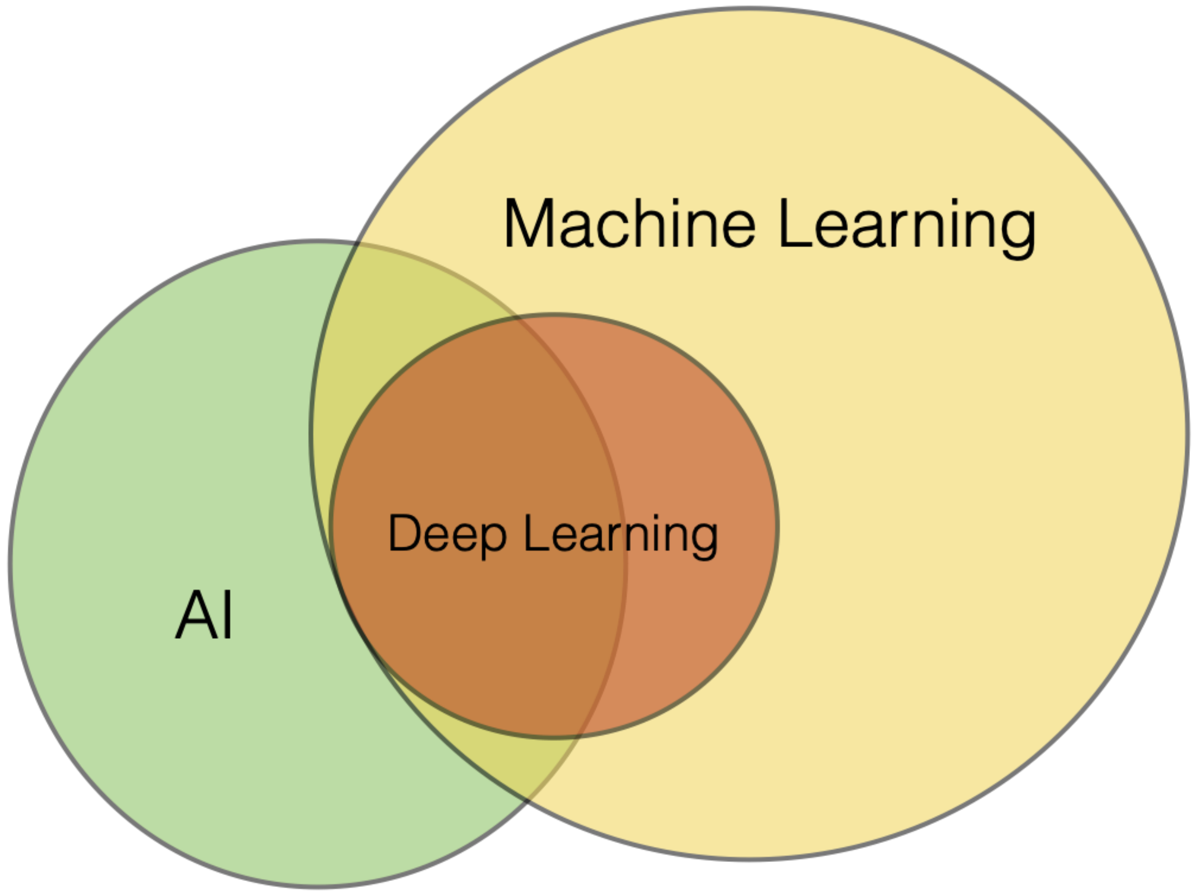
What is Machine Learning?



What is Machine Learning?



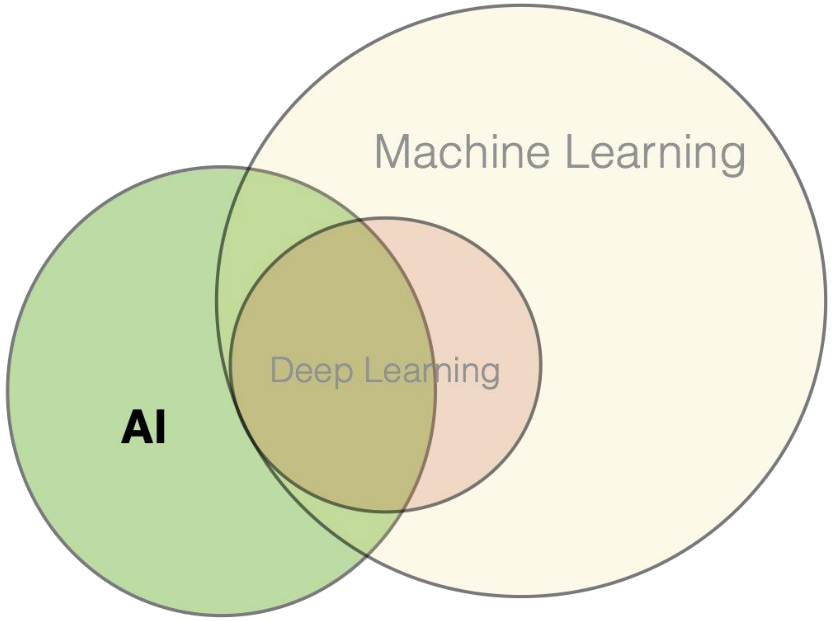
The Connection Between Fields



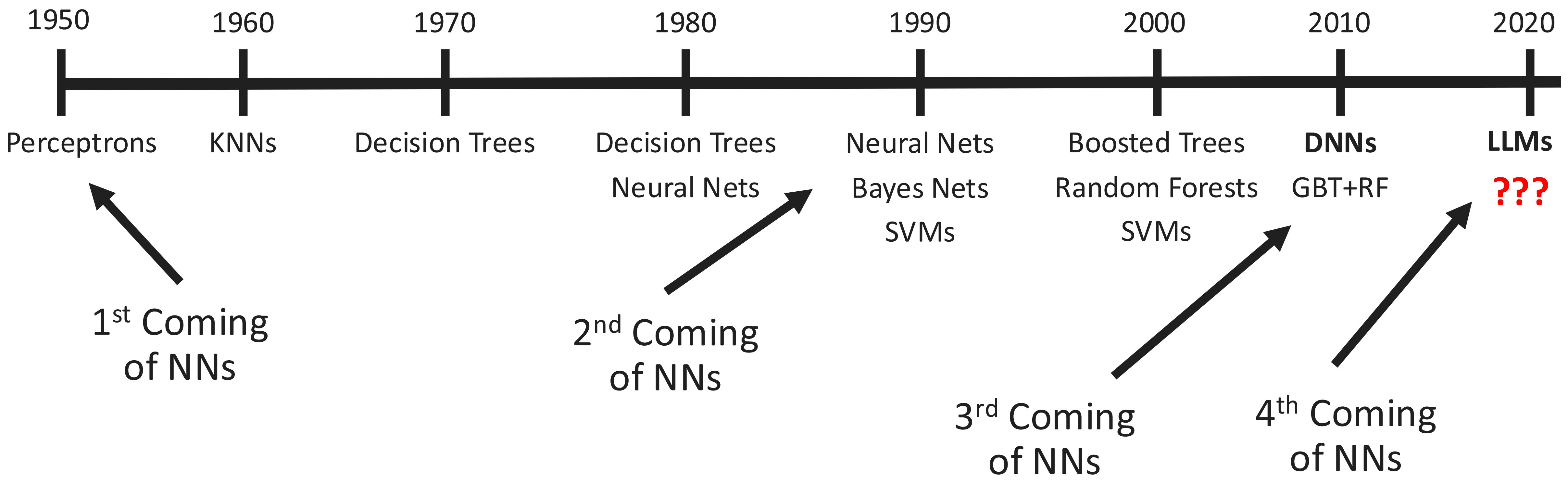
Not all AI Systems involve Machine Learning

Deep Blue used custom VLSI chips to execute the **alpha-beta search** algorithm in parallel, an example of GOFAI (Good Old-Fashioned Artificial Intelligence).

Image Source: https://mashable.com/2016/02/10/kasparov-deep-blue/

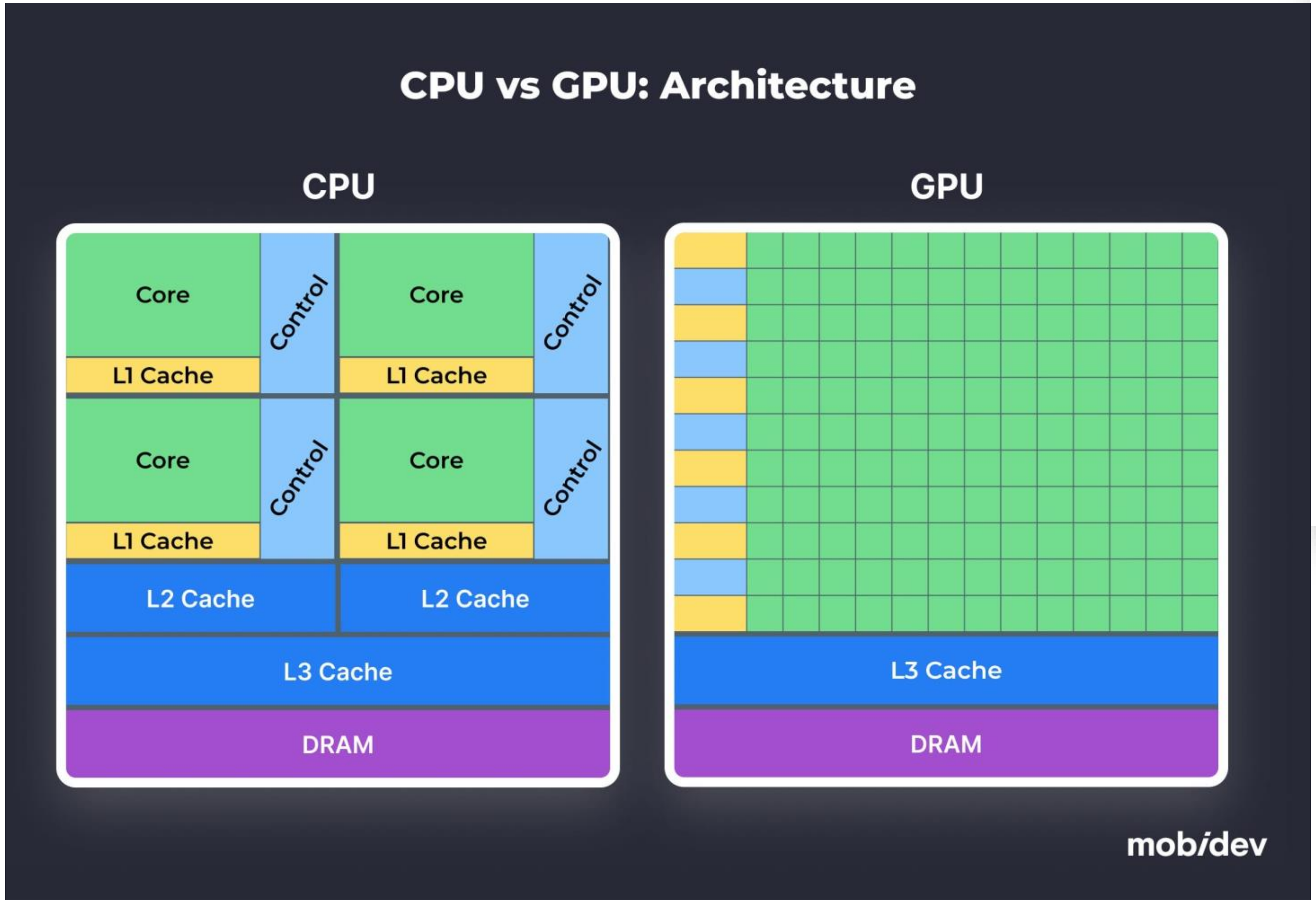


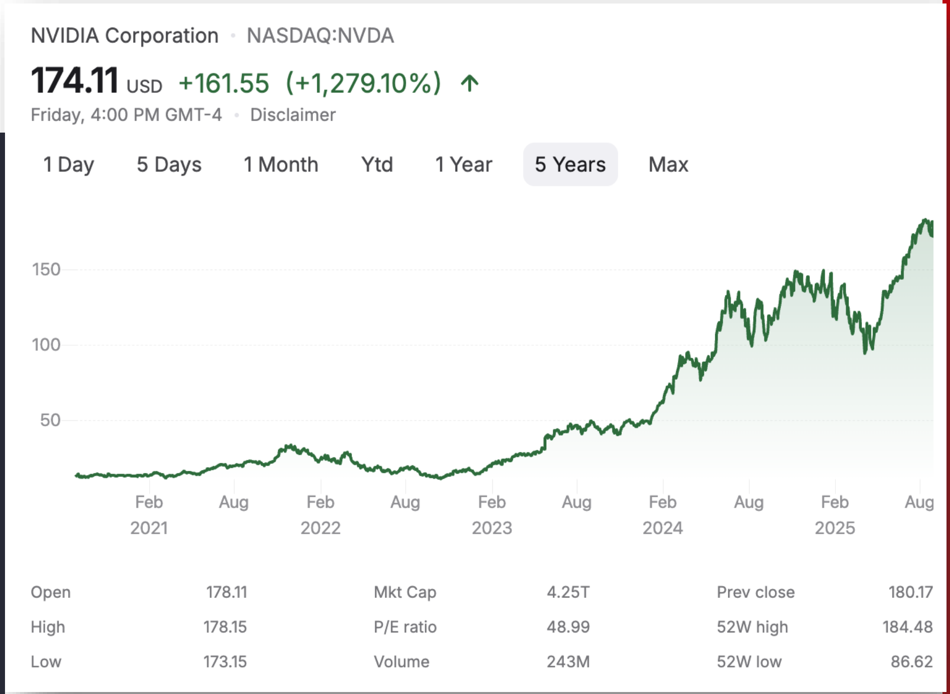
A Brief History of AI



Courtesy of Rich Caruana

The AI Revolution: Matrix Multiplications




NVIDIA Corporation · NASDAQ:NVDA

174.11 USD +161.55 (+1,279.10%) ↑

Friday, 4:00 PM GMT-4 · Disclaimer

Open	178.11	Mkt Cap	4.25T	Prev close	180.17
High	178.15	P/E ratio	48.99	52W high	184.48
Low	173.15	Volume	243M	52W low	86.62

https://mobidev.biz/blog/gpu-machine-learning-on-premises-vs-cloud

2. Three Types of Machine Learning

3 Broad Categories of ML

Supervised Learning	Labeled data Direct feedback Predict outcome/future
Unsupervised Learning	No labels/targets No feedback Find hidden structure in data
Reinforcement Learning	Decision process Reward system Learn series of actions

Source: Raschka and Mirjalily (2019). *Python Machine Learning, 3rd Edition*

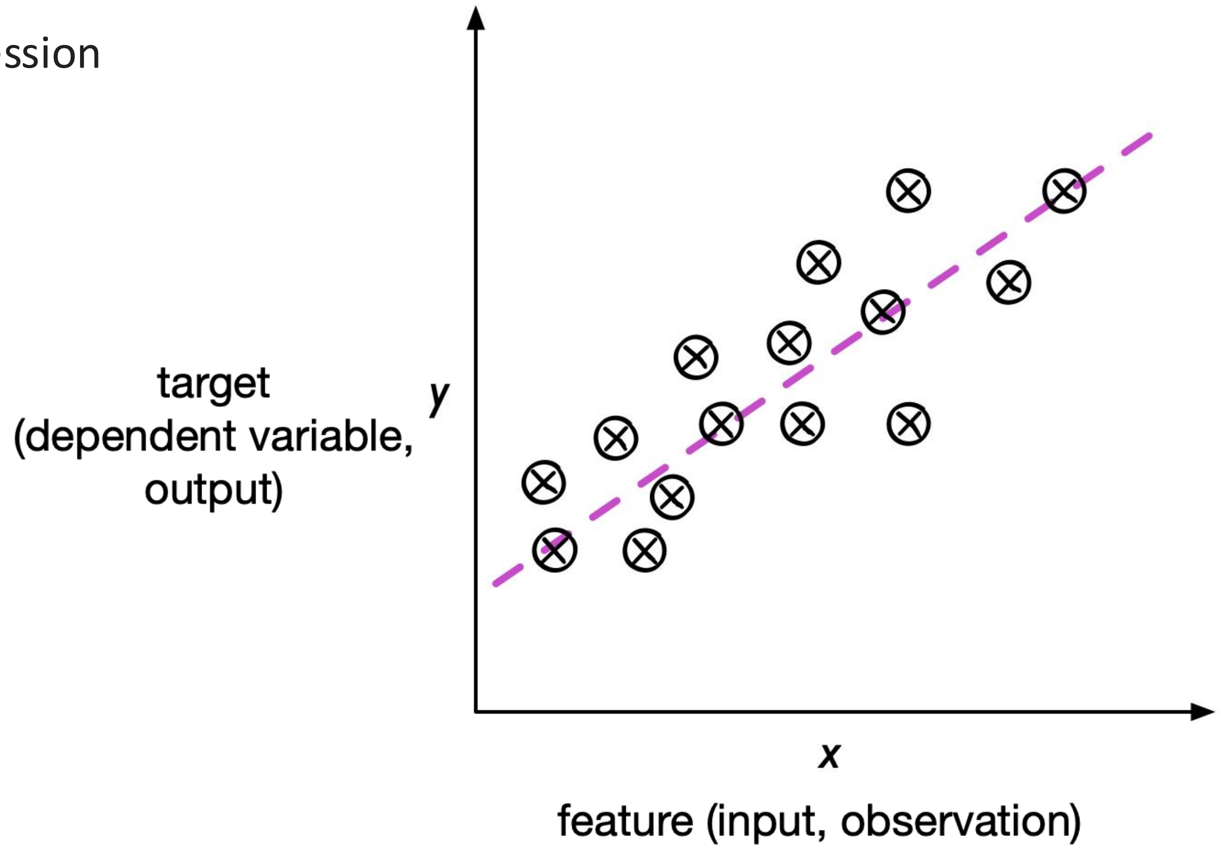
3 Broad Categories of ML

Supervised Learning

- Labeled data
- Direct feedback
- Predict outcome/future

Supervised Learning

Ex: Regression

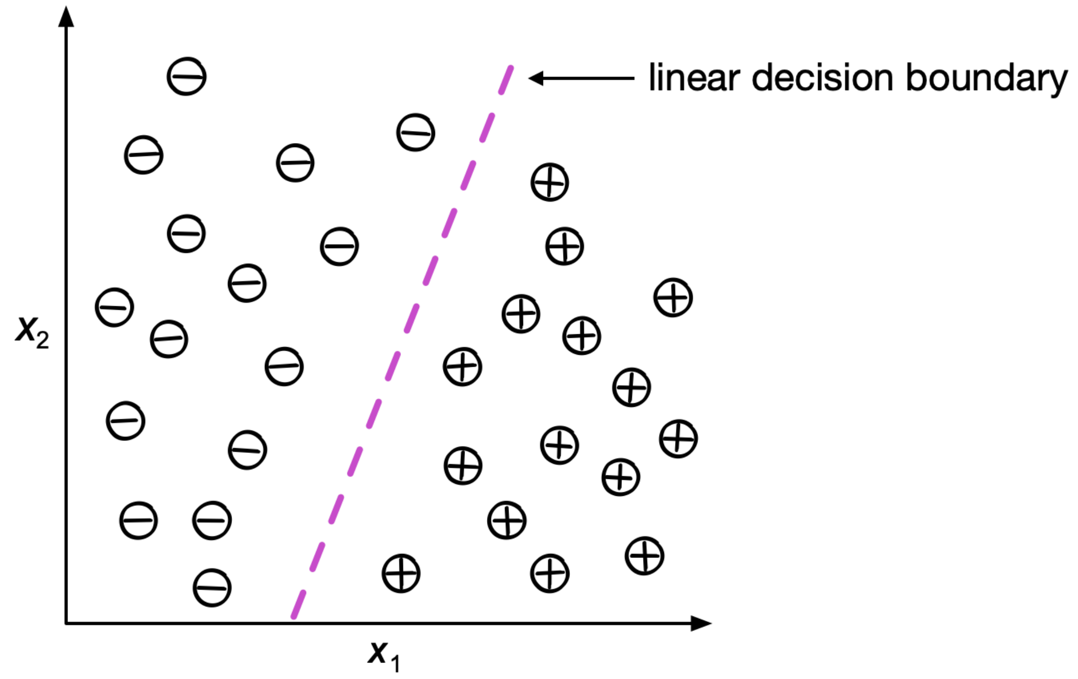


Source: Raschka and Mirjalili (2019). *Python Machine Learning, 3rd Edition*

Supervised Learning

Ex: Classification

What are the class labels (y's)?



Source: Raschka and Mirjalily (2019). Python Machine Learning, 3rd Edition

Unsupervised Learning

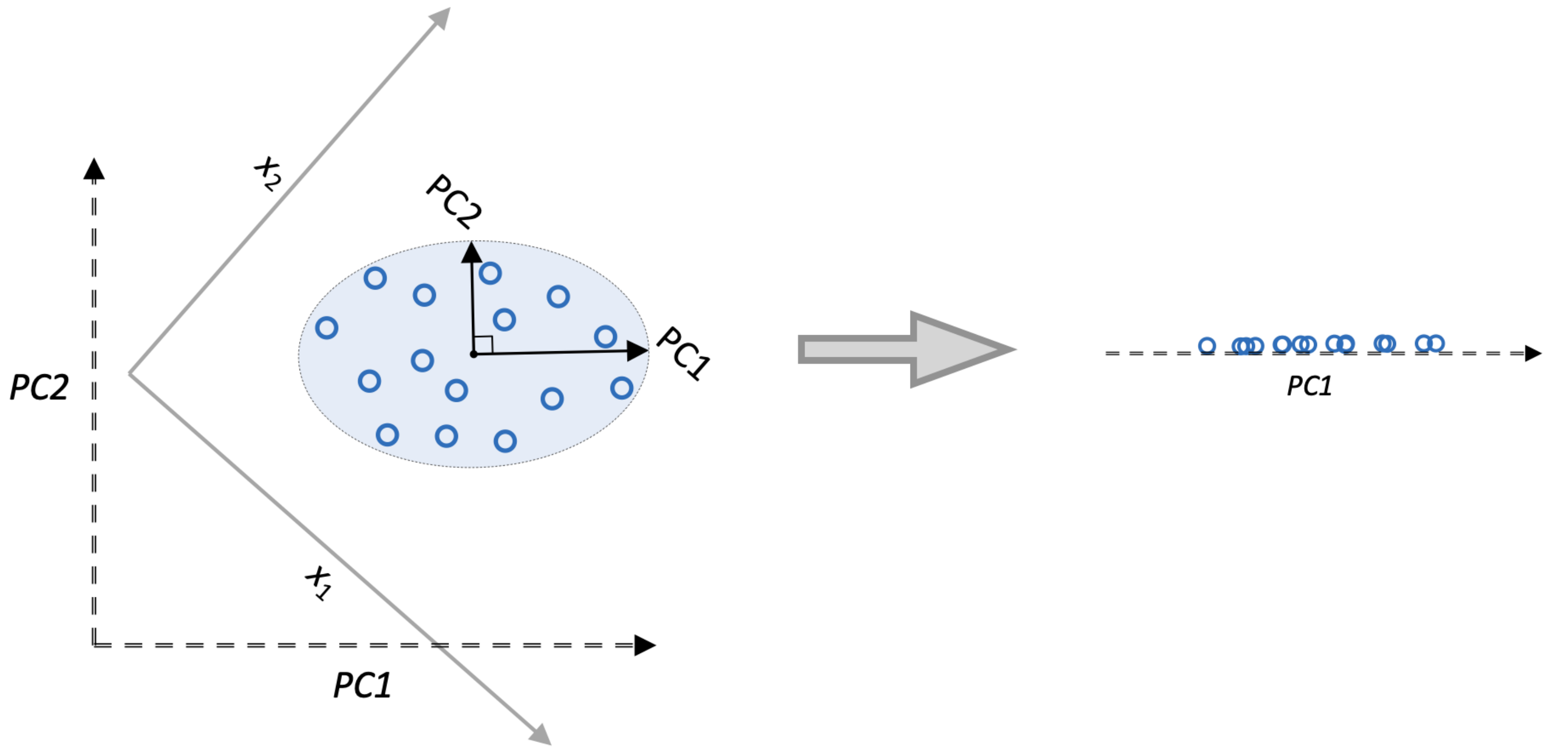
Unsupervised Learning

- No labels/targets
- No feedback
- Find hidden structure in data

Source: Raschka and Mirjalily (2019). Python Machine Learning, 3rd Edition

Unsupervised Learning

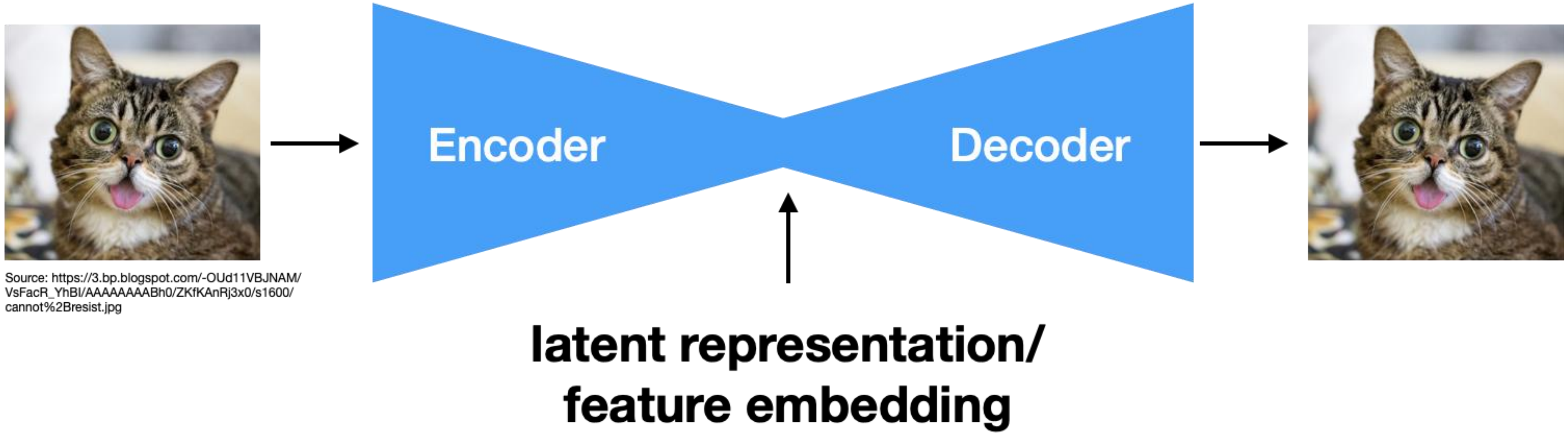
Ex: Representation Learning / Dimensionality Reduction with PCA



Source: Raschka and Mirjalily (2019). Python Machine Learning, 3rd Edition

Unsupervised Learning

Ex: Representation Learning / Dimensionality Reduction with Autoencoders



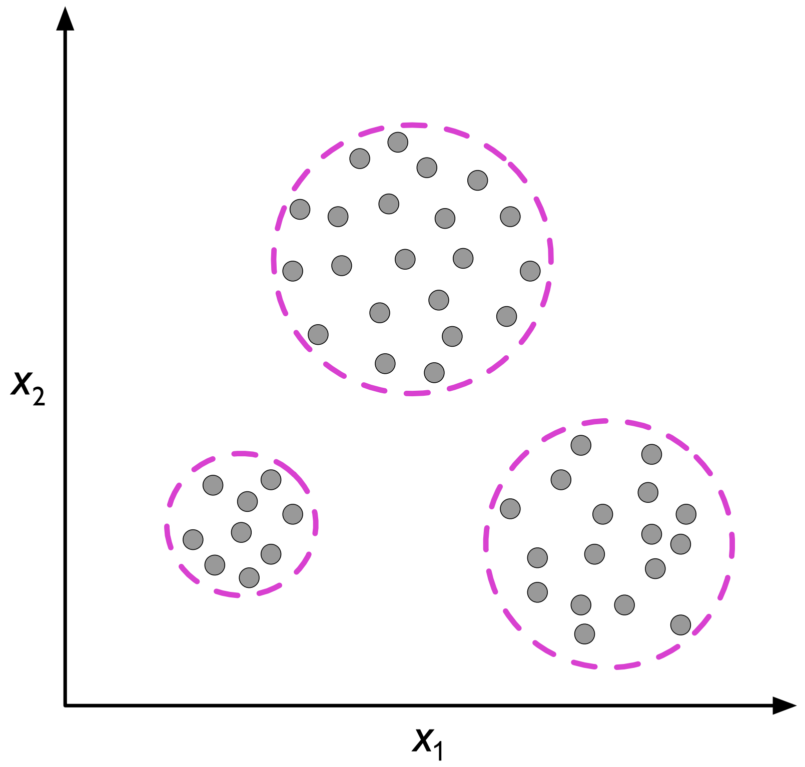
Source: https://3.bp.blogspot.com/-OUd11VBJNAM/VsFacR_YhBI/AAAAAAAABh0/ZKfKAnRj3x0/s1600/cannot%2Bresist.jpg

Fun fact: PCA = linear Autoencoder

Source: Raschka and Mirjalily (2019). Python Machine Learning, 3rd Edition

Unsupervised Learning

Ex: Clustering



Source: Raschka and Mirjalily (2019). Python Machine Learning, 3rd Edition

Reinforcement Learning

Reinforcement Learning

- Decision process
- Reward system
- Learn series of actions

Source: Raschka and Mirjalily (2019). *Python Machine Learning, 3rd Edition*

Reinforcement Learning

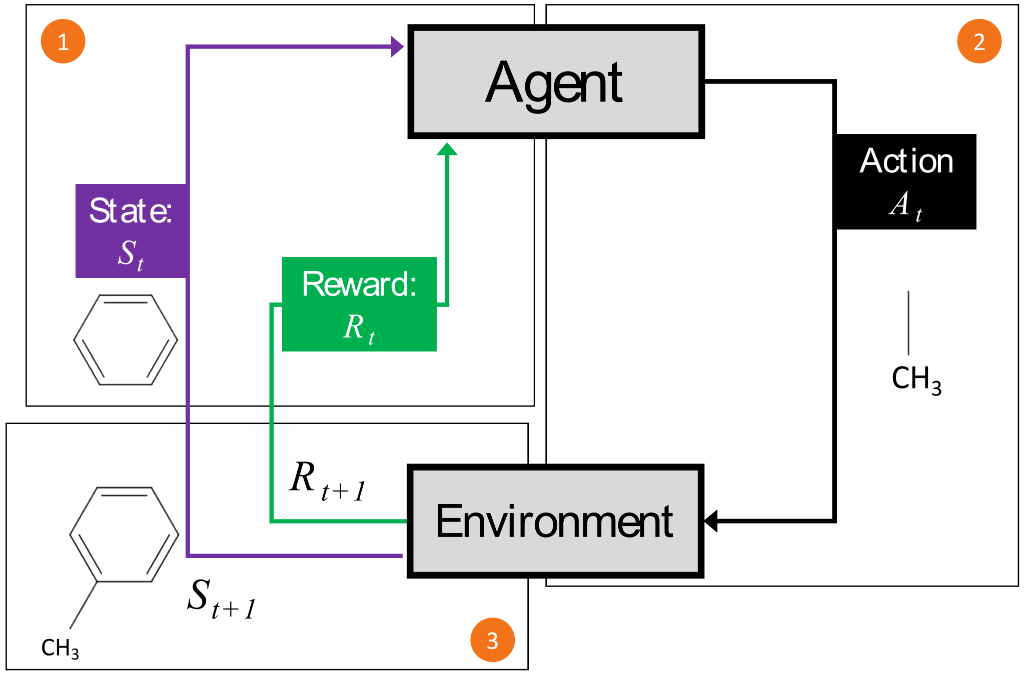
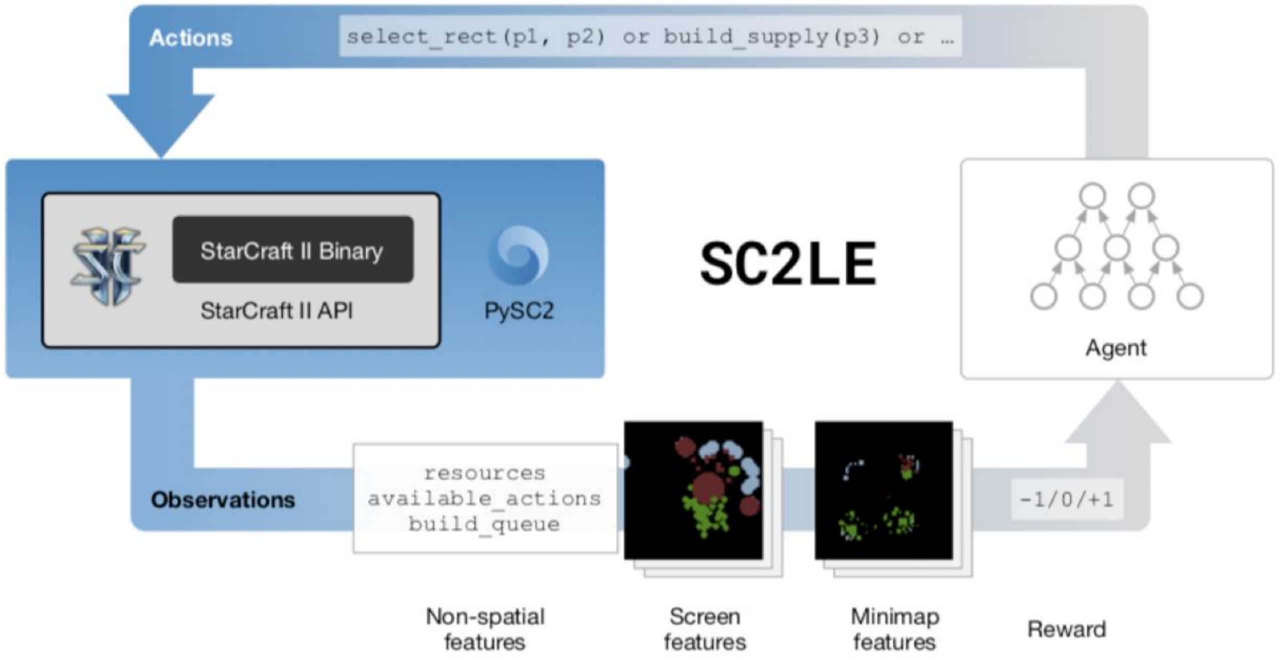


Figure 5: Representation of the basic reinforcement learning paradigm with a simple molecular example. (1) Given a benzene ring (state S_t at iteration t) and some reward value R_t at iteration t, (2) the agent selects an action A_t that adds a methyl group to the benzene ring. (3) The environment considers this information for producing the next state (S_{t+1}) and reward (R_{t+1}). This cycle repeats until the episode is terminated.

Source: Sebastian Raschka and Benjamin Kaufman (2020)
Machine learning and AI-based approaches for bioactive ligand discovery and GPCR-ligand recognition

Reinforcement Learning



Vinyals, Oriol, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani et al. "Starcraft II: A new challenge for reinforcement learning." *arXiv preprint arXiv:1708.04782* (2017).

The 3 Broad Categories of ML

Supervised Learning

- Labeled data
- Direct feedback
- Predict outcome/future

Unsupervised Learning

- No labels/targets
- No feedback
- Find hidden structure in data

Reinforcement Learning

- Decision process
- Reward system
- Learn series of actions

Source: Raschka and Mirjalily (2019). *Python Machine Learning, 3rd Edition*

Semi-Supervised Learning

- Mix between supervised and unsupervised learning
- Some training examples contain outputs, but some don't

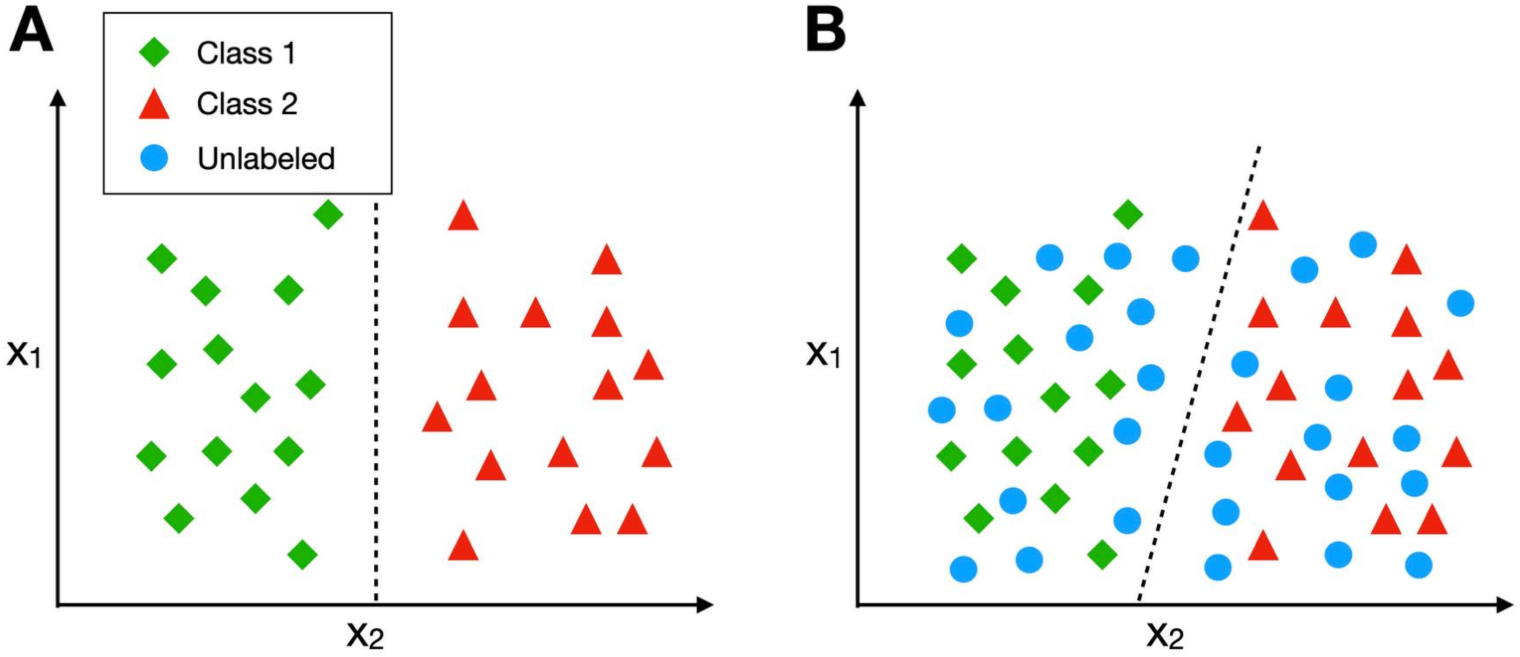
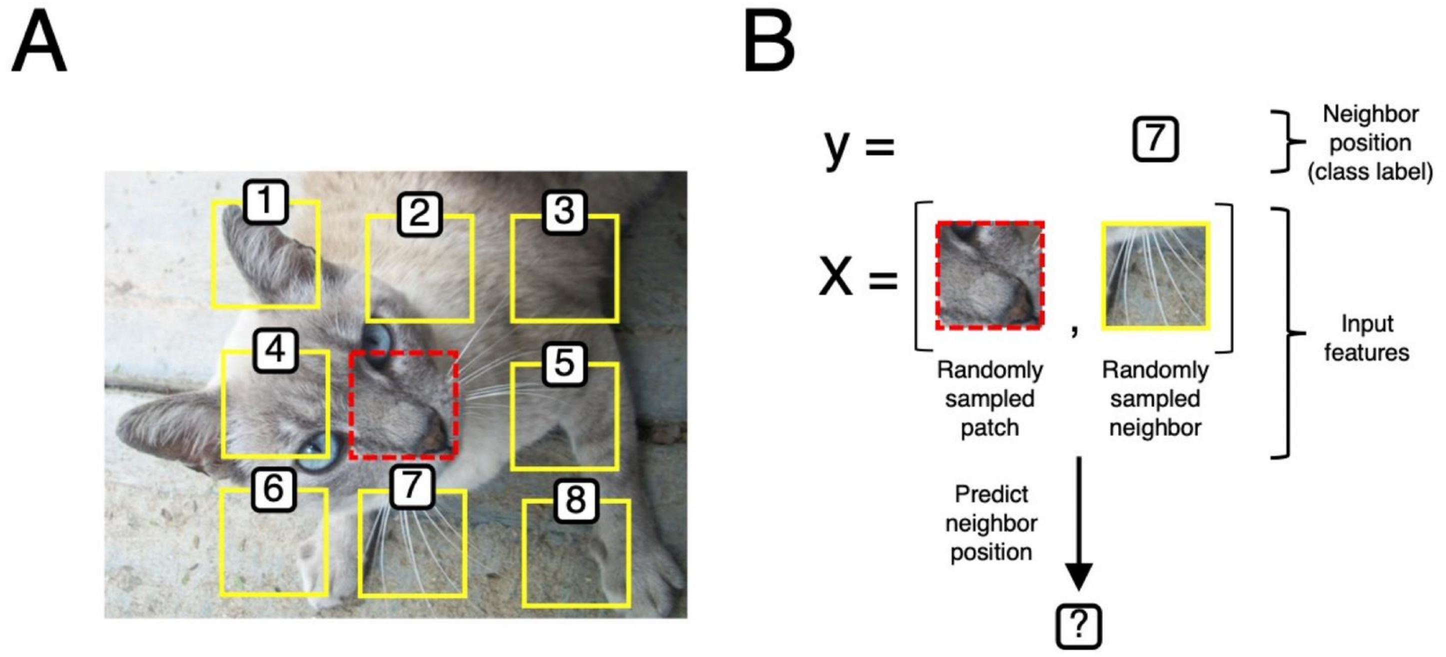


Illustration of semi-supervised learning incorporating unlabeled examples. (A) A decision boundary derived from the labeled training examples only. (B) A decision boundary based on both labeled and unlabeled examples.

Self-Supervised Learning

- A process of deriving and using label information directly from the data itself rather than having humans annotating it


A
B
1
2
3
4
5
6
7
8
y =
7
Neighbor position (class label)
X =
Randomly sampled patch
Randomly sampled neighbor
Input features
Predict neighbor position
?

Self-supervised learning via context prediction. (A) A random patch is sampled (red square) along with 9 neighboring patches. (B) Given the random patch and a random neighbor patch, the task is to predict the position of the neighboring patch relative to the center patch (red square).

3. Some Necessary Jargon

Machine Learning Jargon

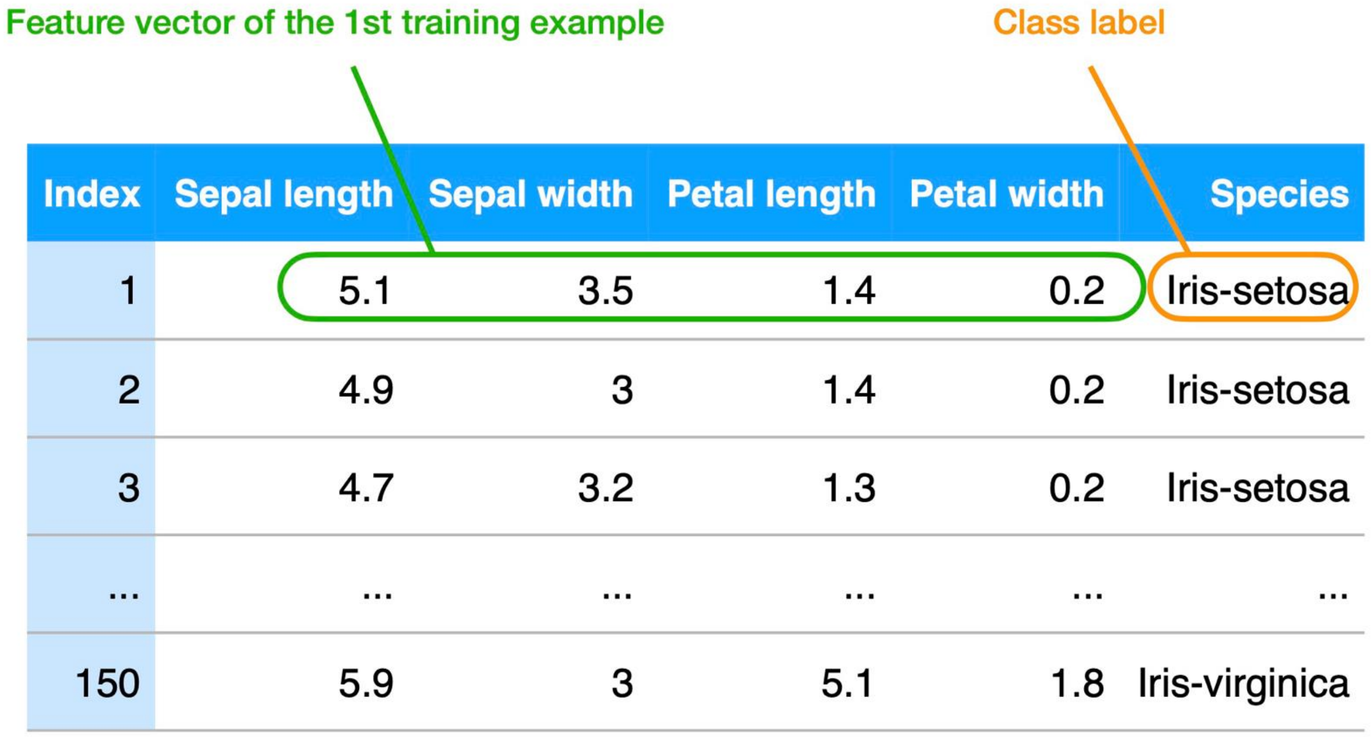
- **Training a model** = fitting a model = parameterizing a model = learning from data
- **Training example**, synonymous to training record, training instance, training sample (in some contexts, sample refers to a collection of training examples)
- **Feature**, synonymous to observation, predictor, variable, independent variable, input, attribute, covariate
- **Target**, synonymous to outcome, ground truth, output, response variable, dependent variable, (class) label (in classification)
- **Output / Prediction**, use this to distinguish from targets; here, means output from the model

Machine Learning Jargon

- **Supervised learning**
  - Learning function to map input x (features) to output y (targets)
- **Structured data**
  - Databases, spreadsheets/csv files, etc
- **Unstructured data**
  - Features like image pixels, audio signals, text sentences

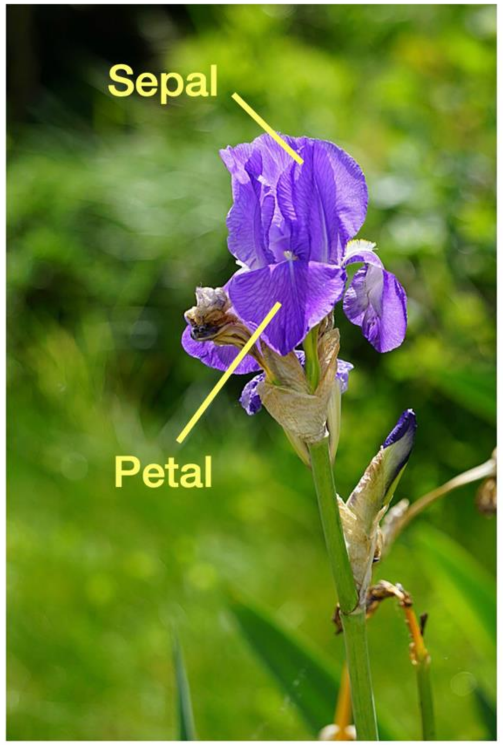
Structured vs Unstructured Data

A


Feature vector of the 1st training example
Class label

Index	Sepal length	Sepal width	Petal length	Petal width	Species
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
...	...	...	...	...	...
150	5.9	3	5.1	1.8	Iris-virginica

B



Data Representation

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

Feature vector

Data Representation

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

Feature vector

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix}$$

Feature Matrix / Design Matrix

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

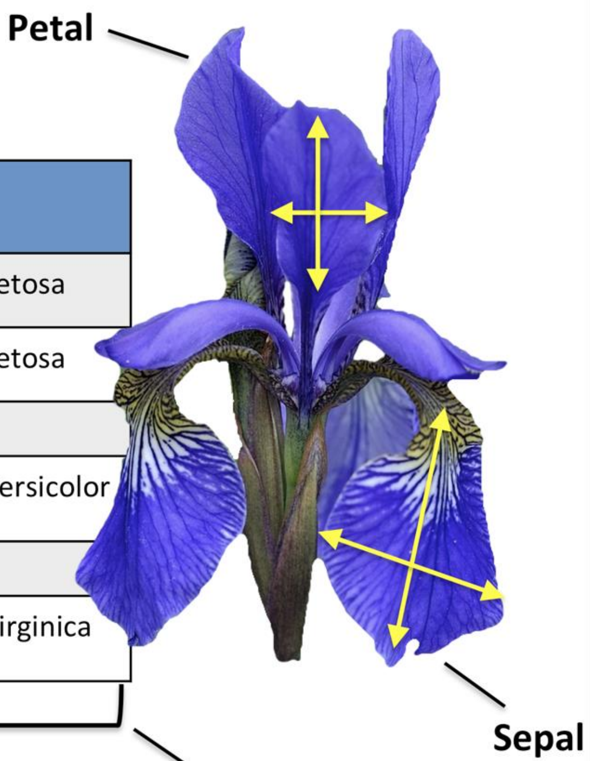
Feature Matrix / Design Matrix

Data Representation (structured data)

$m =$ ____

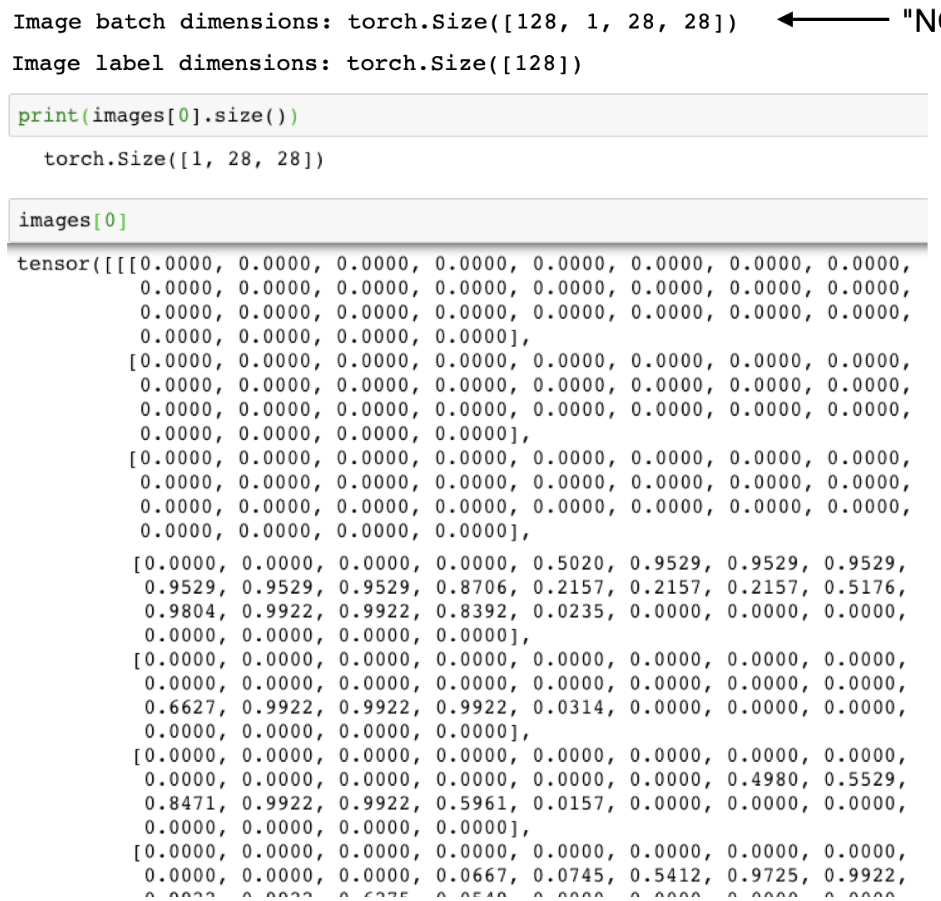
$n =$ ____

	Sepal length	Sepal width	Petal length	Petal width	
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			…		
50	6.4	3.5	4.5	1.2	Versicolor
			…		
150	5.9	3.0	5.0	1.8	Virginica



Data Representation (unstructured data; images)

Convolutional Neural Networks



```
Image batch dimensions: torch.Size([128, 1, 28, 28])
Image label dimensions: torch.Size([128])
```

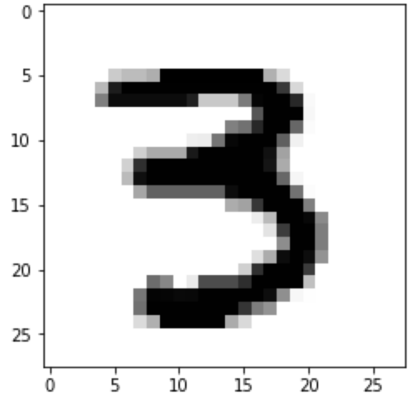
"NCHW" representation (more on that later)

```
print(images[0].size())
```

```
torch.Size([1, 28, 28])
```

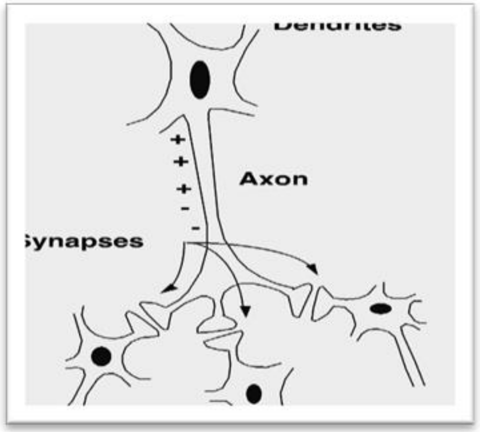
```
images[0]
```

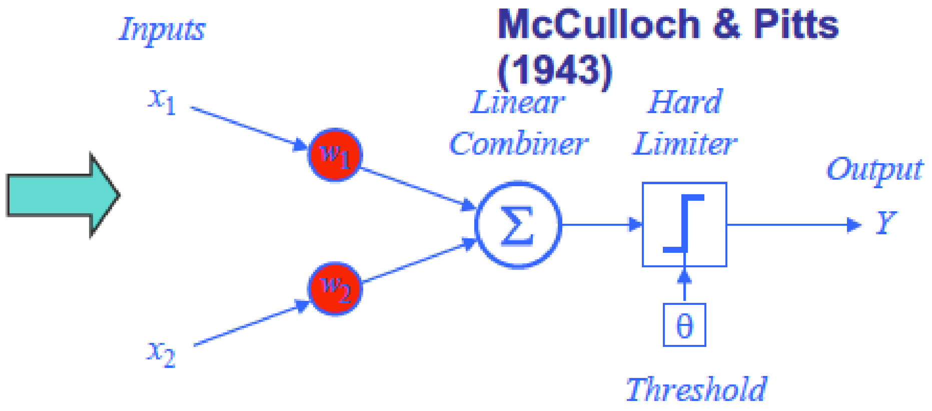
```
tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],

         [0.0000, 0.0000, 0.0000, 0.0000, 0.5020, 0.9529, 0.9529, 0.9529,
          0.9529, 0.9529, 0.9529, 0.8706, 0.2157, 0.2157, 0.2157, 0.5176,
          0.9804, 0.9922, 0.9922, 0.8392, 0.0235, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.6627, 0.9922, 0.9922, 0.9922, 0.0314, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.4980, 0.5529,
          0.8471, 0.9922, 0.9922, 0.5961, 0.0157, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0667, 0.0745, 0.5412, 0.9725, 0.9922,
```

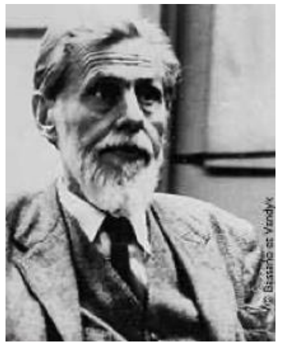


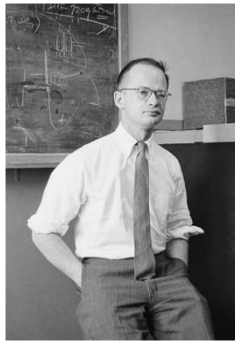
4. The building blocks of Deep Learning

McCulloch & Pitt’s neuron model (1943)



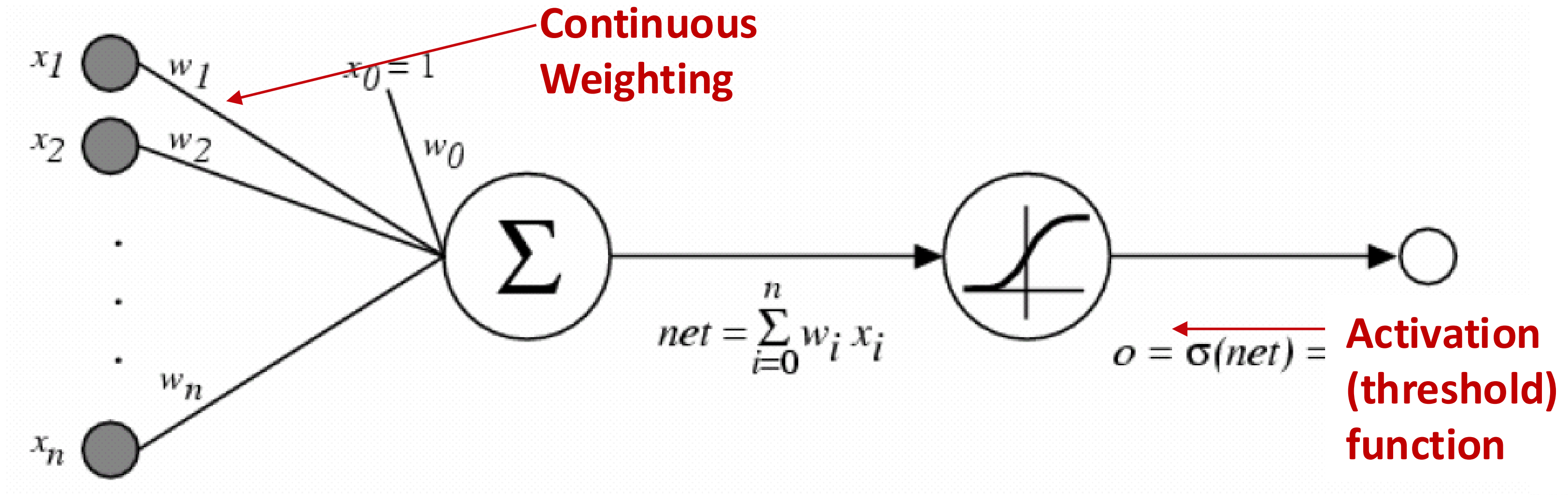



Warren McCulloch


Walter Pitts

Rosenblatt's Perceptron

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

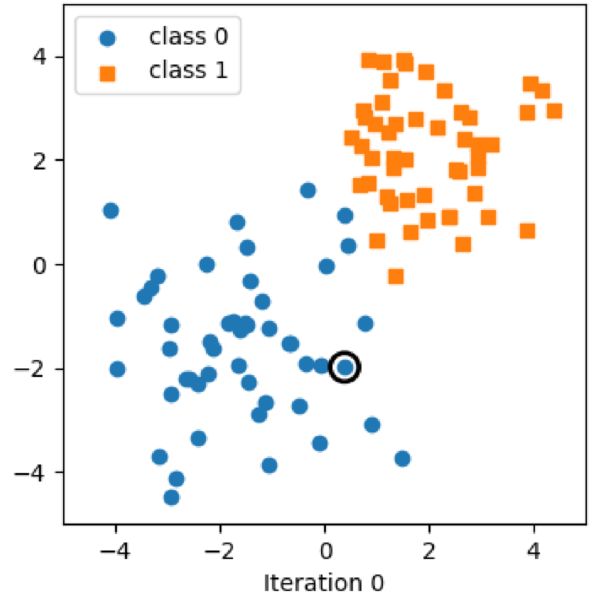


Generalizes MP neurons a bit…

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

Perceptron Learning Algorithm

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



[animated GIF]

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

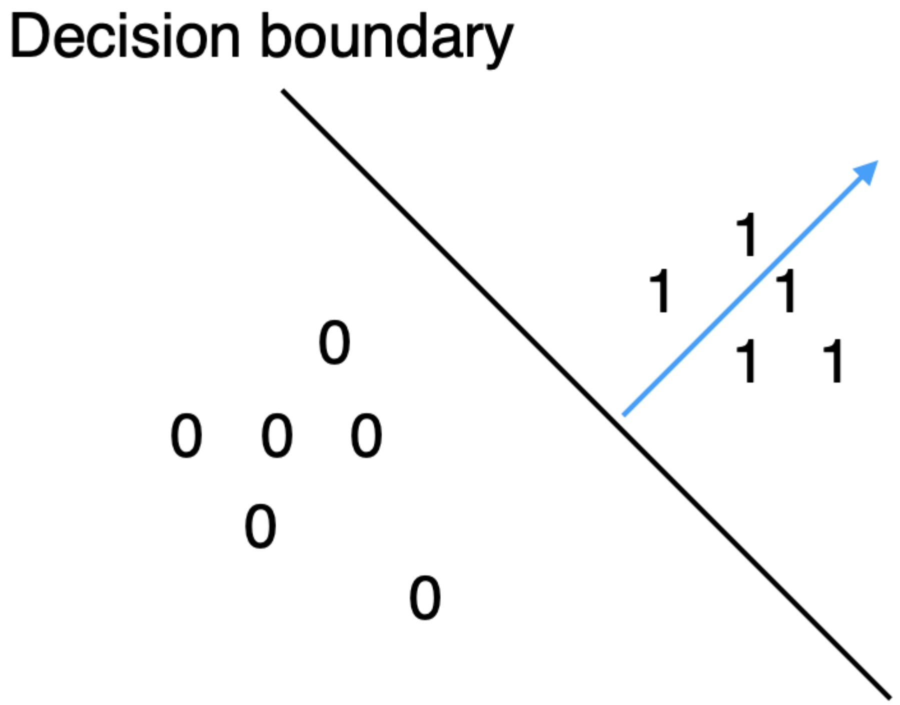
Perceptron Learning Algorithm (pseudocode)

Let

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

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

Perceptron Geometric Intuition



Weight vector is perpendicular to the boundary. Why?

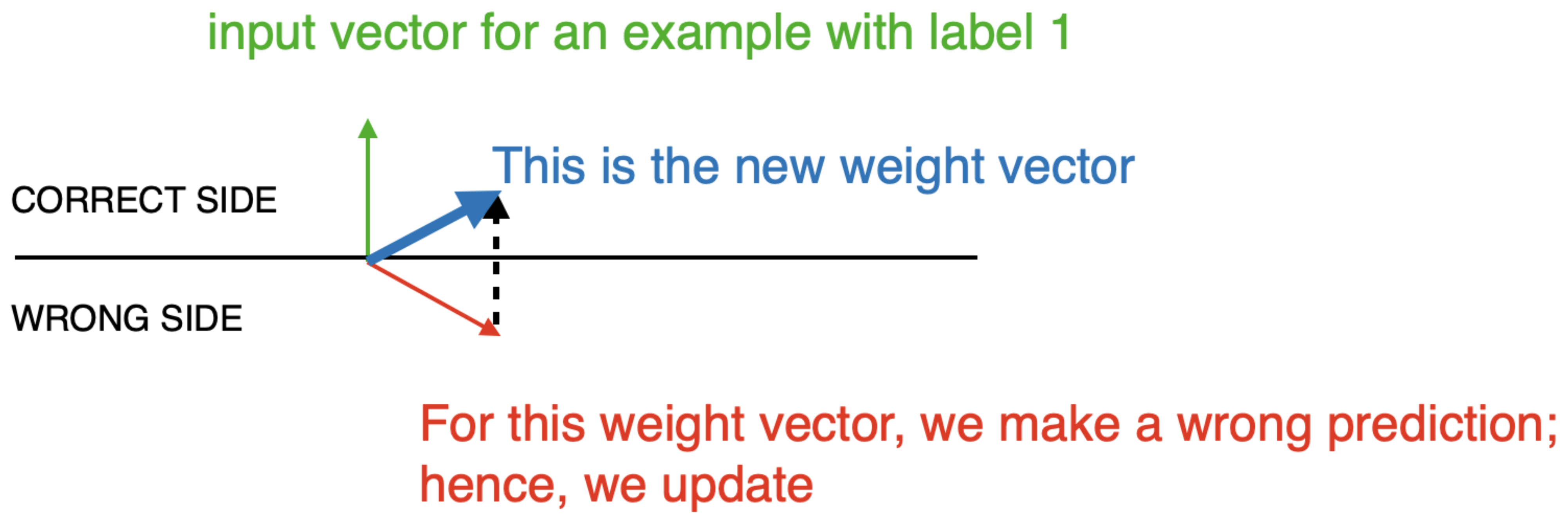
Remember,

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

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

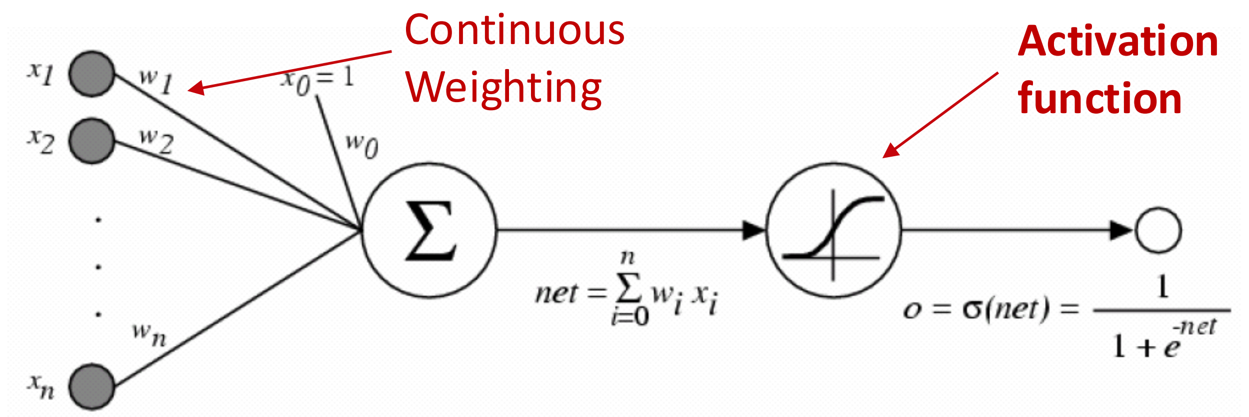
So this needs to be 0 at the boundary, and it is zero at 90°

Perceptron Geometric Intuition: Learning


input vector for an example with label 1
This is the new weight vector
CORRECT SIDE
WRONG SIDE
For this weight vector, we make a wrong prediction; hence, we update

Beyond Rosenblatt's Perceptron

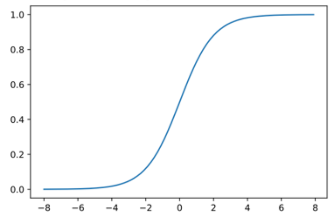
Perceptrons generalize MP Neurons



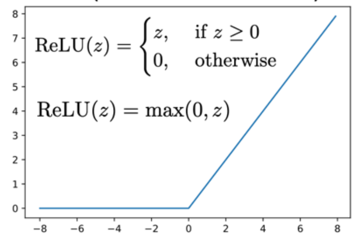
$$\begin{tikzcd} & \text{threshold function} \arrow[r] & \text{Classic Rosenblatt Perceptron} \\ \text{Activation function} \arrow[ur] \arrow[dr] & & \\ & \text{sigmoid} \arrow[r] & \text{DL "Perceptron" / sigmoid unit} \end{tikzcd}$$

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

(Logistic) Sigmoid

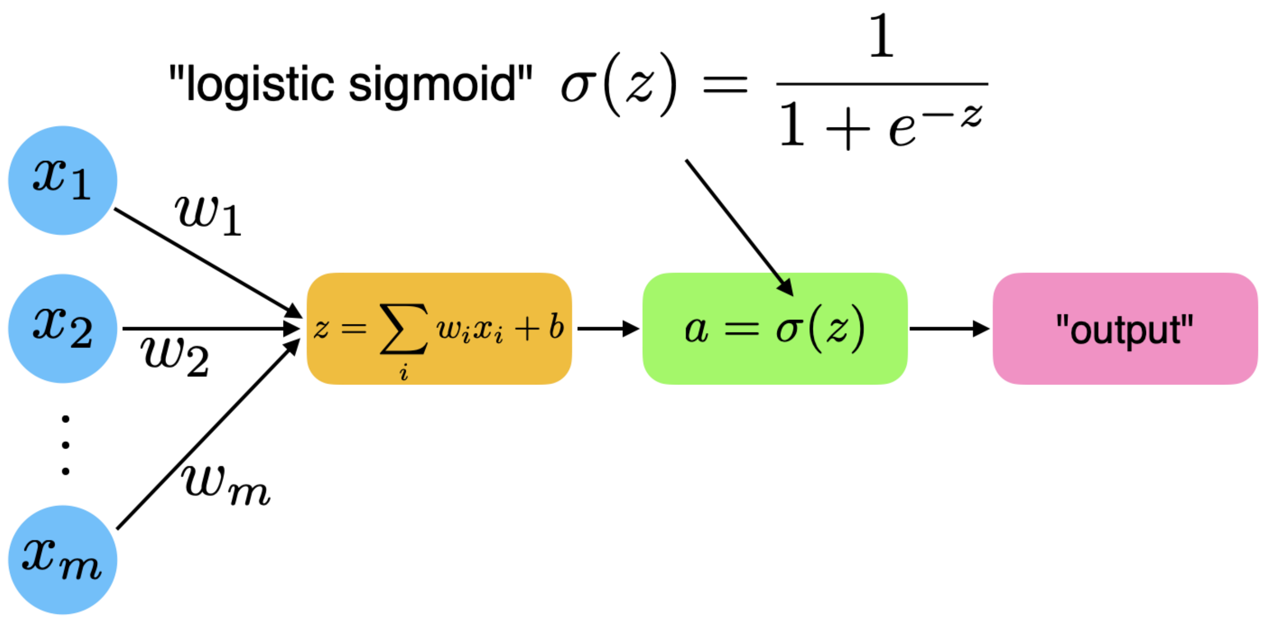


ReLU (Rectified Linear Unit)



Sigmoid unit: Logistic regression gives an optimizer

- For binary classes $y \in \{0, 1\}$



Sigmoid unit: Logistic regression gives an optimizer

- Given the output:

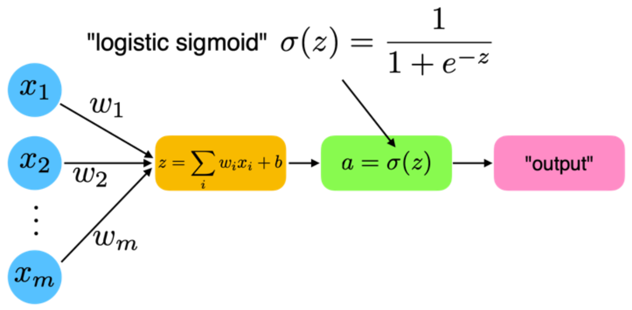
$$h(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

- We compute the probability as

$$P(y|\mathbf{x}) = \begin{cases} h(\mathbf{x}) & \text{if } y = 1 \\ 1 - h(\mathbf{x}) & \text{if } y = 0 \end{cases}$$

$$\downarrow$$

$$P(y|\mathbf{x}) = a^y (1-a)^{(1-y)}$$



Recall Bernoulli distribution…

Sigmoid unit: Logistic regression gives an optimizer

- Given the probability:

$$P(y|\mathbf{x}) = a^y(1-a)^{(1-y)}$$

- Under MLE estimation, we would like to maximize the multi-sample likelihood:

$$P(y^{[i]}, \ldots, y^{[n]}|\mathbf{x}^{[1]}, \ldots, \mathbf{x}^{[n]}) = \prod_{i=1}^{n} P(y^{[i]}|\mathbf{x}^{[i]})$$

$$= \underbrace{\prod_{i=1}^{n} \left(\sigma(z^{(i)})\right)^{y^{(i)}} \left(1-\sigma(z^{(i)})\right)^{1-y^{(i)}}}_{\text{Likelihood}}$$

Sigmoid unit: Logistic regression gives an optimizer

$$P\left(y^{[i]}, \ldots, y^{[n]} \middle| \mathbf{x}^{[1]}, \ldots, \mathbf{x}^{[n]}\right) = \underbrace{\prod_{i=1}^{n} \left(\sigma\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \sigma\left(z^{(i)}\right)\right)^{1-y^{(i)}}}_{\text{Likelihood}}$$

- We are going to optimize via gradient descent, so let’s apply the logarithm to separate components:

$$\begin{aligned} l(\mathbf{w}) &= \log L(\mathbf{w}) \\ &= \underbrace{\sum_{i=1}^{n} \left[y^{(i)} \log\left(\sigma\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \sigma\left(z^{(i)}\right)\right)\right]}_{\text{Log-Likelihood}} \end{aligned}$$

Sigmoid unit: Logistic regression gives an optimizer

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{\partial \mathcal{L}}{\partial a} \frac{da}{dz} \frac{\partial z}{\partial w_j}$$

$$\begin{tikzcd}
\frac{\partial \mathcal{L}}{\partial a} = \frac{a-y}{a-a^2} \arrow[dr] & & \\
\frac{da}{dz} = \frac{e^{-z}}{(1+e^{-z})^2} = a \cdot (1-a) \arrow[r] & \frac{\partial \mathcal{L}}{\partial z} = a - y \arrow[r] & \frac{\partial \mathcal{L}}{\partial w_j} = (a-y)x_j \\
\frac{\partial z}{w_j} = x_j \arrow[urr] & &
\end{tikzcd}$$

Logistic Regression: Gradient Descent learning Rule

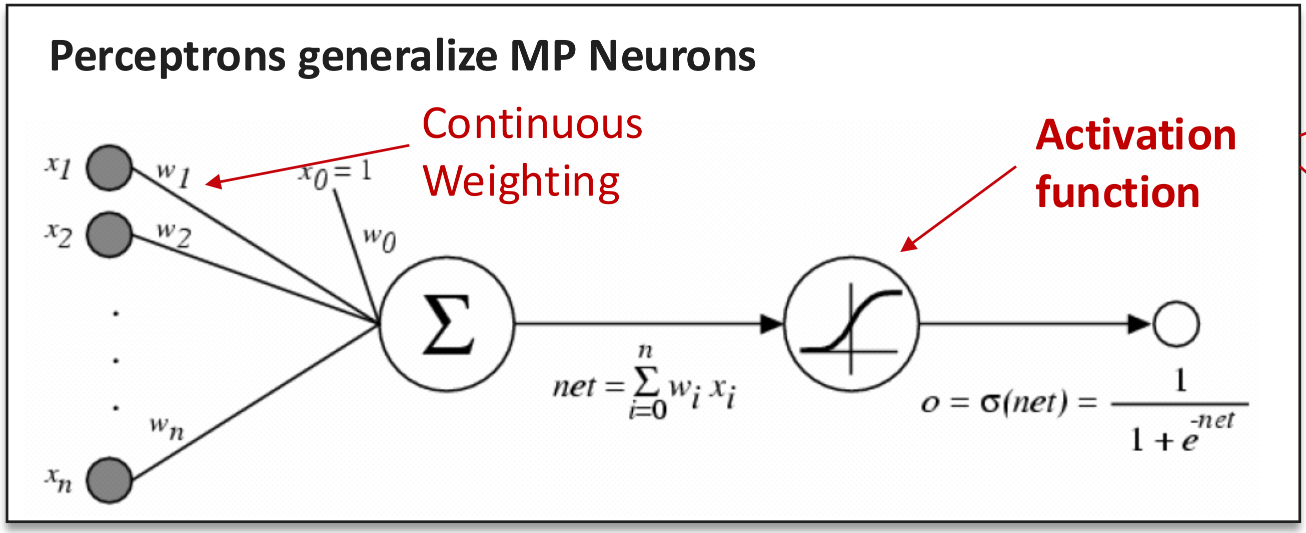
Stochastic gradient descent:

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$, $\mathbf{b} := 0$
2. For every training epoch:
    A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
        (a) $\hat{y}^{[i]} := \sigma\big(\mathbf{x}^{[i]T}\mathbf{w} + b\big)$
        (b) $\nabla_{\mathbf{w}}\mathcal{L} = -\big(y^{[i]} - \hat{y}^{[i]}\big)\mathbf{x}^{[i]}$
            $\nabla_{b}\mathcal{L} = -\big(y^{[i]} - \hat{y}^{[i]}\big)$
        (c) $\mathbf{w} := \mathbf{w} + \eta \times (-\nabla_{\mathbf{w}}\mathcal{L})$
            $b := b + \underset{\text{learning rate}}{\eta} \times \underbrace{(-\nabla_{b}\mathcal{L})}_{\text{negative gradient}}$
```

Note

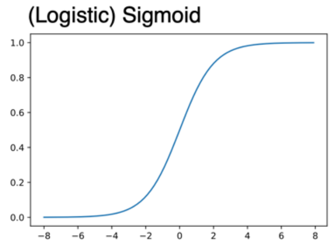
$$a - y \Leftrightarrow -\big(y^{[i]} - \hat{y}^{[i]}\big)$$

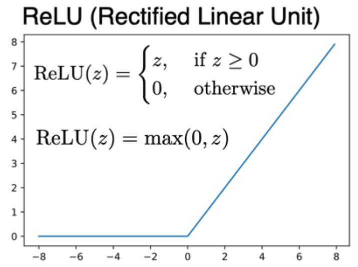
Beyond Rosenblatt's Perceptron


Perceptrons generalize MP Neurons
Continuous Weighting
Activation function
x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

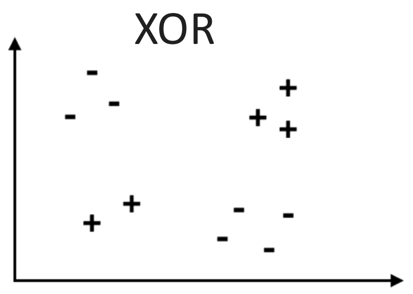
$$\begin{tikzcd} & \text{threshold function} \arrow[r] & \text{Classic Rosenblatt Perceptron} \\ \text{Activation function} \arrow[ur] \arrow[dr] & & \\ & \text{sigmoid} \arrow[r] & \text{DL "Perceptron" / sigmoid unit} \arrow[d] \\ & & \text{Gradient Descent} \end{tikzcd}$$

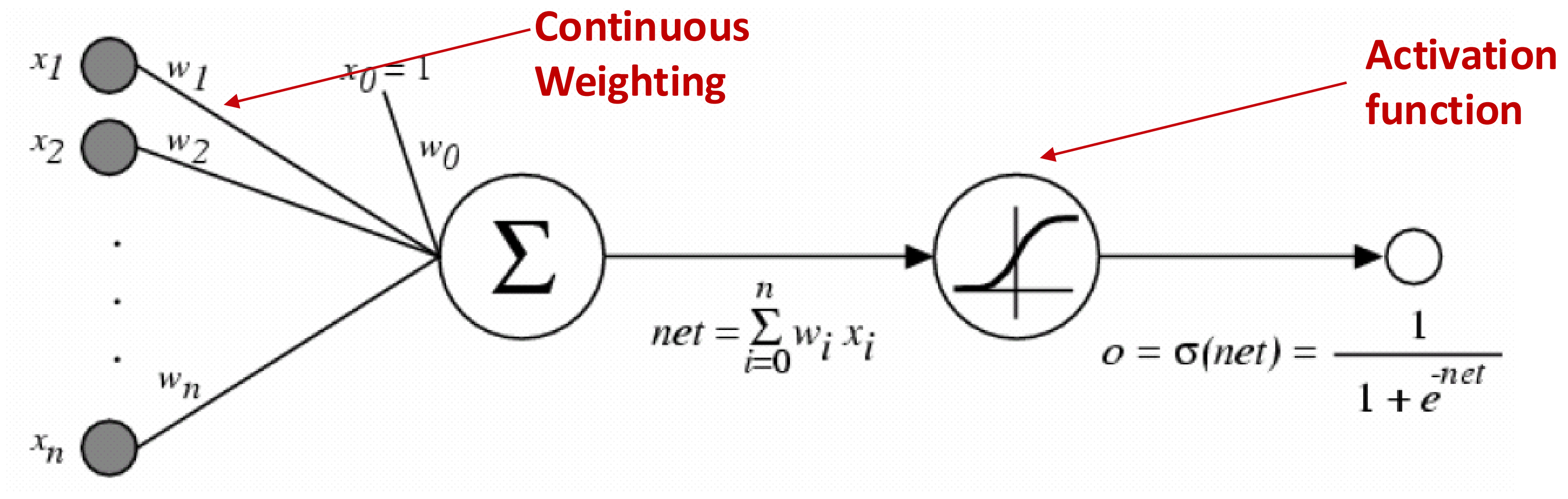
- Many activation functions:
  - Threshold function (perceptron, 1950+)
  - Sigmoid function (before 2000)
  - ReLU function (popular since CNNs)
  - Many variants of ReLU, e.g. leaky ReLU, GeLU





Can a Perceptron represent XOR?

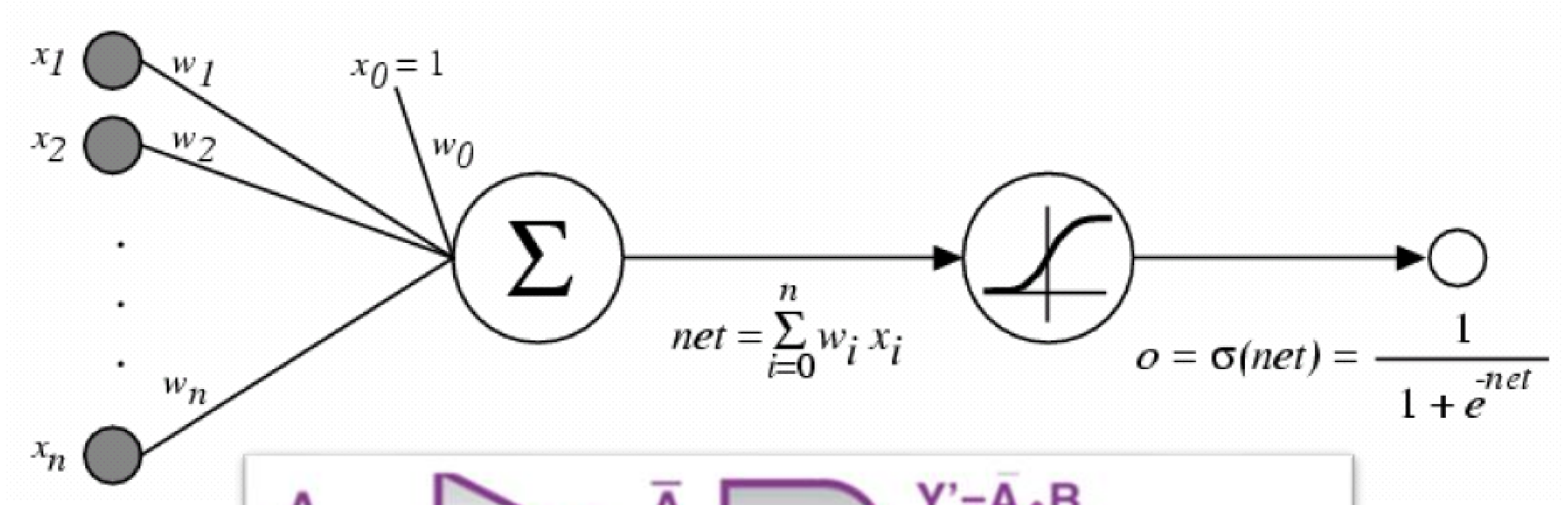




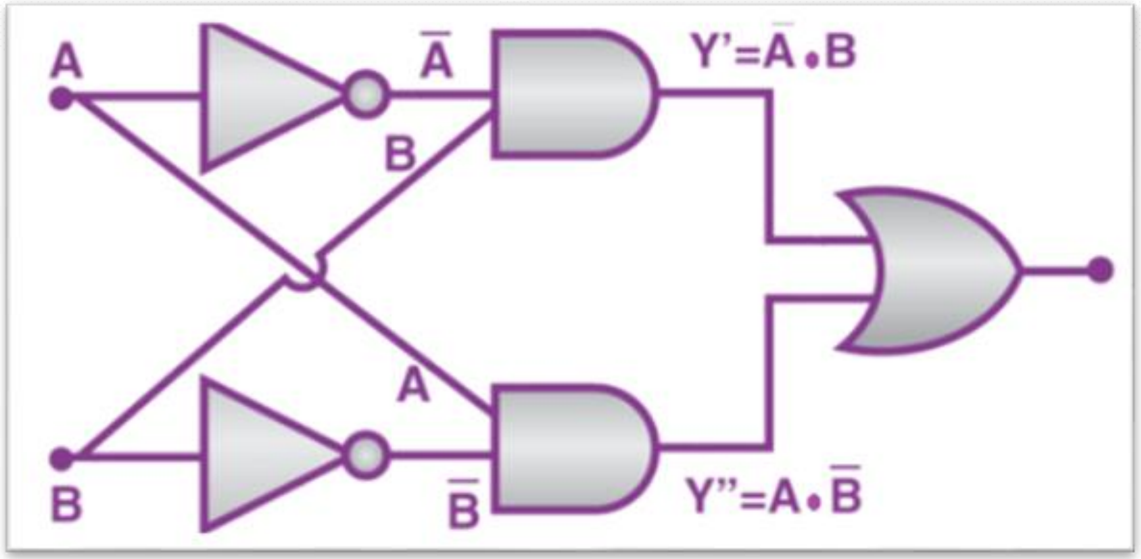
- No
- If there were, then there would be constants w_1 and w_2 such that:
  - When $x_1 = x_2$, then $\sigma(w_1 x_1 + w_2 x_2) < \theta$
  - When $x_1 \neq x_2$, then $\sigma(w_1 x_1 + w_2 x_2) \geq \theta$
  - Let $x_1 = 1, x_2 = 0$
    - $\text{Eq. (1): } \sigma(\text{w}_1) \geq \theta$
  - Let $x_1 = 0, x_2 = 1$
    - Eq. (2): $\sigma(\text{w}_2) \geq \theta$
  - Let $x_1 = 1, x_2 = 1$:
    - Eq. (3): $\sigma(w_1 + w_2) < \theta$

Eq. (1) + Eq. (2) contradicts Eq. (3)

An XOR Logic Gate



x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1+e^{-net}}$

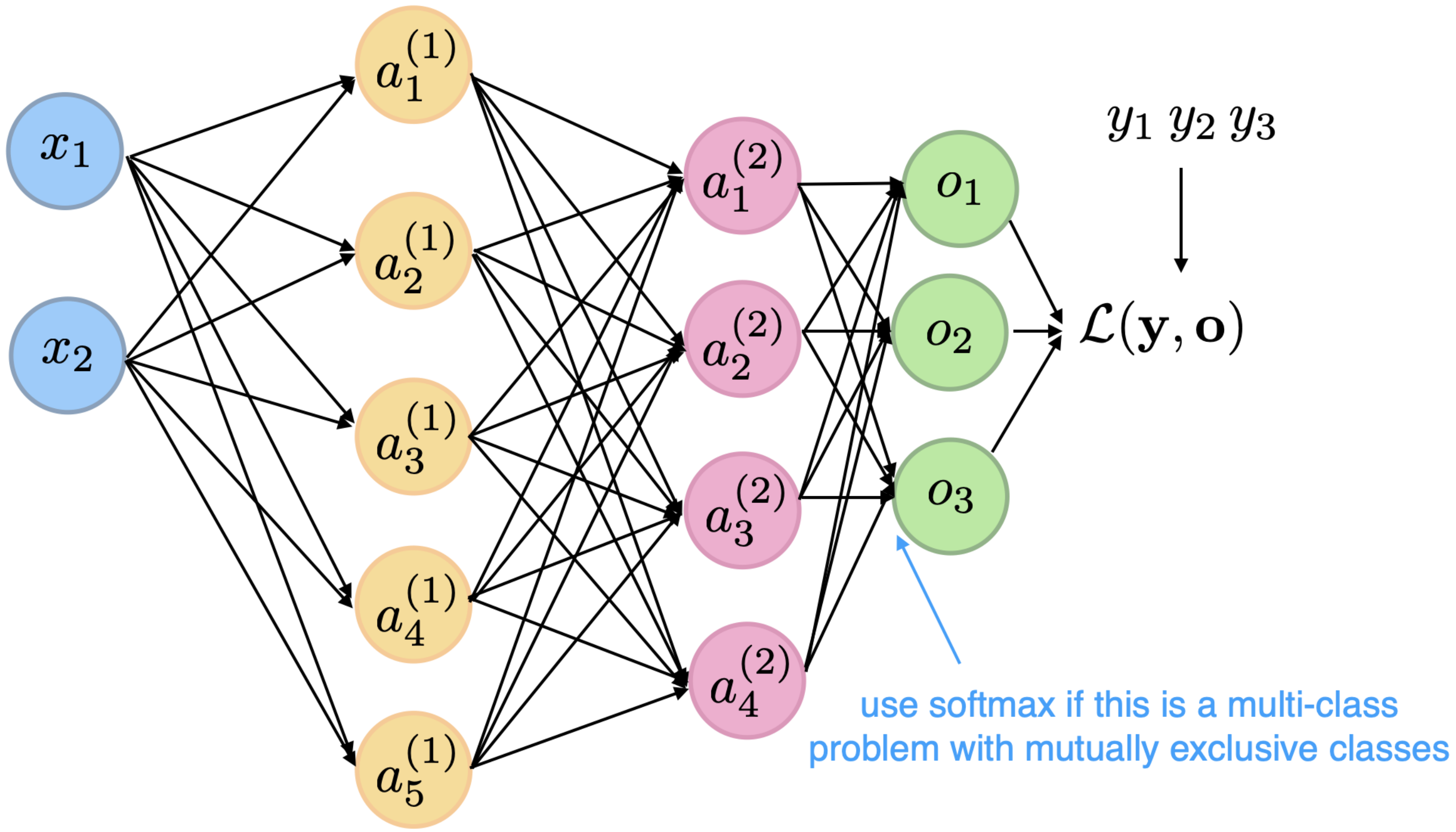


https://byjus.com/jee/basic-logic-gates/

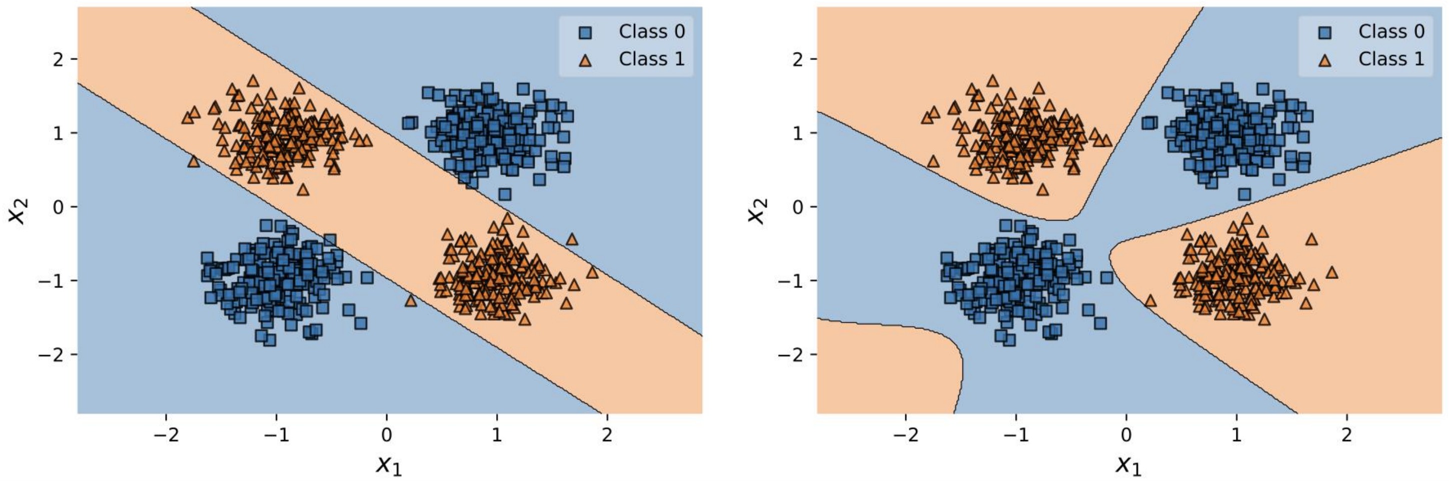
Multi-layer Perceptron?

Multilayer Perceptron

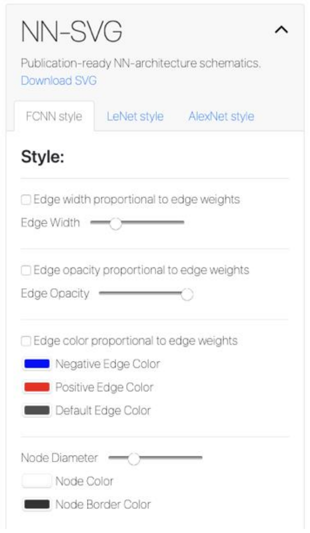
- Computation Graph with Multiple Fully-Connected Layers



Multilayer Perceptrons Can Solve XOR



Decision boundaries of two different multilayer perceptrons on simulated data solving the XOR problem



NN-SVG

Publication-ready NN-architecture schematics.

Download SVG

FCNN style | LeNet style | AlexNet style

Style:

☐ Edge width proportional to edge weights

Edge Width

☐ Edge opacity proportional to edge weights

Edge Opacity

☐ Edge color proportional to edge weights

🟦 Negative Edge Color

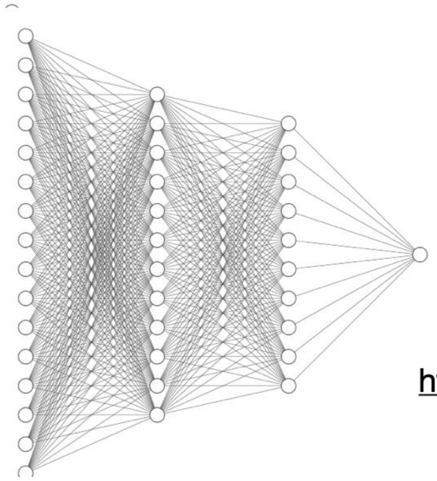
🟥 Positive Edge Color

⬛ Default Edge Color

Node Diameter

⬜ Node Color

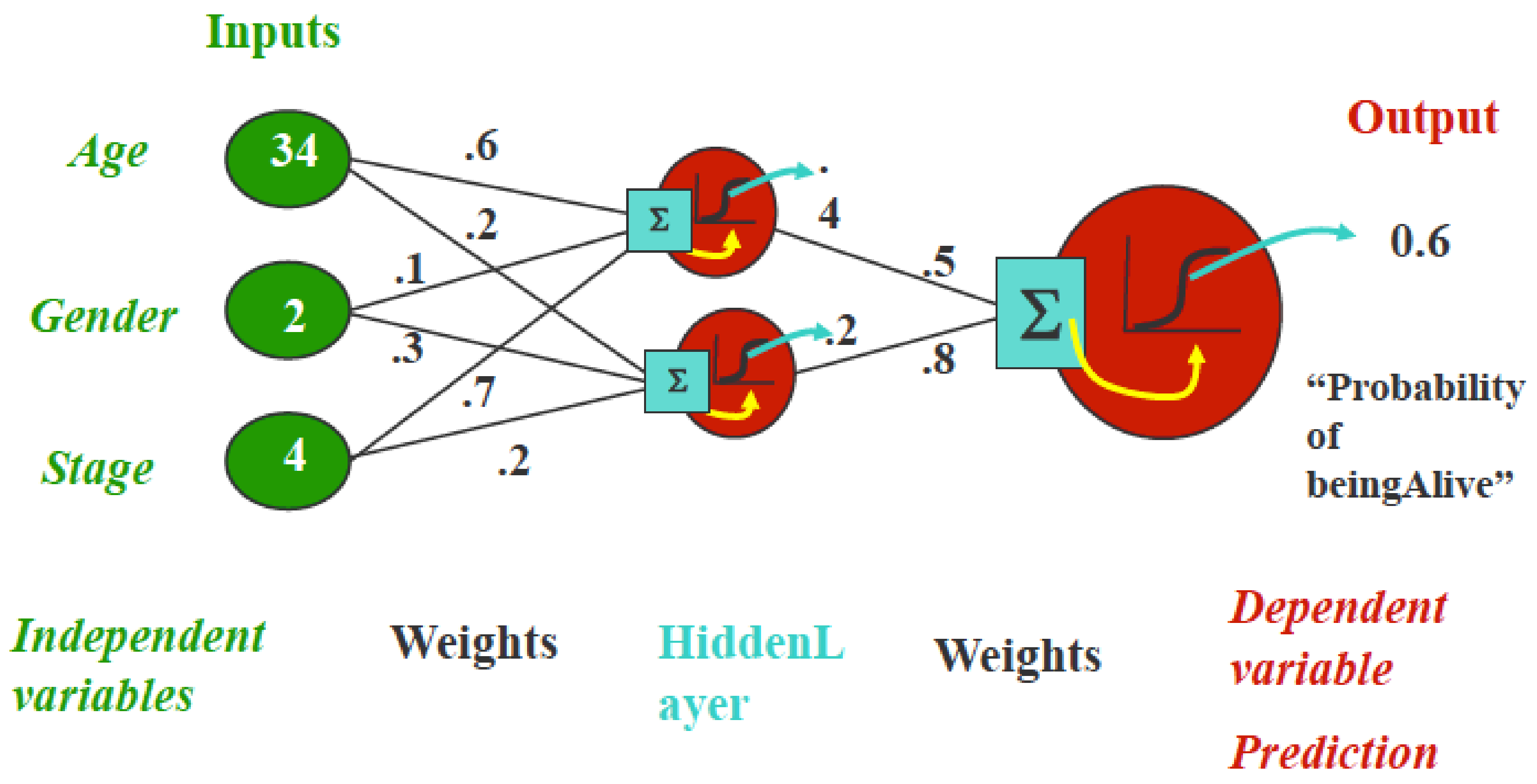
⬛ Node Border Color



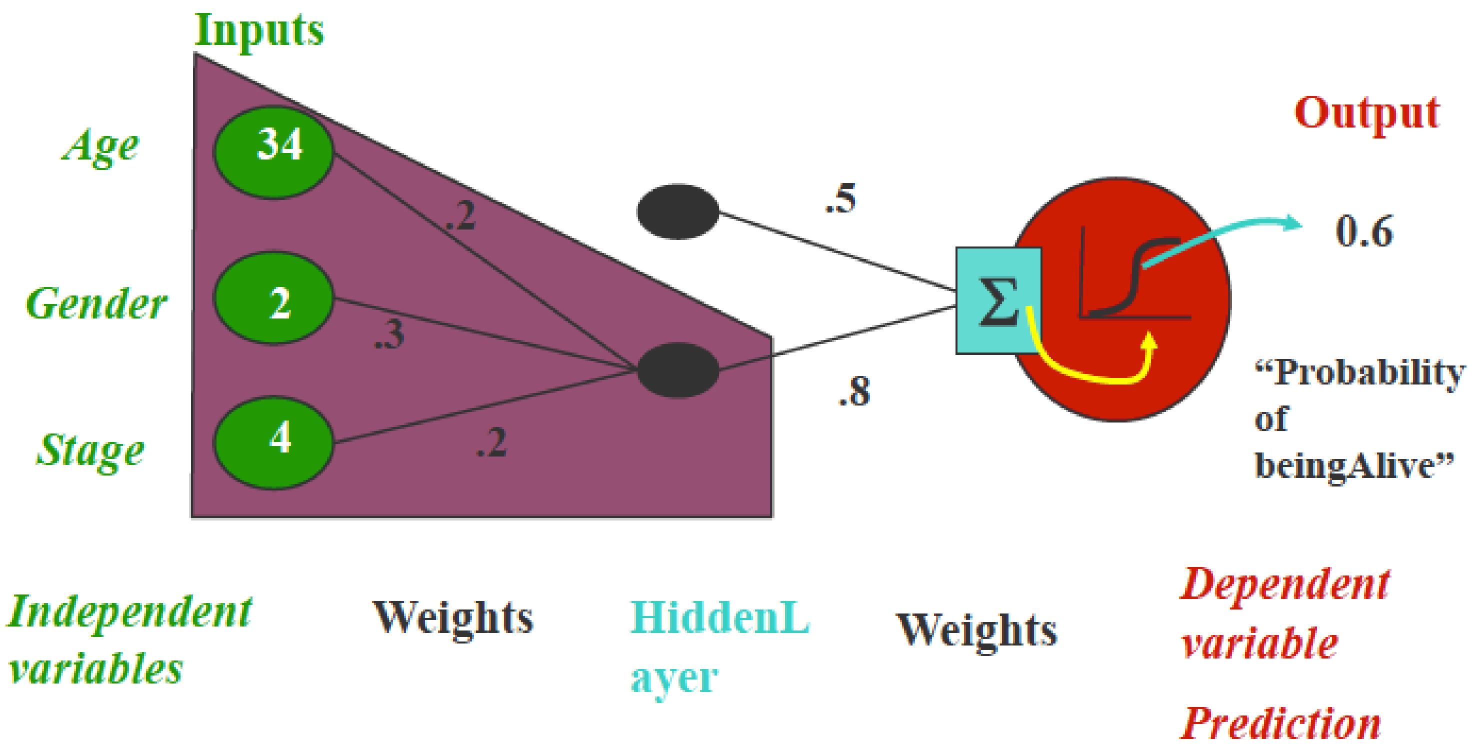
https://alexlenail.me/NN-SVG/index.html

Multi-Layer Perceptrons (MLPs)

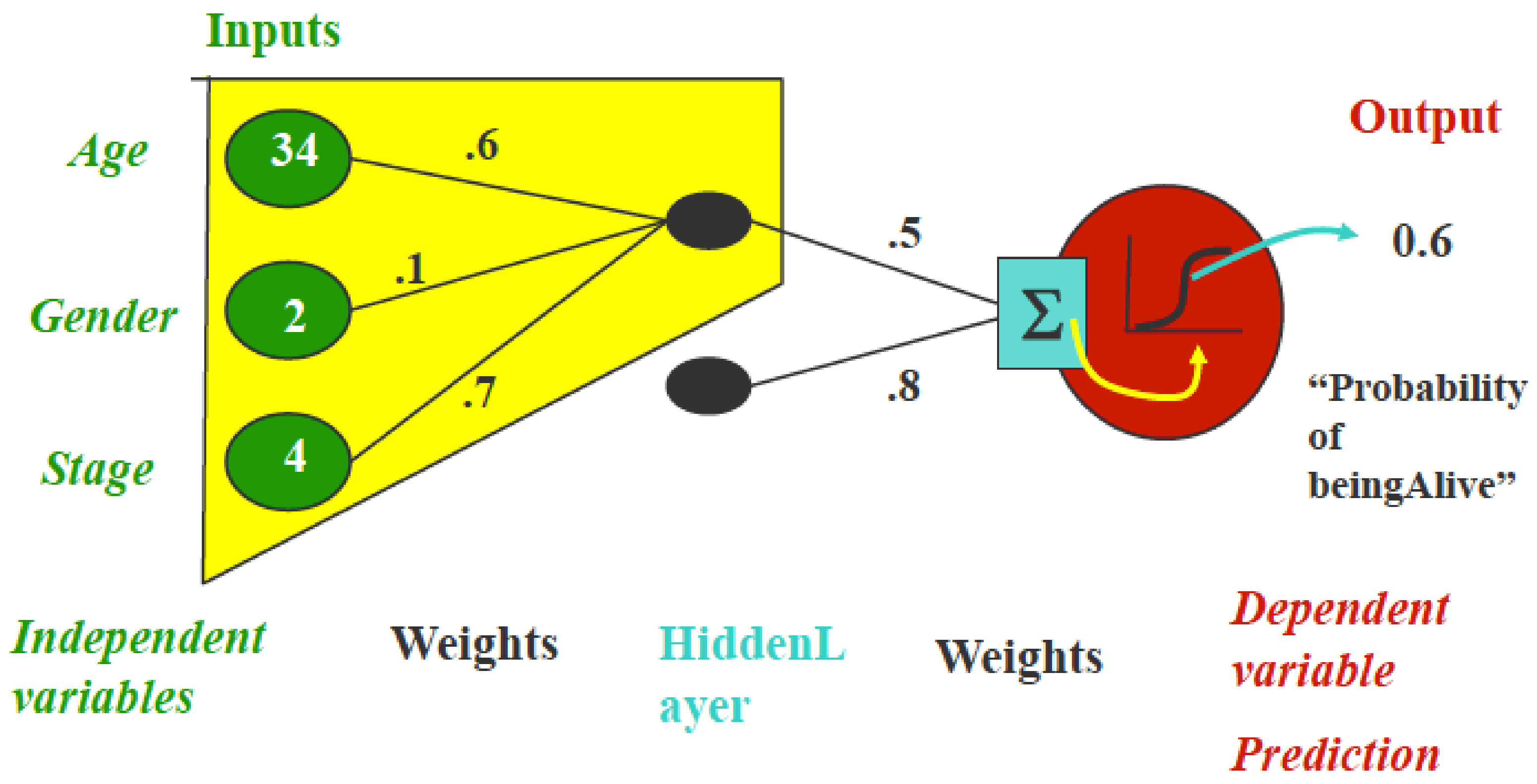
aka Multi-layer Neural Networks



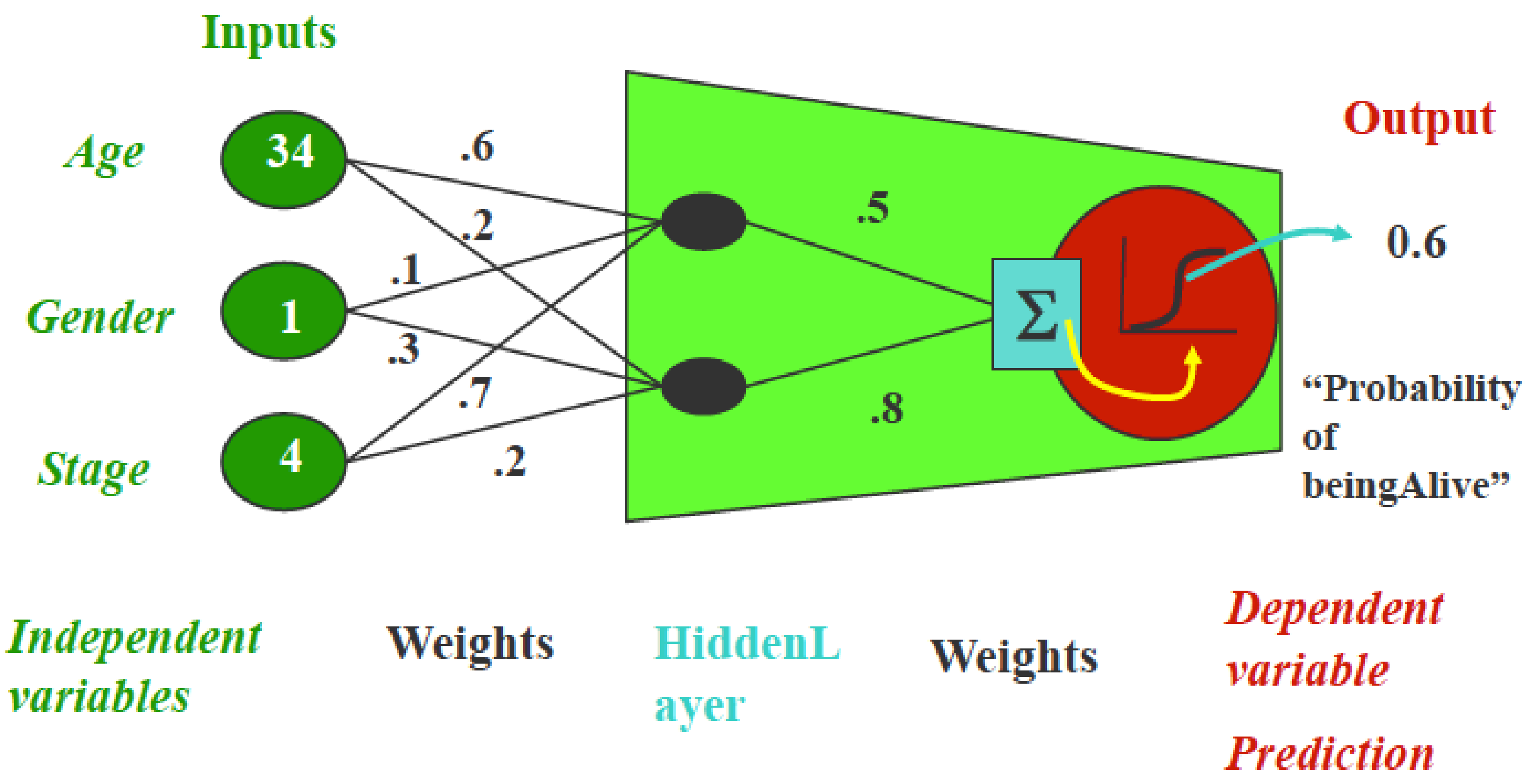
“Combined Logistic Models”



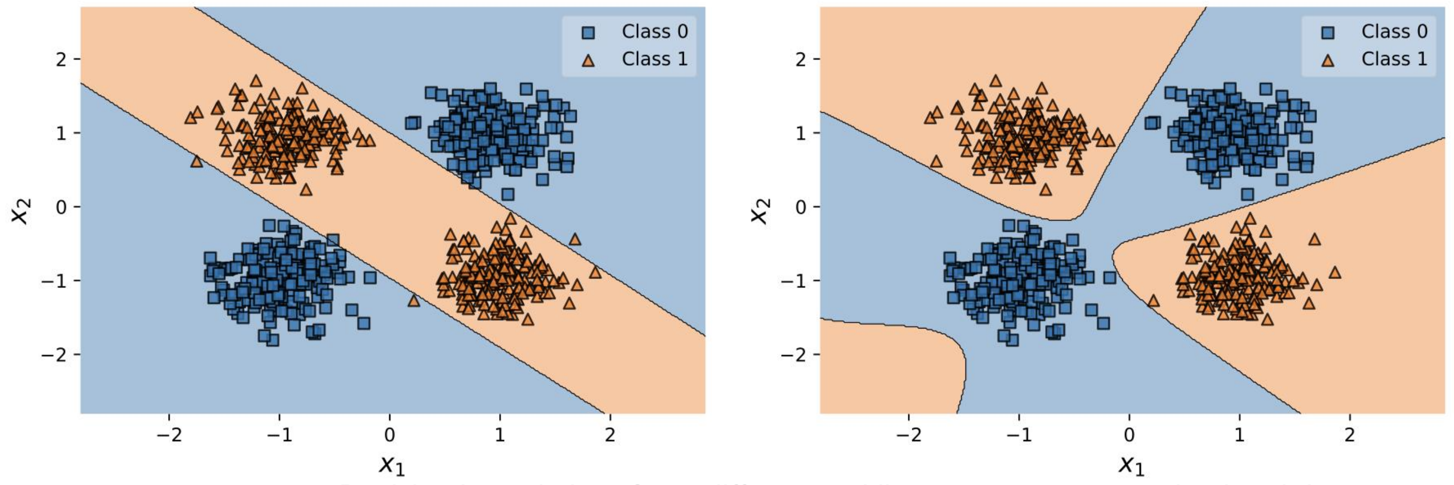
“Combined Logistic Models”



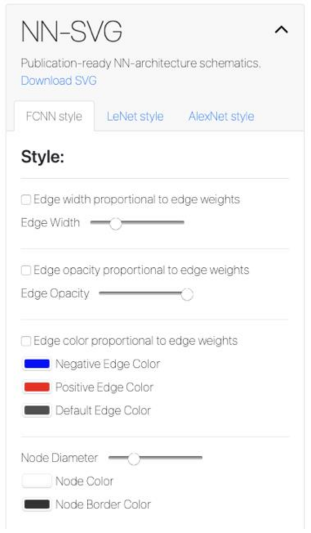
“Combined Logistic Models”



Multilayer Neural Networks Can Solve XOR



Decision boundaries of two different multilayer perceptrons on simulated data solving the XOR problem



NN-SVG

Publication-ready NN-architecture schematics.
Download SVG

FCNN style | LeNet style | AlexNet style

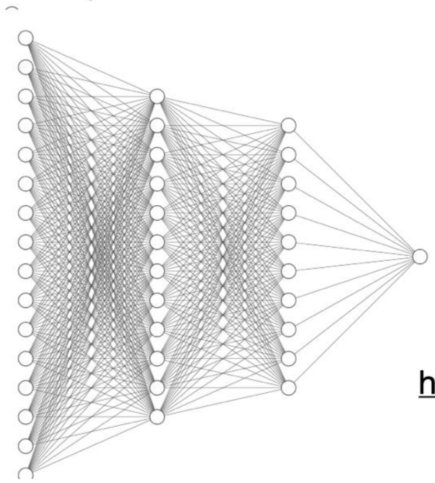
Style:

☐ Edge width proportional to edge weights
Edge Width

☐ Edge opacity proportional to edge weights
Edge Opacity

☐ Edge color proportional to edge weights
🟦 Negative Edge Color
🟥 Positive Edge Color
⬛ Default Edge Color

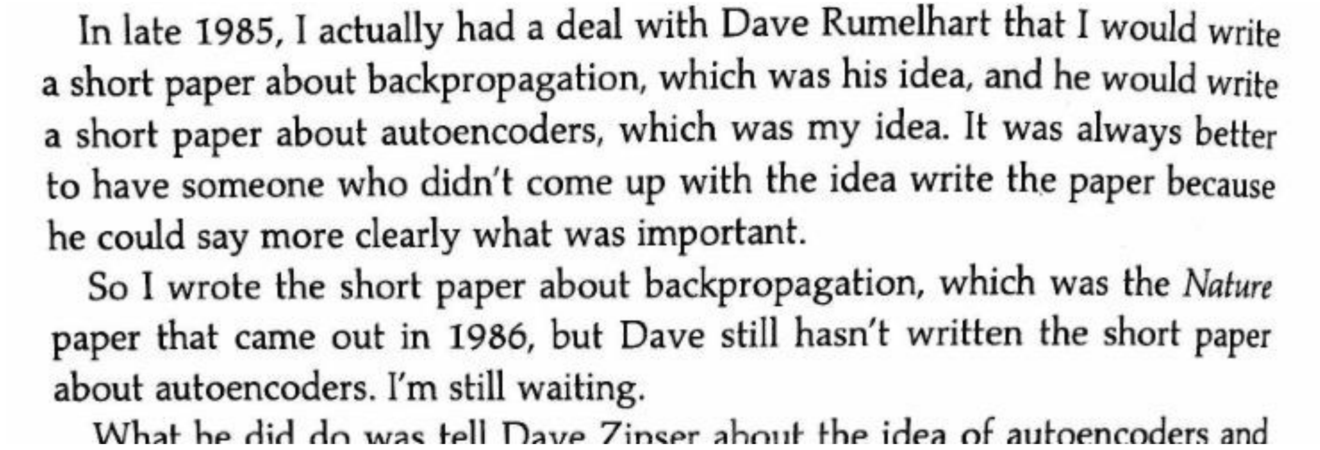
Node Diameter
⬜ Node Color
⬛ Node Border Color



https://alexlenail.me/NN-SVG/index.html

A new problem: Training

- How can we train a multilayer model?
  - No targets / ground truth for the hidden nodes
- Solution: Backpropagation
  - Independently formulated many times
    - http://people.idsia.ch/~juergen/who-invented-backpropagation.html
  - Rumelhart and Hinton (1986) showed that it really works
    - Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). *Learning representations by back-propagating errors.* Nature, 323(6088), 533.


In late 1985, I actually had a deal with Dave Rumelhart that I would write a short paper about backpropagation, which was his idea, and he would write a short paper about autoencoders, which was my idea. It was always better to have someone who didn't come up with the idea write the paper because he could say more clearly what was important.

So I wrote the short paper about backpropagation, which was the *Nature* paper that came out in 1986, but Dave still hasn't written the short paper about autoencoders. I'm still waiting.

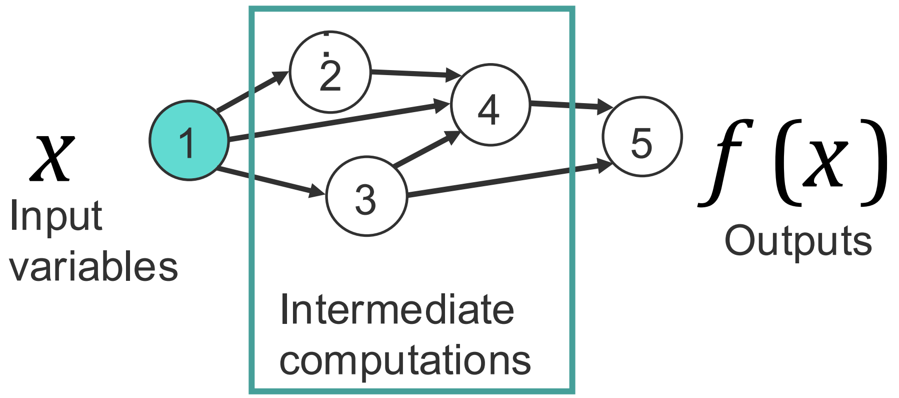
What he did do was tell Dave Zinser about the idea of autoencoders and

– Geoffrey Hinton in Talking Nets - An Oral History of Neural Networks, pg. 380

5. Training Deep Models

Backpropagation

- Neural networks are function compositions that can be represented as computation graphs:



$$\frac{\partial f_n}{\partial x} =$$

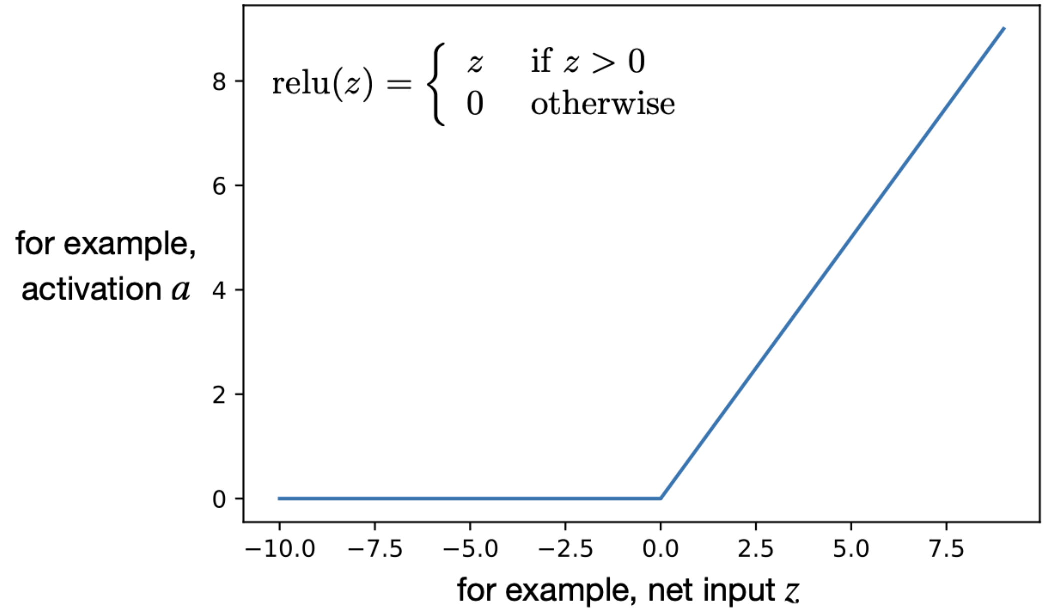
- By applying the chain rule, and working in reverse order, we get:

$$\frac{\partial f_n}{\partial x} = \sum_{i_1 \in \pi(n)} \frac{\partial f_n}{\partial f_{i_1}} \frac{\partial f_{i_1}}{\partial x} = \sum_{i_1 \in \pi(n)} \frac{\partial f_n}{\partial f_{i_1}} \sum_{i_2 \in \pi(i_1)} \frac{\partial f_{i_1}}{\partial f_{i_2}} \frac{\partial f_{i_1}}{\partial x} = \ldots$$

Computation graphs: ReLU

Suppose we have the following activation function:

$$a(x, w, b) = relu(w \cdot x + b)$$

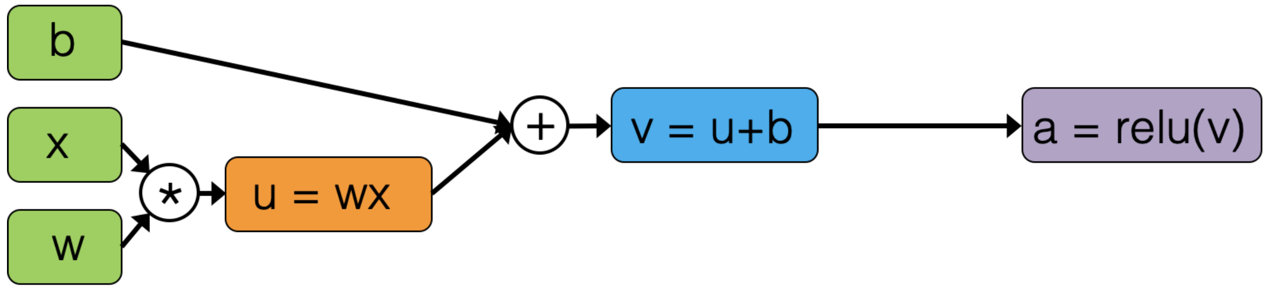


ReLU = Rectified Linear Unit

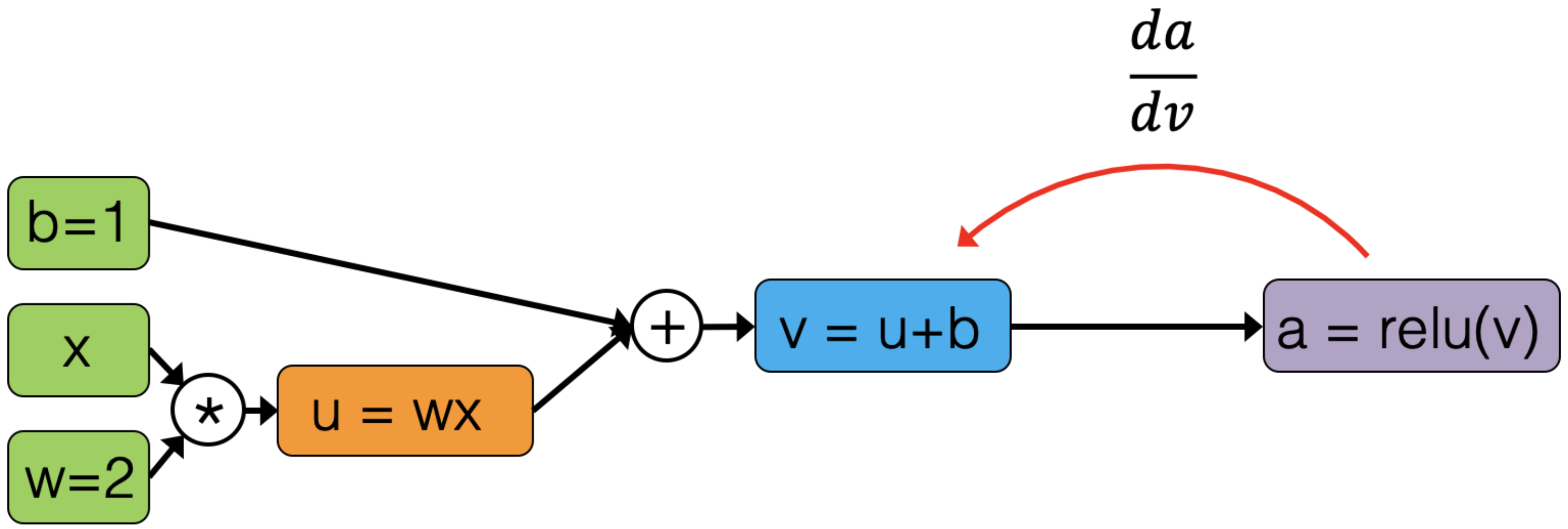
(prob. the most commonly used activation function in DL)

Computation graphs: ReLU

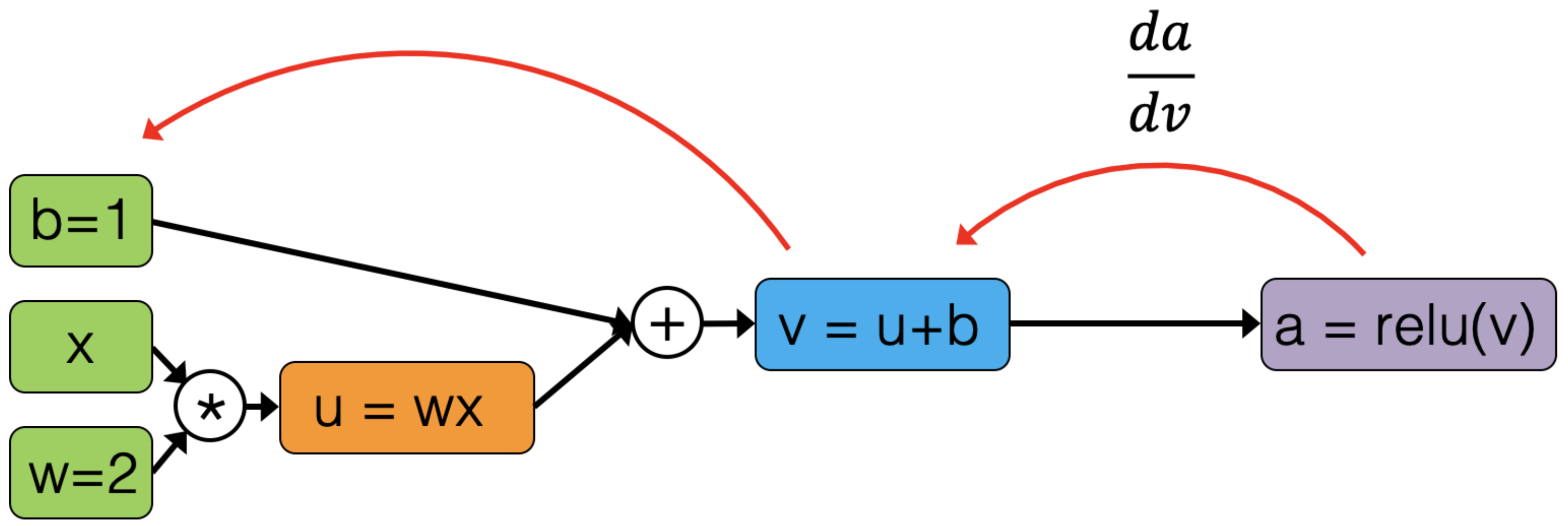
$$a(x, w, b) = relu(\underbrace{\underbrace{w \cdot x}_{u} + b}_{v})$$



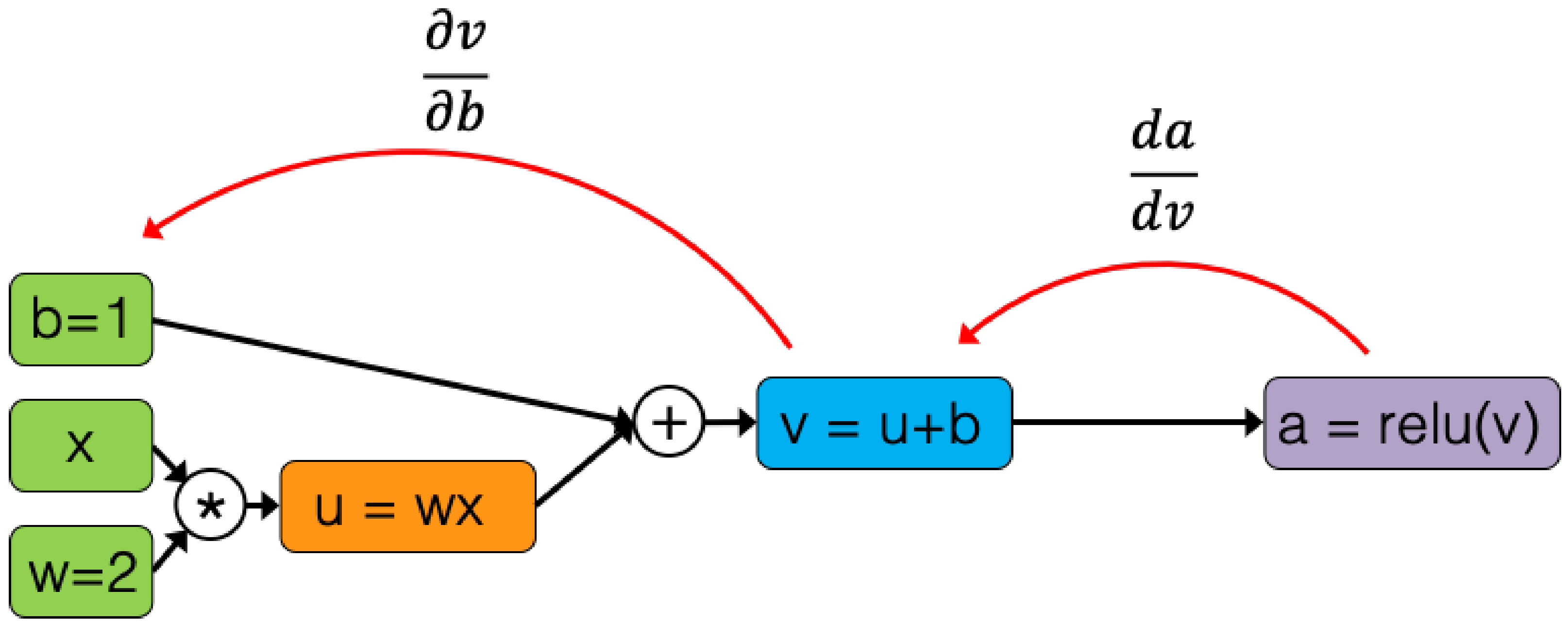
Computation graphs: ReLU



Computation graphs: ReLU

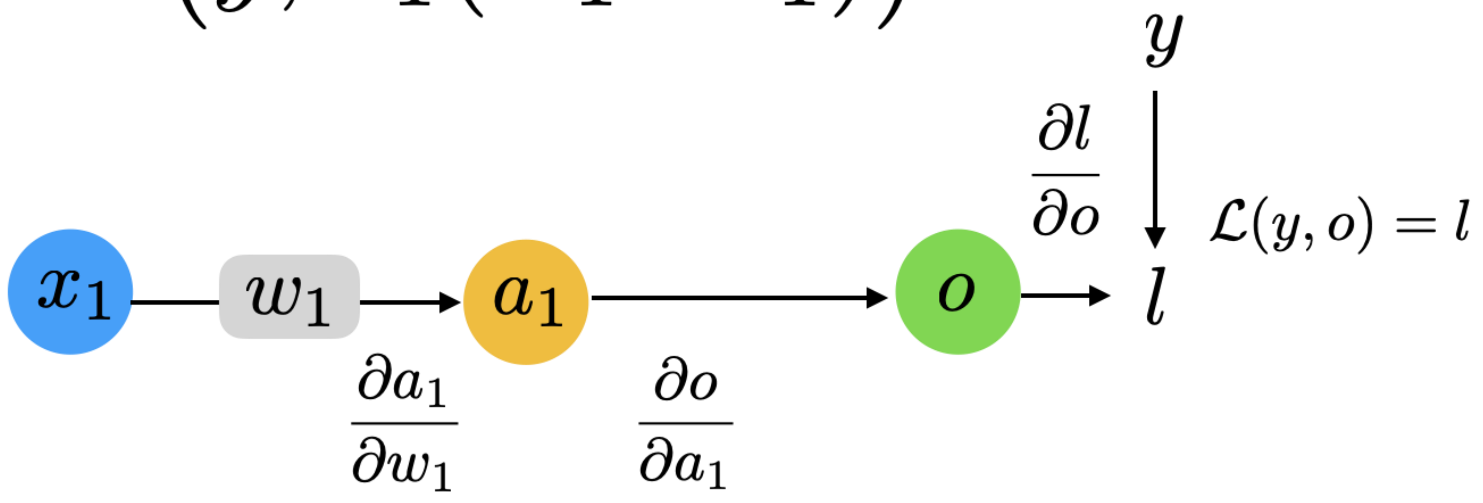


Computation graphs: ReLU



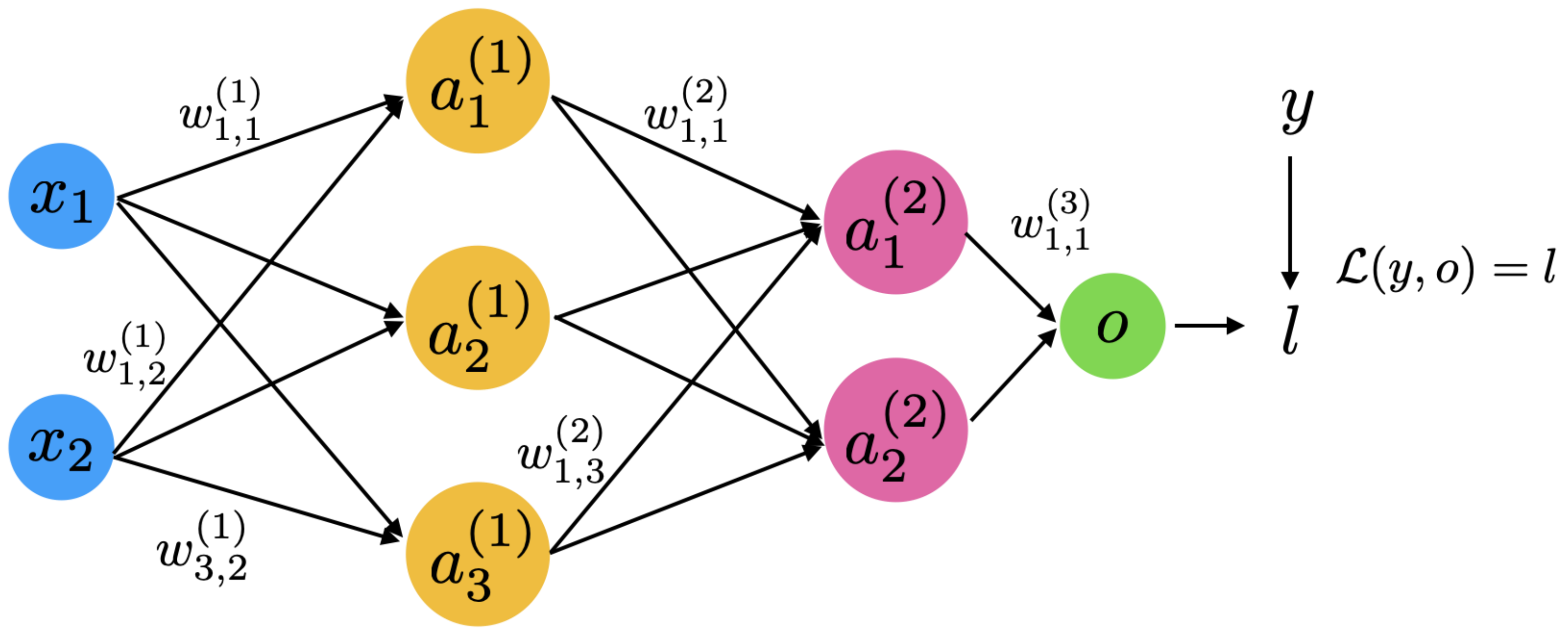
Computation graphs: Single-path

$$\mathcal{L}(y, \sigma_1(w_1 \cdot x_1))$$


y
$\frac{\partial l}{\partial o}$
$\mathcal{L}(y, o) = l$
x_1
w_1
a_1
o
l
$\frac{\partial a_1}{\partial w_1}$
$\frac{\partial o}{\partial a_1}$

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1}$$ (univariate chain rule)

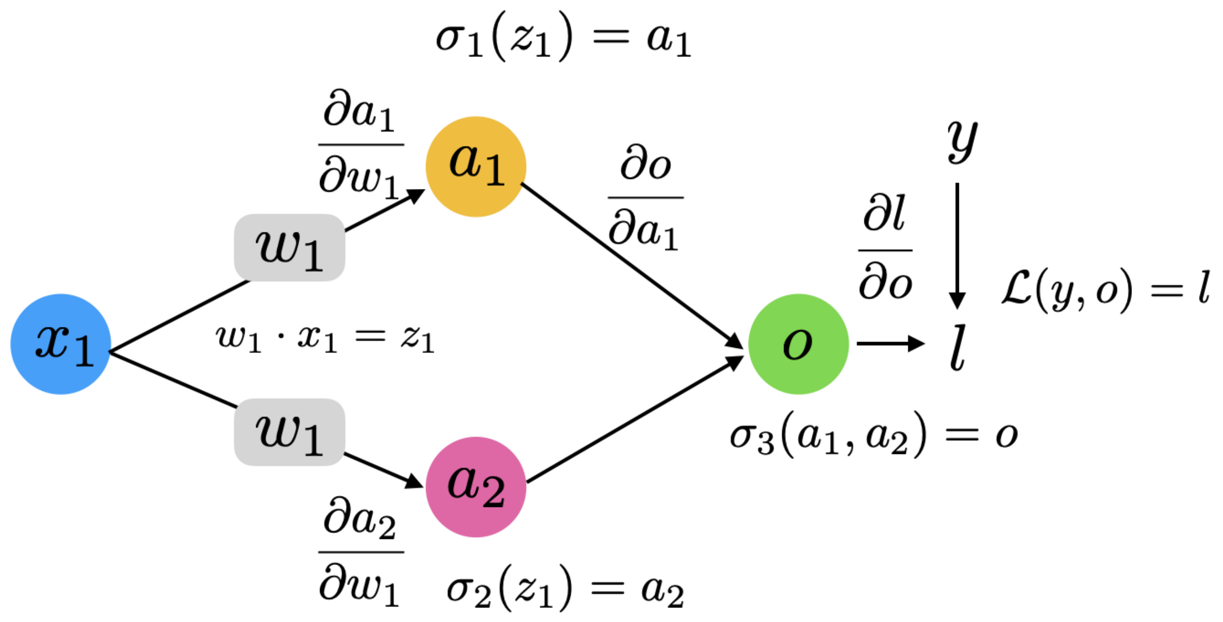
Computation graphs: Fully-Connected Layer



$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

Computation graphs: Weight-Sharing

$$\mathcal{L}\Big(y, \sigma_3\big[\sigma_1(w_1 \cdot x_1), \sigma_2(w_1 \cdot x_1)\big]\Big)$$



$$\frac{\partial l}{\partial w_1} = \underbrace{\frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1}}^{\text{Upper path}} + \underbrace{\frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_1}}_{\text{Lower path}}$$ (multivariable chain rule)

PyTorch: Automated Differentiation

PyTorch Usage: Step 1 (Definition)

```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_feat, num_h1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_h1, num_h2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_h2, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_2(out)
        out = F.relu(out)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas
```

Backward will be inferred automatically if we use the nn.Module class!

Define model parameters that will be instantiated when created an object of this class

Define how and it what order the model parameters should be used in the forward pass

PyTorch Usage: Step 2 (Creation)

```
torch.manual_seed(random_seed)
model = MultilayerPerceptron(num_features=num_features,
                             num_classes=num_classes)

model = model.to(device)

optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
```

Instantiate model
(creates the model parameters)

Define an optimization method

PyTorch Usage: Step 3 (Training)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        cost = F.cross_entropy(probas, targets)
        optimizer.zero_grad()

        cost.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy
```

Run for a specified number of epochs

Iterate over minibatches in epoch

If your model is on the GPU, data should also be on the GPU

y = model(x) calls .__call__ and then .forward(), where some extra stuff is done in __call__;

don't run y = model.forward(x) directly

Gradients at each leaf node are accumulated under the .grad attribute, not just stored. This is why we have to zero them before each backward pass

PyTorch Usage: Step 3 (Training)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy
```

This will run the forward() method

Define a loss function to optimize

Set the gradient to zero
(could be non-zero from a previous forward pass)

Compute the gradients, the backward is automatically constructed by "autograd" based on the forward() method and the loss function

Use the gradients to update the weights according to the optimization method (defined on the previous slide)
E.g., for SGD, $w := w + \text{learning\_rate} \times \text{gradient}$

PyTorch Usage: Step 3 (Training)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy
```

For evaluation, set the model to eval mode (will be relevant later when we use DropOut or BatchNorm)

This prevents the computation graph for backpropagation from automatically being build in the background to save memory

Questions?

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON