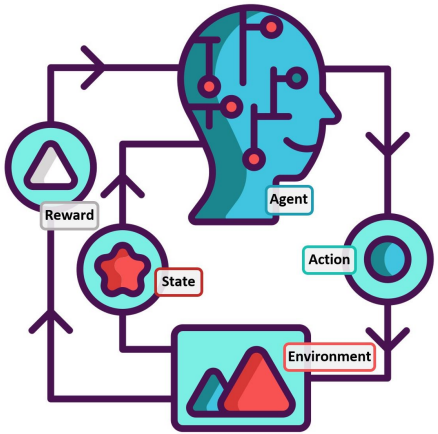
AMCO
ARTIFICIAL INTELLIGENCE, MACHINE LEARNING AND CONTROL RESEARCH GROUP

Reinforcement Learning
2025/2026

RL2
Reinforcement Learning Research Lab @ UniPD

Lecture #15
Neural Networks and Deep Learning



Riccardo De Monte
Gian Antonio Susto

About me

Research interests:

1. Continual Learning for vision.
2. RL and vision.
3. RL and robotics.
4. Unlearning for vision.

My email: riccardo.demonte@phd.unipd.it

Once upon a time, in very dark times...

Last lab we saw prediction and control using dynamic programming, exploiting the knowledge of the state transition matrix **P** of the MDP. However, it might be that such transition is unknown, or too messy to deal with.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

It’s a joke

No RL today

but ...

Recall: Approximate Solution Methods

With problems with really large state space (or state-action space), computation of 'exact' value function is unfeasible.

Instead of considering tables representations, we will consider parametric approximation of value functions (and state-value functions):

$$\hat{v}(s,\mathbf{w}) \approx v_\pi(s)$$

Chess

Number of states ~ 10^{123}
41 zeros

Go

Number of states ~ 10^{360}
120 zeros

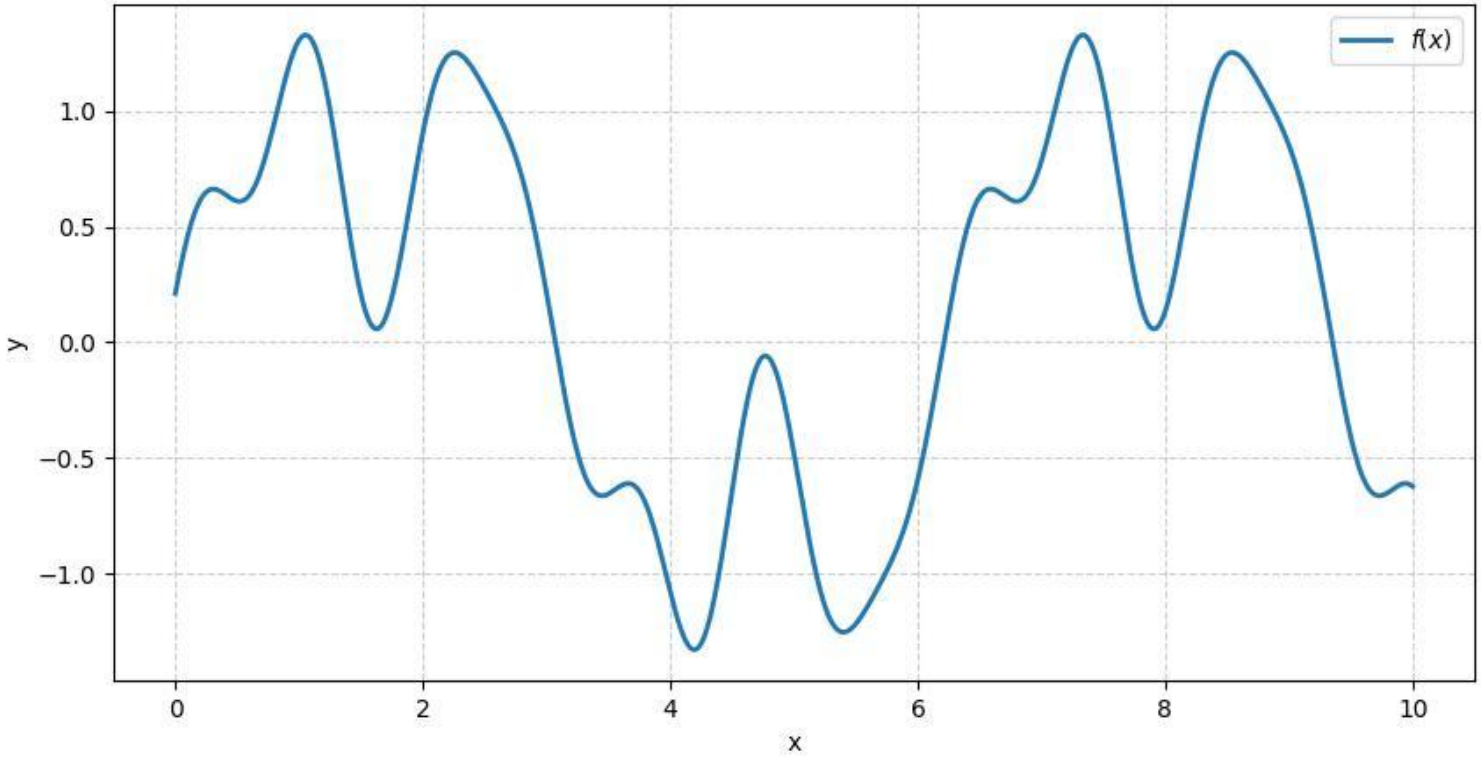
PART 1

Back to Machine Learning

A famous ML problem: regression

We aim to learn a function mapping some input to an output, given some examples.

$$y = f(x) \quad x \in \mathbb{R}, y \in \mathbb{R}$$

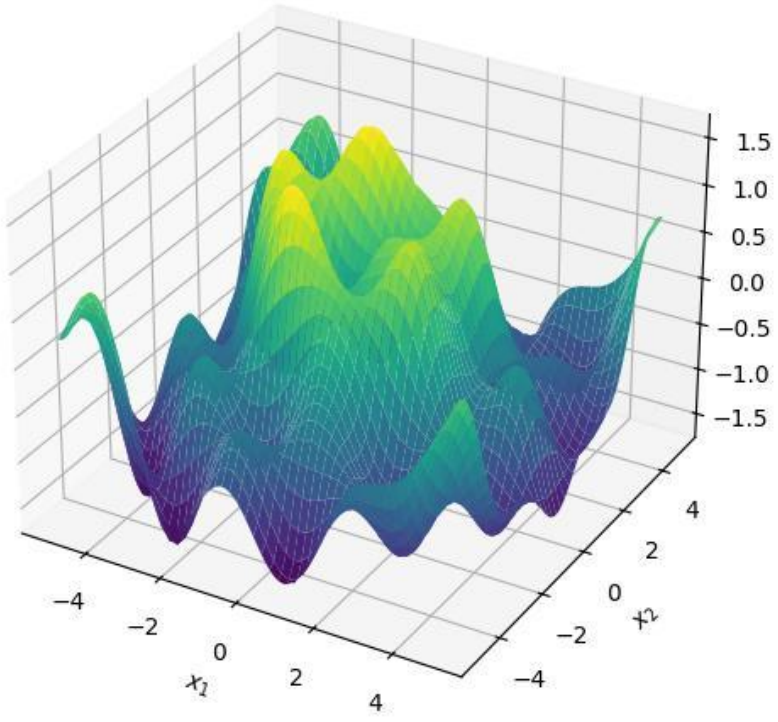

f(x)
1.0
0.5
0.0
−0.5
−1.0
y
0
2
4
6
8
10
x

A famous ML problem: regression

We aim to learn a function mapping some input to an output, given some examples.

$$y = f(x) \quad x \in \mathbb{R}, y \in \mathbb{R}$$

$$y = f(\boldsymbol{x}) \quad \boldsymbol{x} \in \mathbb{R}^d, y \in \mathbb{R}$$


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4
−2
0
2
4
x_1
4
2
0
−2
−4
x_2

A famous ML problem: regression

Function f is unknown

$$y = f(x) \quad x \in \mathbb{R}, y \in \mathbb{R}$$

A famous ML problem: regression

Function f is unknown

$$y = f(x) \quad x \in \mathbb{R}, y \in \mathbb{R}$$

We are given a dataset with N samples:

$$\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$$

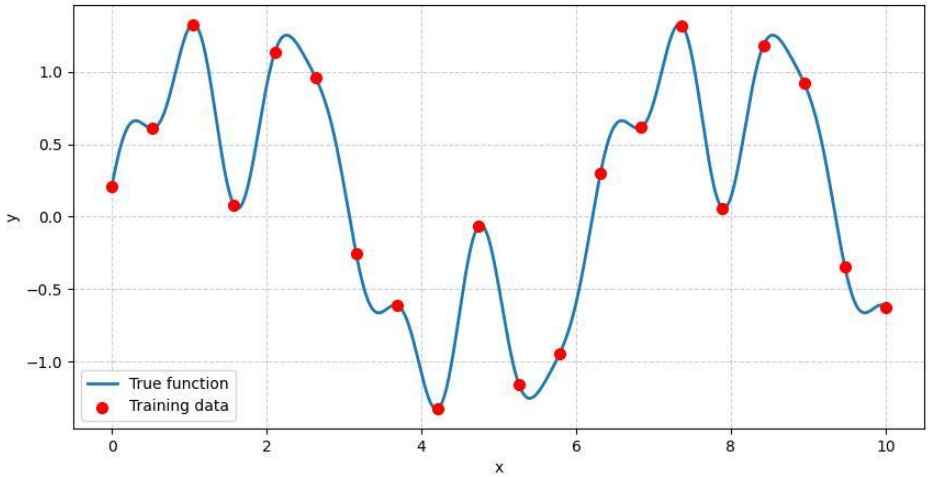
A famous ML problem: regression

Function f is unknown

$$y = f(x) \quad x \in \mathbb{R}, y \in \mathbb{R}$$

We are given a dataset with N samples:

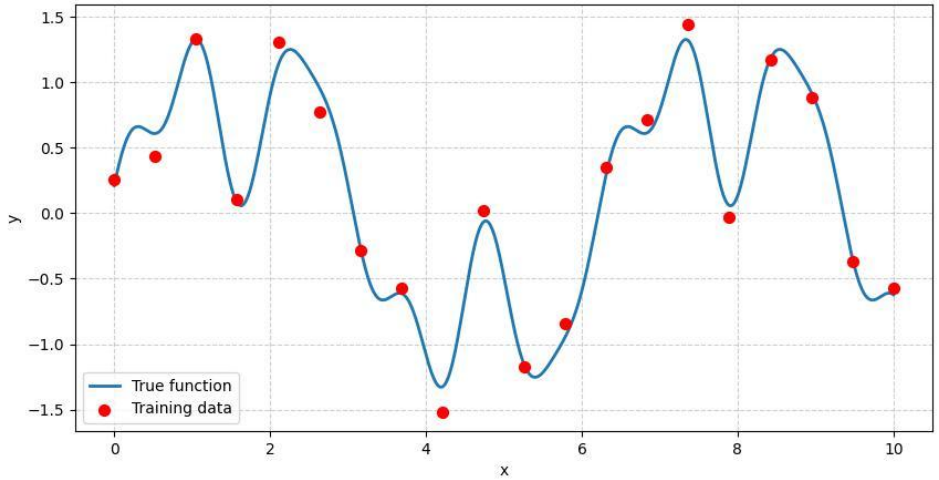
$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$$



However, in general, ...

There might exist a “true” function f, but we do not have access to a real couple (x,f(x)).
Instead we have (x,y), where y is a noisy version f(x).

A common example: $$y = f(x) + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, \sigma)$$



We can assume there exists an unknown distribution

In general:

$$(\boldsymbol{x}, y) \sim p(\boldsymbol{x}, y) = p(y|\boldsymbol{x})p(\boldsymbol{x})$$

We can assume there exists an unknown distribution

In general:

$$(\boldsymbol{x}, y) \sim p(\boldsymbol{x}, y) = p(y|\boldsymbol{x})p(\boldsymbol{x})$$

In the previous example (noisy measurements):

$$y = f(x) + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, \sigma) \implies p(y|x) = \mathcal{N}(f(x), \sigma)$$

For simplicity, consider the noisy measurements example

$$y = f(x) + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, \sigma) \implies p(y|x) = \mathcal{N}(f(x), \sigma)$$

For simplicity, consider the noisy measurements example

$$y = f(x) + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, \sigma) \implies p(y|x) = \mathcal{N}(f(x), \sigma)$$

1. We are given a dataset

$$\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$$

For simplicity, consider the noisy measurements example

$$y = f(x) + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, \sigma) \implies p(y|x) = \mathcal{N}(f(x), \sigma)$$

1. We are given a dataset

$$\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$$

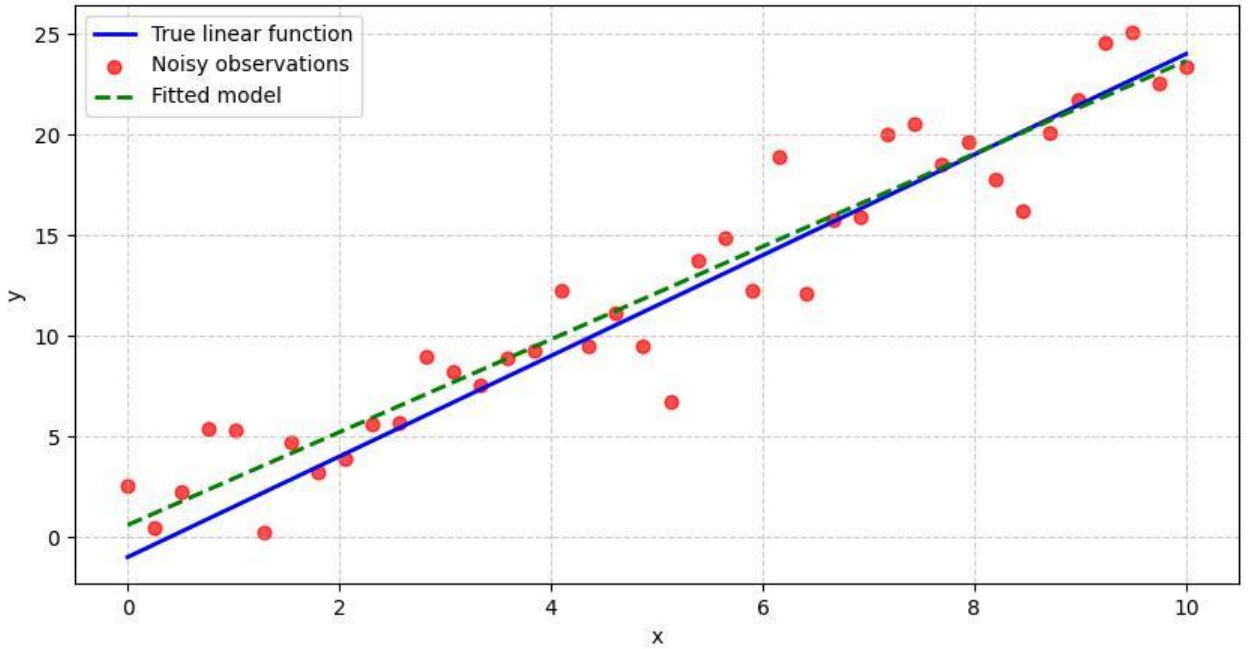
2. We aim to approximate f, by looking for a good (parametric) approximator

$$\hat{y} = \hat{f}(\boldsymbol{x}; \boldsymbol{w}) \approx f(\boldsymbol{x}) \qquad \boldsymbol{w} \in \mathbb{R}^n$$

A simple parametric family for the approximator: linear models

$$\hat{f}(\boldsymbol{x};\boldsymbol{w}) = \sum_{i=1}^{d} v_i x_i + b = \boldsymbol{v}^\top \boldsymbol{x} + b, \quad \boldsymbol{w} = \begin{bmatrix} v \\ b \end{bmatrix} \in \mathbb{R}^{n=d+1}$$

A simple parametric family for the approximator: linear models



Up to now

We are given a regression task with unknown f. Moreover:

1. We assume to have a reasonable family of possible parametric function.

Up to now

We are given a regression task with unknown f. Moreover:

1. We assume to have a reasonable family of possible parametric function.

2. We are given a dataset.

Up to now

We are given a regression task with unknown f. Moreover:

1. We assume to have a reasonable family of possible parametric function.

2. We are given a dataset.

3. We need a criteria to evaluate any possible function that approximates f.

Loss function

For regression, we usually use the squared loss (L2 loss):

$$l(x, y, w) = (y - \hat{f}(x; w))^2$$

Generalized Error

We aim to find the best w:

$$\min_w \mathrm{GE}(w) = \min_w \mathbb{E}_{p(x,y)}\left[l(x,y,w)\right]$$

The best we can do: Empirical Risk Minimization

(Recall Monte Carlo estimation)

$$\min_{w} \mathcal{L}(\mathcal{D}, w) = \min \frac{1}{N} \sum_{i=1}^{N} l(x_i, y_i, w)$$

(Supervised) Machine Learning

1. We aim to learn a good approximation of an unknown function f.

(Supervised) Machine Learning

1. We aim to learn a good approximation of an unknown function f.
2. We are given a dataset D with many samples (x,y).

(Supervised) Machine Learning

1. We aim to learn a good approximation of an unknown function f.
2. We are given a dataset D with many samples (x,y).
3. We define a loss (cost) function.

(Supervised) Machine Learning

1. We aim to learn a good approximation of an unknown function f.
2. We are given a dataset D with many samples (x,y).
3. We define a loss (cost) function.
4. We aim to find w that minimizes the Empirical Risk:

$$\min_{w} \mathcal{L}(\mathcal{D}, w) = \min \frac{1}{N} \sum_{i=1}^{N} l(x_i, y_i, w)$$

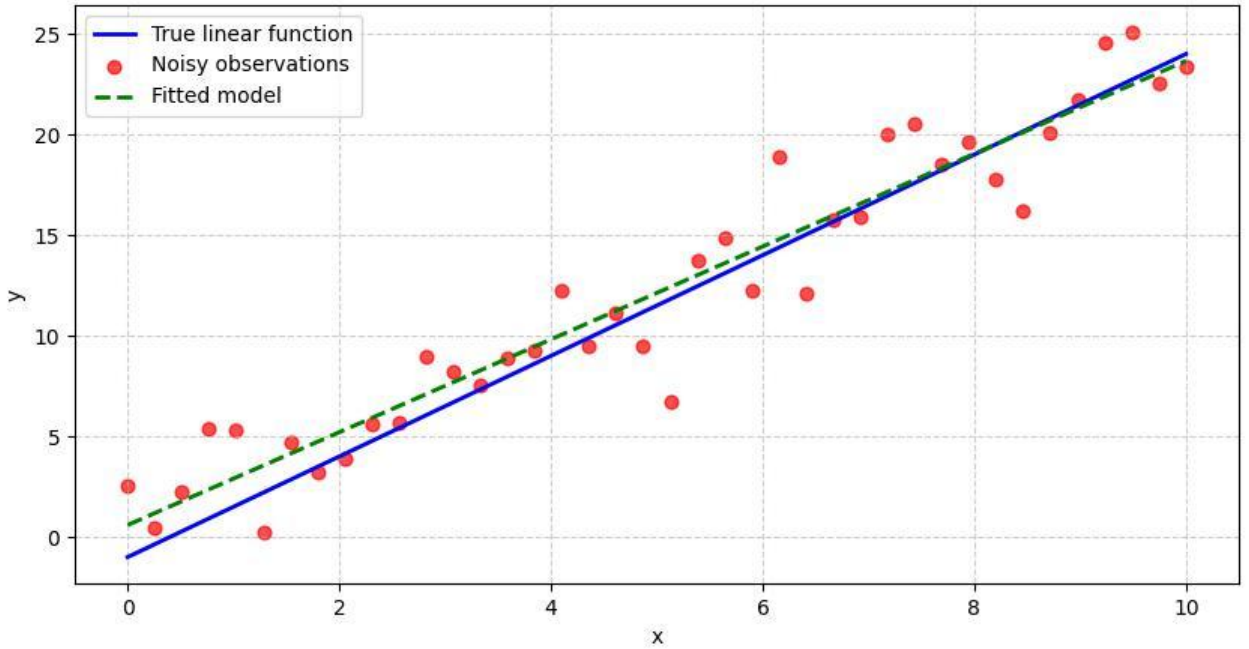
(Supervised) Machine Learning

1. We aim to learn a good approximation of an unknown function f.
2. We are given a dataset D with many samples (x,y).
3. We define a loss (cost) function.
4. We aim to find w that minimizes the Empirical Risk:

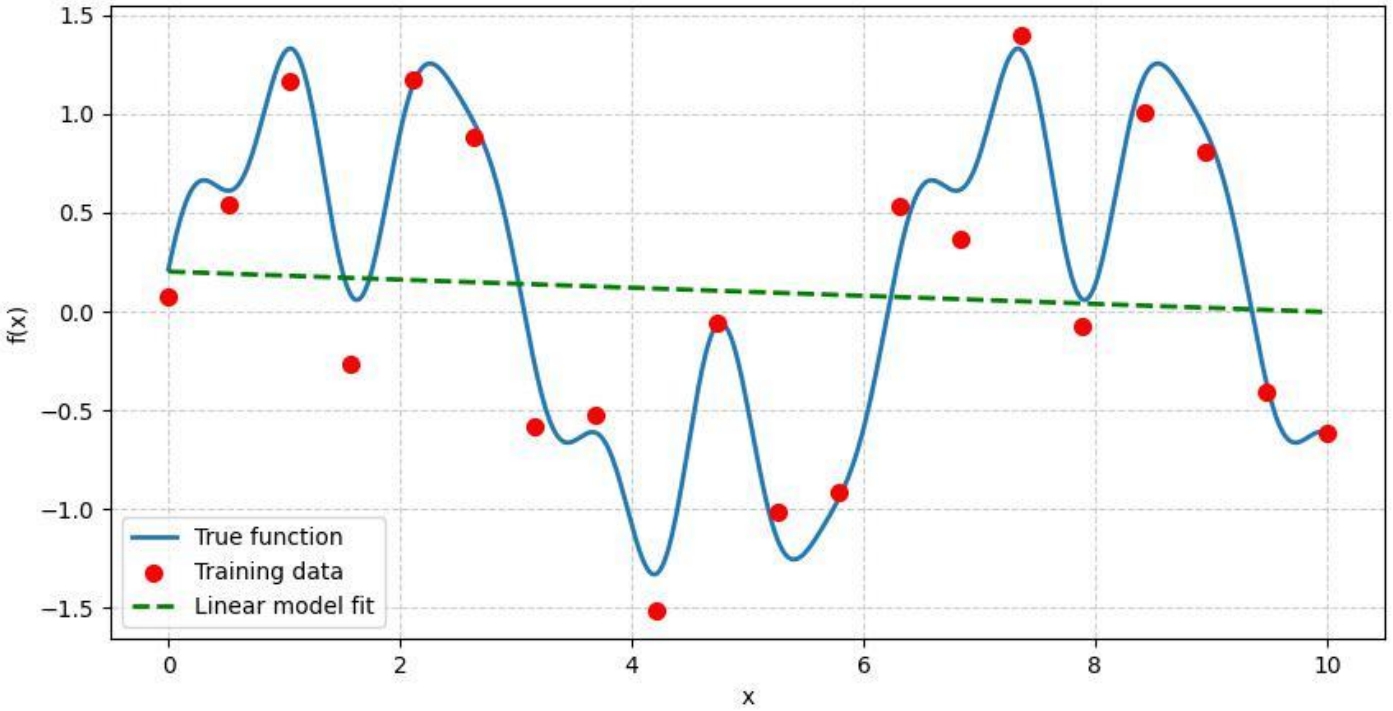
$$\min_{w} \mathcal{L}(\mathcal{D}, w) = \min \frac{1}{N} \sum_{i=1}^{N} l(x_i, y_i, w)$$

5. We “just” need a “procedure” to solve this optimization problem.

A simple parametric family for the approximator: linear models



Linear models might be not enough



Possible solutions

1. Kernel functions (recall SVM).

Possible solutions

1. Kernel functions (recall SVM).

2. Feature engineering: some example in Computer Vision (Bag of Visual Words + SVM).

Possible solutions

1. Kernel functions (recall SVM).

2. Feature engineering: some examples in Computer Vision (Bag of Visual Words + SVM).

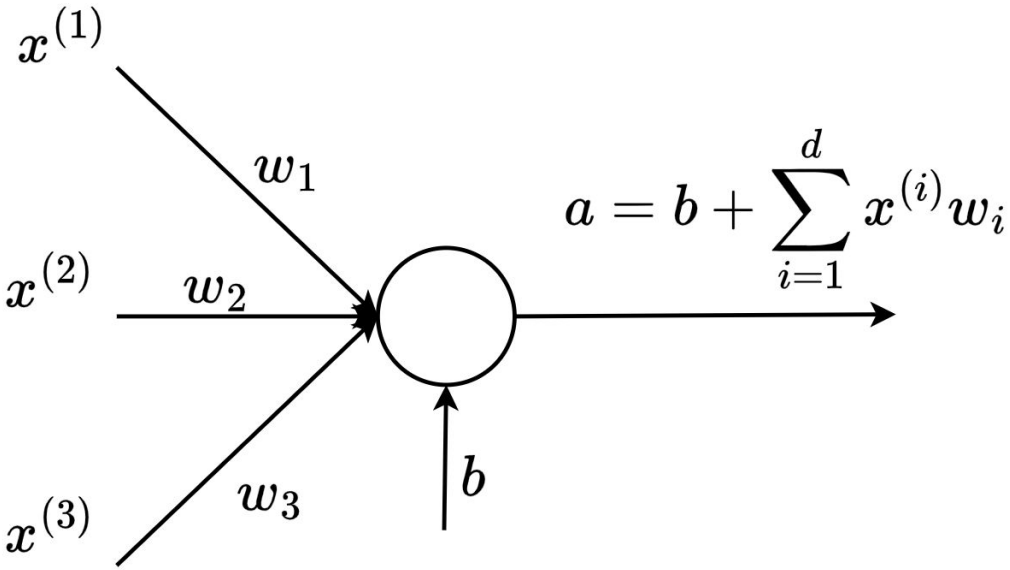
3. **Neural Networks**.

Neural Networks

No Perceptron

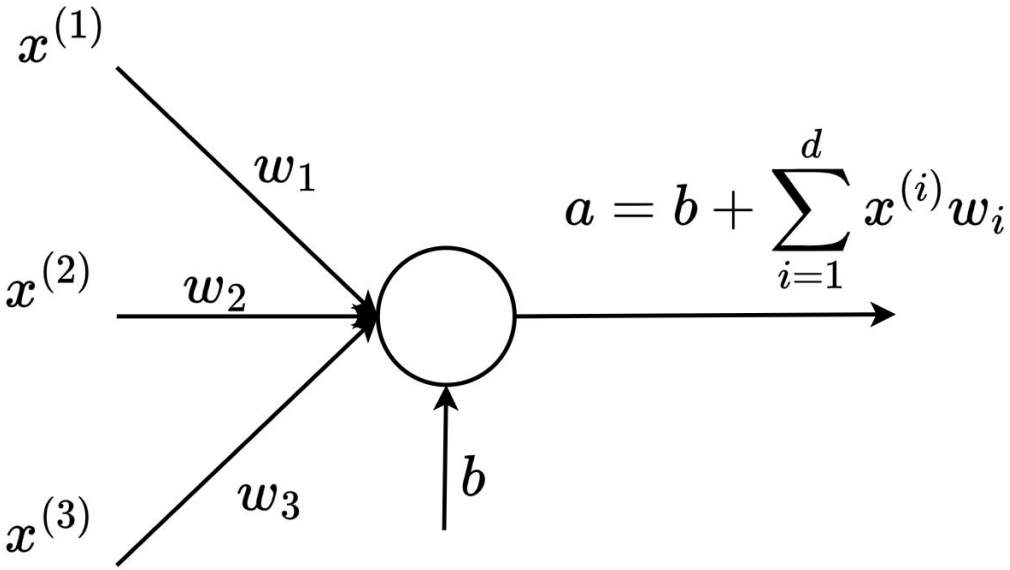
It’s boring and I don’t have enough time

Linear Unit = Linear model



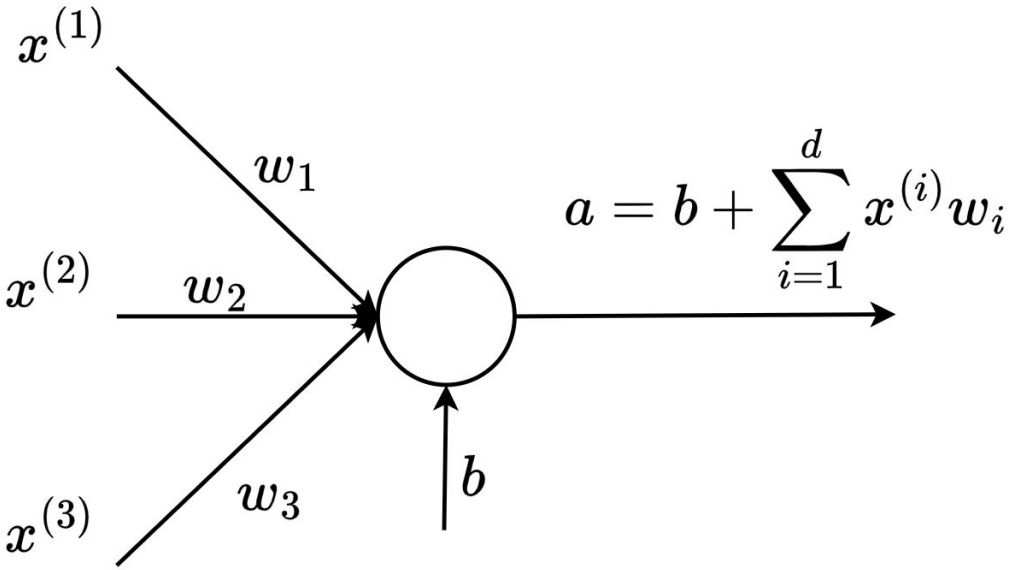
1. x is a d-dimensional input

Linear Unit = Linear model



1. x is a d-dimensional input
2. A linear unit has a set of parameters w: d weights and a bias

Linear Unit = Linear model

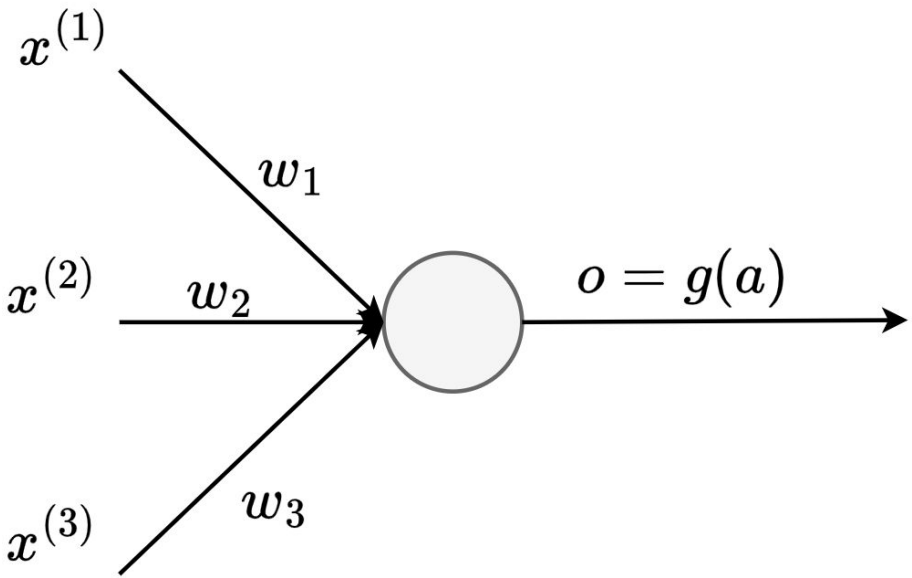


1. x is a d-dimensional input
2. A linear unit has a set of parameters w: d weights and a bias
3. The output is called **activation**

We need more **expressiveness**

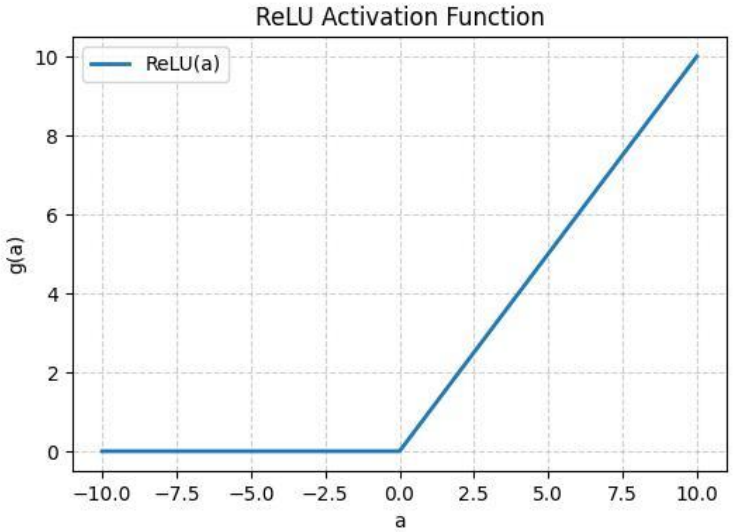
The Artificial Neuron: Linear Unit + Activation Function

We just add a non-linearity



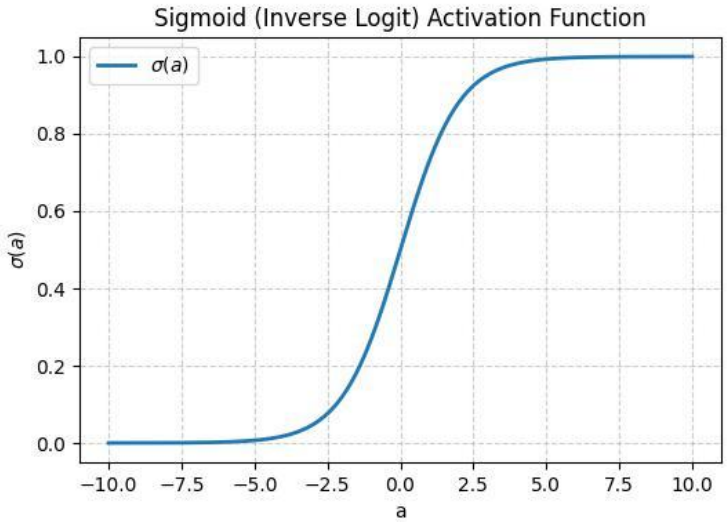
Rectified Linear Unit (ReLU)

$$g(a) = \begin{cases} a & a \geq 0 \\ 0 & a < 0 \end{cases}$$



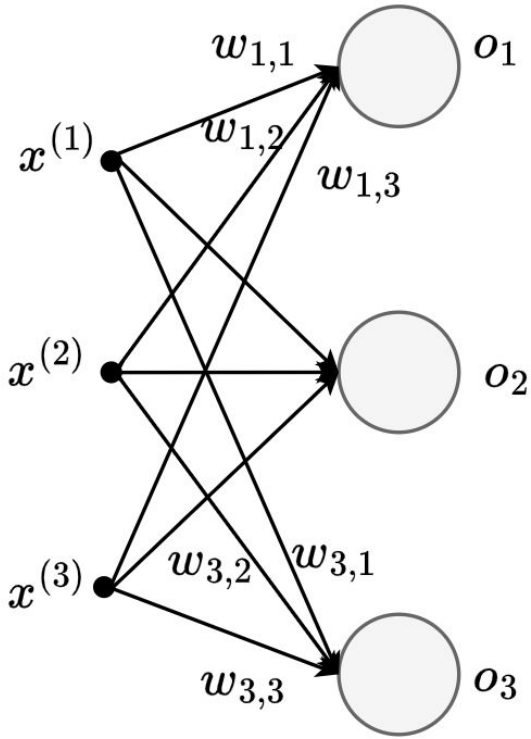
Sigmoid

$$g(a) = \sigma(a) = \frac{1}{1 + e^{-a}}$$



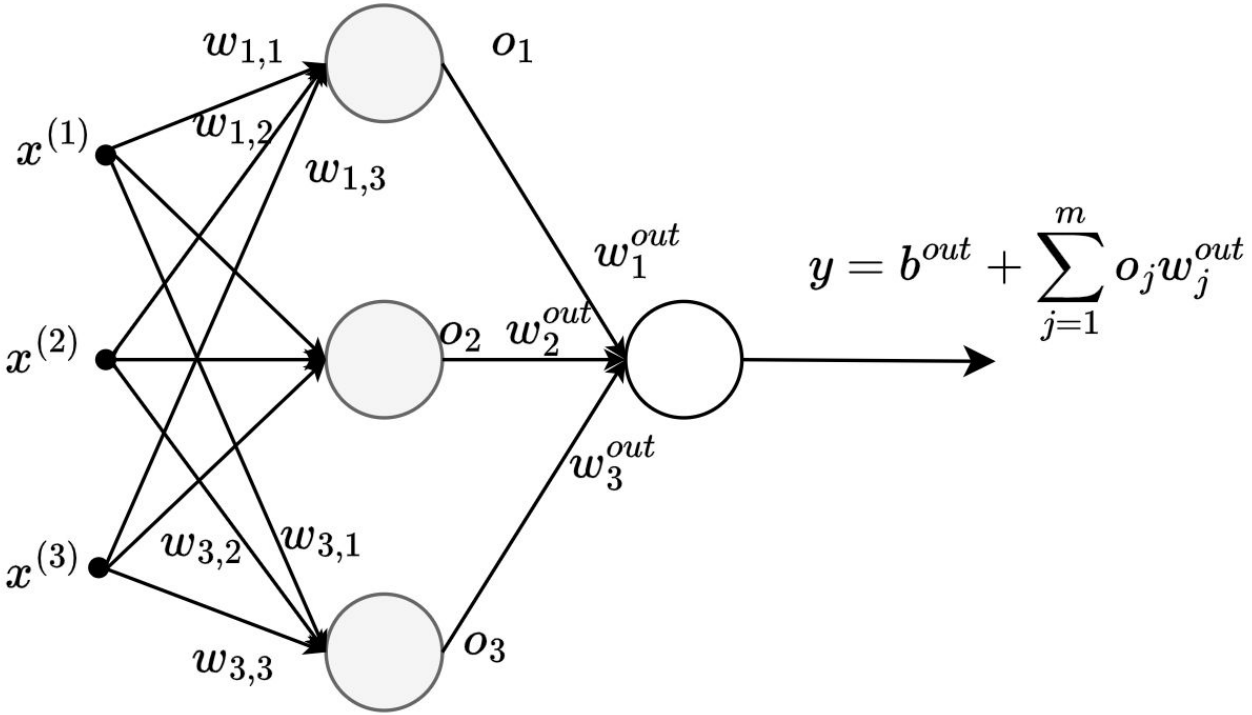
We need more **expressiveness**

More neurons: a layer of neurons



$$o_2 = g\left(b_2 + \sum_{i=1}^{d} x^{(i)} w_{2,i}\right)$$

Single-layer Feed Forward Neural Network



Single-layer Feed Forward Neural Network

1. The layer with neurons is called **hidden layer**

2. The previous Neural Network has just one output. In general, we can have multiple outputs (e.g. for classification).

3. Each neuron has its own set of weights (d weights and one bias).

4. The output unit has m weights, where m = neurons of the hidden layer, and a bias.

Neural Networks, Vol. 2, pp. 359–366, 1989
Printed in the USA. All rights reserved.

0893-6080/89 $3.00 + .00

ORIGINAL CONTRIBUTION

Multilayer Feedforward Networks are Universal Approximators

KUR' HORNIK

Technische Universität Wien

MAXWELL STINCHCOMBE AND HALBERT WHITE

University of California, San Diego

(*Received* 16 *September* 1988; *revised and accepted* 9 *March* 1989)

Abstract—*This paper rigorously establishes that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.*

Keywords—Feedforward networks, Universal approximation, Mapping networks, Network representation capability, Stone-Weierstrass Theorem, Squashing functions, Sigma-Pi networks, Back-propagation networks.

Matrix Notation

1. Each neuron has a set of d weights. We can consider each set of weights as one row of a matrix with d columns:

$$\boldsymbol{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,d} \\ & \vdots & & \\ w_{p,1} & w_{p,2} & \cdots & w_{p,d} \\ & \vdots & & \\ w_{m,1} & w_{m,2} & \cdots & w_{m,d} \end{bmatrix} \in \mathbb{R}^{m \times d}, \qquad \boldsymbol{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_p \\ \vdots \\ b_m \end{bmatrix} \in \mathbb{R}^m$$

2. In matrix notation the output of the hidden layer is a m-dimensional vector:

$$\boldsymbol{a} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}, \quad \boldsymbol{o} = g(\boldsymbol{a}) = g\big(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\big)$$


THAT'S NOT ENOUGH
WE HAVE TO GO DEEPER

From shallow NNs to Deep NNs

1. The previous paper states: with one hidden layer we can approximate any function.

From shallow NNs to Deep NNs

1. The previous papers says: with one hidden layer we can approximate any function.

2. Instead of one hidden layer we can have many hidden layers.

From shallow NNs to Deep NNs

1. The previous papers says: with one hidden layer we can approximate any function.

2. Instead of one hidden layer we can have many hidden layers.

3. Some papers show that empirically deeper networks are easier to “train”.

From shallow NNs to Deep NNs

1. The previous papers says: with one hidden layer we can approximate any function.

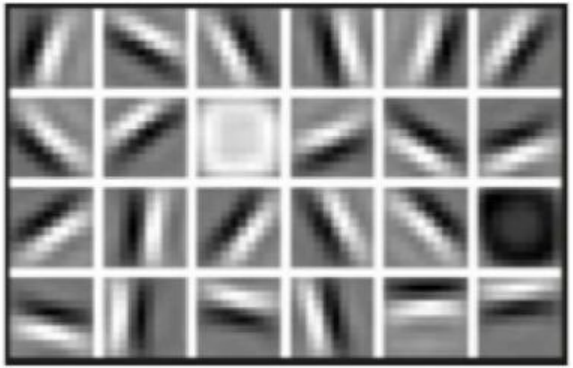
2. Instead of one hidden layer we can have many hidden layers.

3. Some papers show that empirically deeper networks are easier to “train”.

4. Depth allows neural networks to build hierarchical representations; successively transforming raw input into more abstract and meaningful features, mirroring the structure of many natural signals

From shallow NNs to Deep NNs

Low Level Features



Lines & Edges

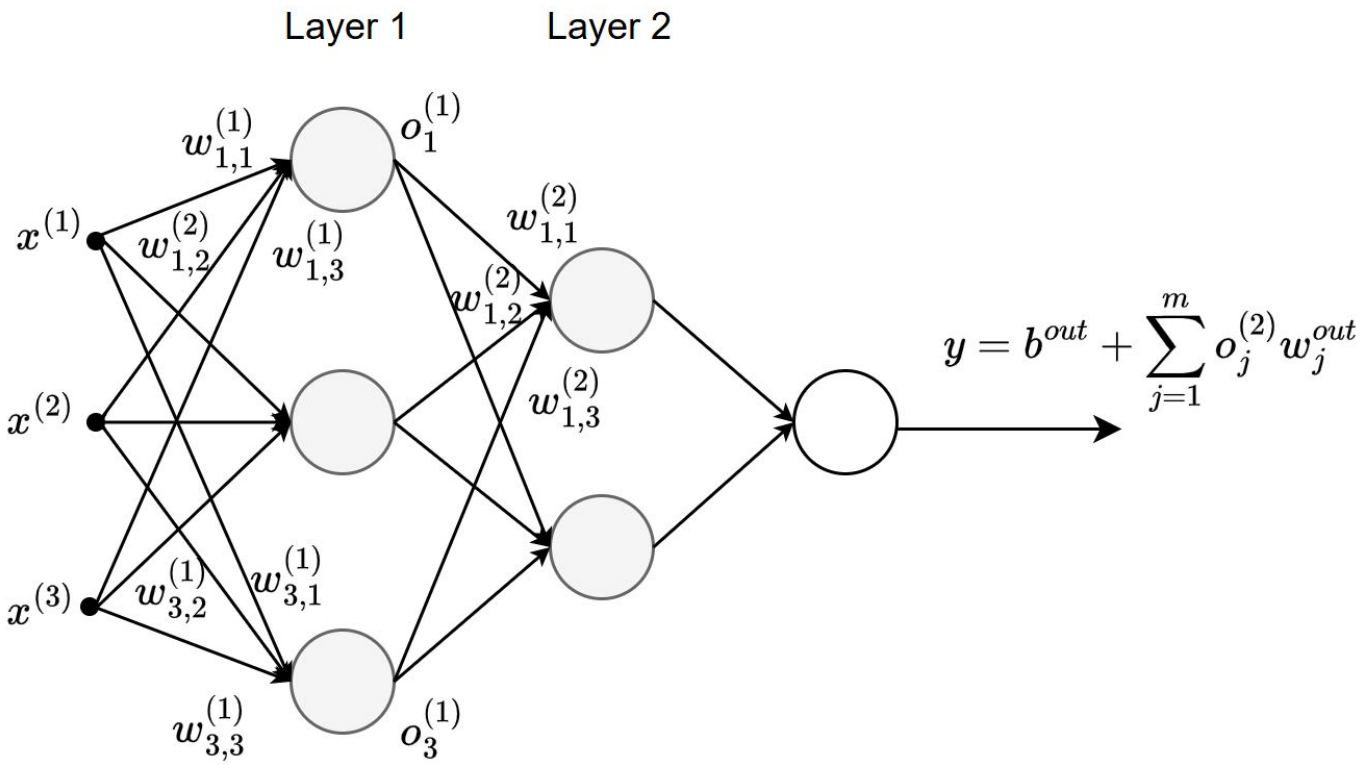
Mid Level Features

Eyes & Nose & Ears

High Level Features

Facial Structure

From shallow NNs to Deep NNs



Matrix notation for multi-layer Neural Networks

$$\boldsymbol{W}^{(l)} = \begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,m^{(l-1)}}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,m^{(l-1)}}^{(l)} \\ \cdots & & & \\ w_{m^l,1}^{(l)} & w_{m^l,2}^{(l)} & \cdots & w_{m^l,m^{(l-1)}}^{(l)} \end{bmatrix} \in \mathbb{R}^{m^{(l)} \times m^{(l-1)}}, \quad \boldsymbol{b}^{(l)} = \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_{m^(l)}^{(l)} \end{bmatrix}$$

Matrix notation for multi-layer Neural Networks

$$\boldsymbol{W}^{(l)} = \begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,m^{(l-1)}}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,m^{(l-1)}}^{(l)} \\ \cdots & & & \\ w_{m^l,1}^{(l)} & w_{m^l,2}^{(l)} & \cdots & w_{m^l,m^{(l-1)}}^{(l)} \end{bmatrix} \in \mathbb{R}^{m^{(l)} \times m^{(l-1)}}, \quad \boldsymbol{b}^{(l)} = \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_{m^(l)}^{(l)} \end{bmatrix}$$

$$\boldsymbol{o}^{(1)} = g(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}), \qquad \boldsymbol{o}^{(l)} = g(\boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)} + \boldsymbol{b}^{(l)})$$

Recap

1. A shallow NN has one hidden layer with m neurons.

2. A deep NN has 2 or more layers.

3. Each layer *l* has *m^(l)* neurons. The input of layer *l* is the output of the previous layer *l-1*.

4. We denote the overall sets of parameters of a NN as:

$$\mathcal{W} = \{\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \cdots, \boldsymbol{W}^{(L)}, \boldsymbol{b}^{(L)}\}$$

```python
import torch
from torch import nn

class FFNN(nn.Module):
    """2-layers FFNN"""

    def __init__(self, input_dim, hidden_size):
        """
        input_dim: d, input dimension
        hidden_size: m, num of neurons per layer
        """
        super().__init__()

        self.first_hidden = nn.Linear(input_dim, hidden_size)
        self.act_1 = nn.ReLU()

        self.second_hidden = nn.Linear(hidden_size, hidden_size)
        self.act_2 = nn.ReLU()

        self.output_layer = nn.Linear(hidden_size, 1)

    def forward(self, x):

        a_1 = self.first_hidden(x)
        o_1 = self.act_1(a_1)

        a_2 = self.second_hidden(o_1)
        o_2 = self.act_2(a_2)

        hat_y = self.output_layer(o_2)

        return hat_y

# fake input
x = torch.randn(10) # d = 10

# init NN
model = FFNN(input_dim=10, hidden_size=128)

# compute output with current weights (random weights)
hat_y = model(x)

print(f"Num parameters: {sum(p.numel() for p in model.parameters())}")
```

✓ 0.0s

```
Num parameters: 18049
```

How to “learn” the optimal parameters?

As for ML

Given a dataset of many samples (x,y), we aim to solve

$$\min_{\mathcal{W}} \mathbb{E}_{p(x,y)}\left[l(x,y,\mathcal{W})\right] \approx \min_{\mathcal{W}} \frac{1}{N}\sum_{i=1}^{N} l(x_i,y_i,\mathcal{W})$$

Empirical Risk Minimization (again)

Fixed the dataset, the overall loss (Empirical Risk) is a function of W

$$\mathcal{L}(\mathcal{D}, \mathcal{W}) = \frac{1}{N} \sum_{i=1}^{N} l(x_i, y_i, \mathcal{W})$$

We can consider W as a super big vector, namely

$$\mathcal{W} \in \mathbb{R}^D, \quad D \gg d$$

Minimize a function in R (D=1)

1. Find stationary points

$$\frac{d\mathcal{L}(\mathcal{D}, w)}{dw} = 0$$

2. If we are lucky, we get a closed form expression for stationary points. Then we can check which point is the global minimum.

What if we have no closed form solution?

What if we have no closed form solution?

$$L(w) = w^2 + \exp(w), \quad L'(w) = 0 \implies \exp(w) = -2w$$

Instead, let's look for an iterative approach

1. We consider w_0, a first (random) guess

2. We compute the first order derivative and evaluate the local slope of the loss

$$\left.\frac{d\mathcal{L}(\mathcal{D}, w)}{dw}\right|_{w=w_0} = \mathcal{L}'(\mathcal{D}, w_0)$$

3. We can do the same and compute the second order derivative:

$$\left.\frac{d\mathcal{L}'(\mathcal{D}, w)}{dw}\right|_{w=w_0} = \mathcal{L}''(\mathcal{D}, w_0)$$

We can use Taylor approximation

1. Given w_0 and the derivatives just computed, we can approximate locally:

$$\mathcal{L}(\mathcal{D}, w) \approx \mathcal{L}(\mathcal{D}, w_0) + \mathcal{L}'(\mathcal{D}, w_0)(w - w_0) + \frac{1}{2}\mathcal{L}''(\mathcal{D}, w_0)(w - w_0)^2$$

We can use Taylor approximation

1. Given w_0 and the derivatives just computed, we can approximate locally:

$$\mathcal{L}(\mathcal{D}, w) \approx \mathcal{L}(\mathcal{D}, w_0) + \mathcal{L}'(\mathcal{D}, w_0)(w - w_0) + \frac{1}{2}\mathcal{L}''(\mathcal{D}, w_0)(w - w_0)^2$$

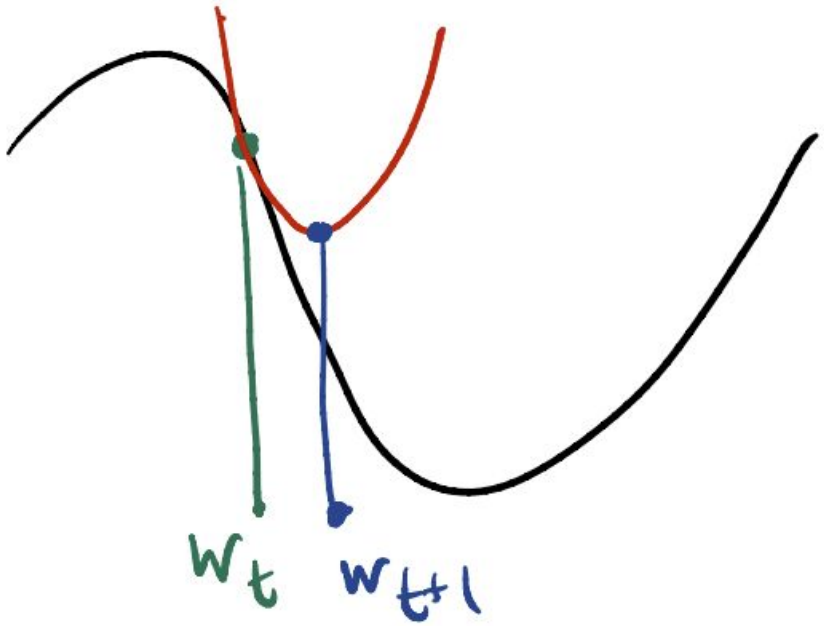
2. Let's compute the stationary points of this:

$$w_1 = w_0 - \frac{\mathcal{L}'(\mathcal{D}, w_0)}{\mathcal{L}''(\mathcal{D}, w_0)}$$

We can repeat this approach multiple times

$$w_t = w_{t-1} - \frac{\mathcal{L}'(\mathcal{D}, w_{t-1})}{\mathcal{L}''(\mathcal{D}, w_{t-1})}$$

We can repeat this approach multiple times

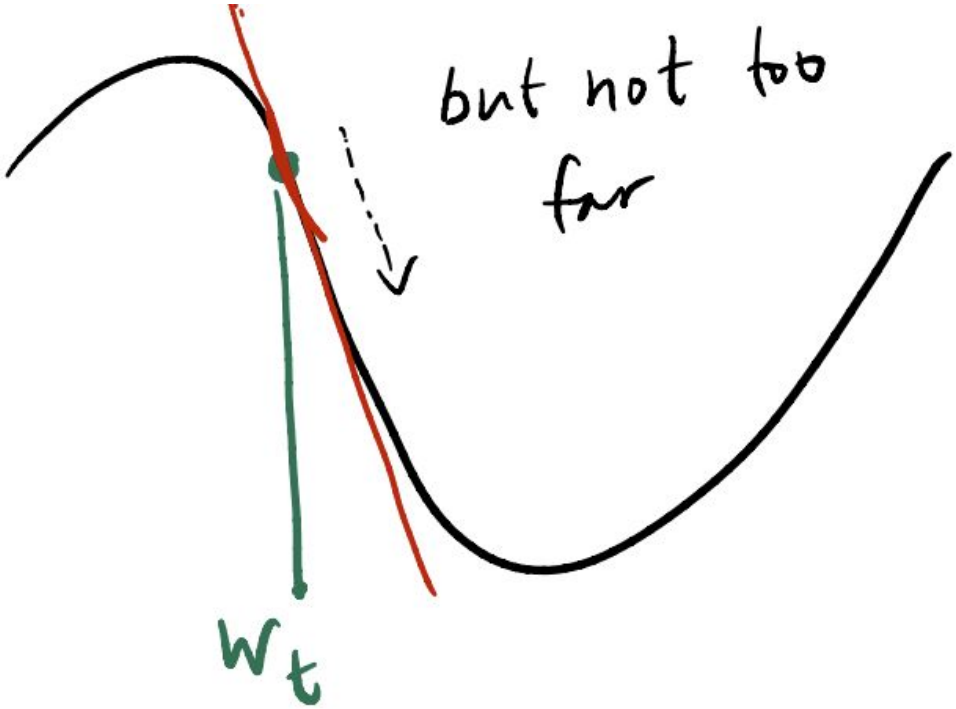


We don’t like second order derivatives

Let’s consider a guess: **step size**

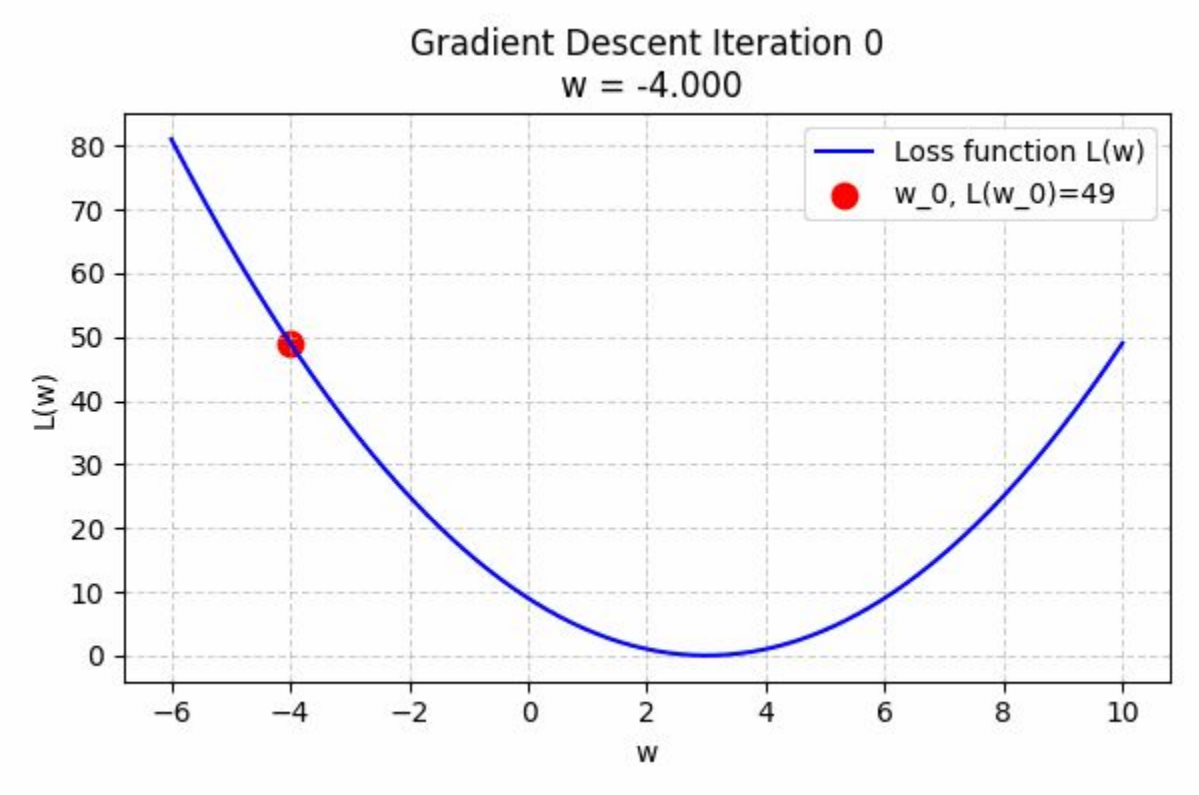
$$\eta_t \approx 1/\mathcal{L}''(\mathcal{D}, w_t), \quad w_{t+1} = w_t - \eta_t \mathcal{L}'(\mathcal{D}, w_t)$$

Gradient Descent (GD)

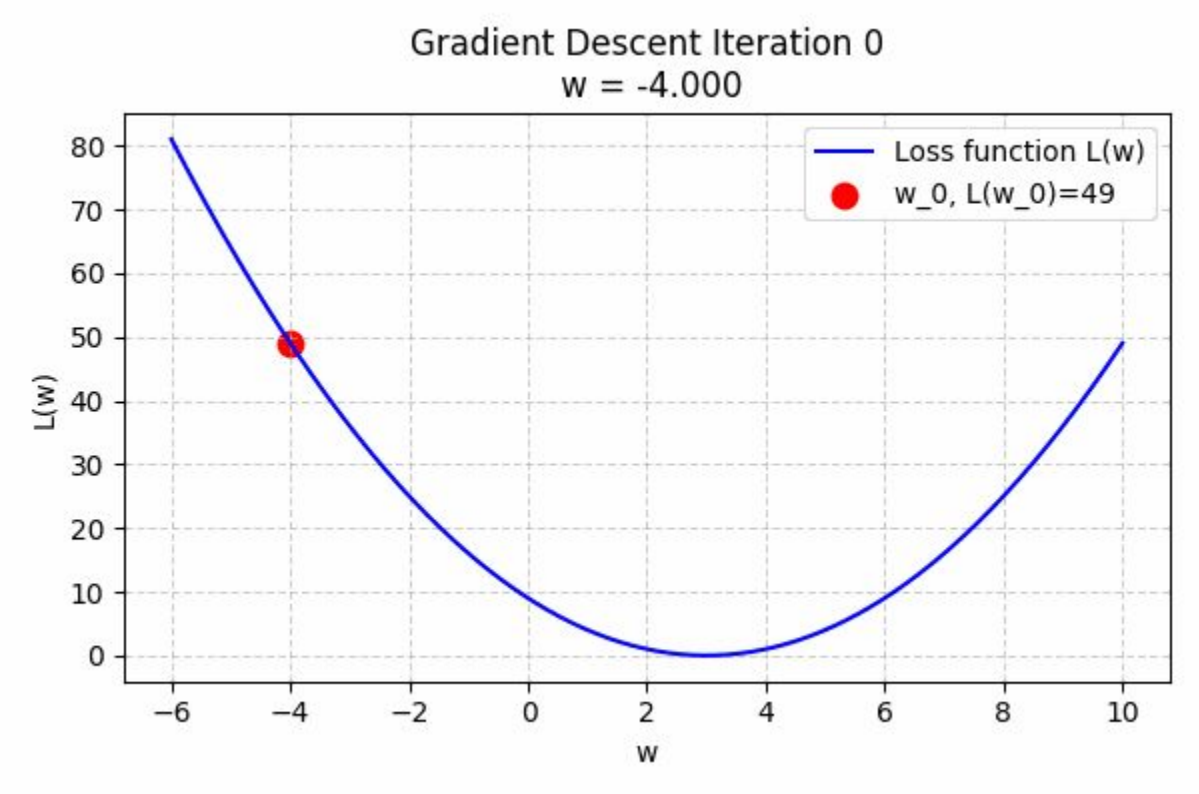


$$w_{t+1} = w_t - \eta_t \mathcal{L}'(\mathcal{D}, w_t)$$

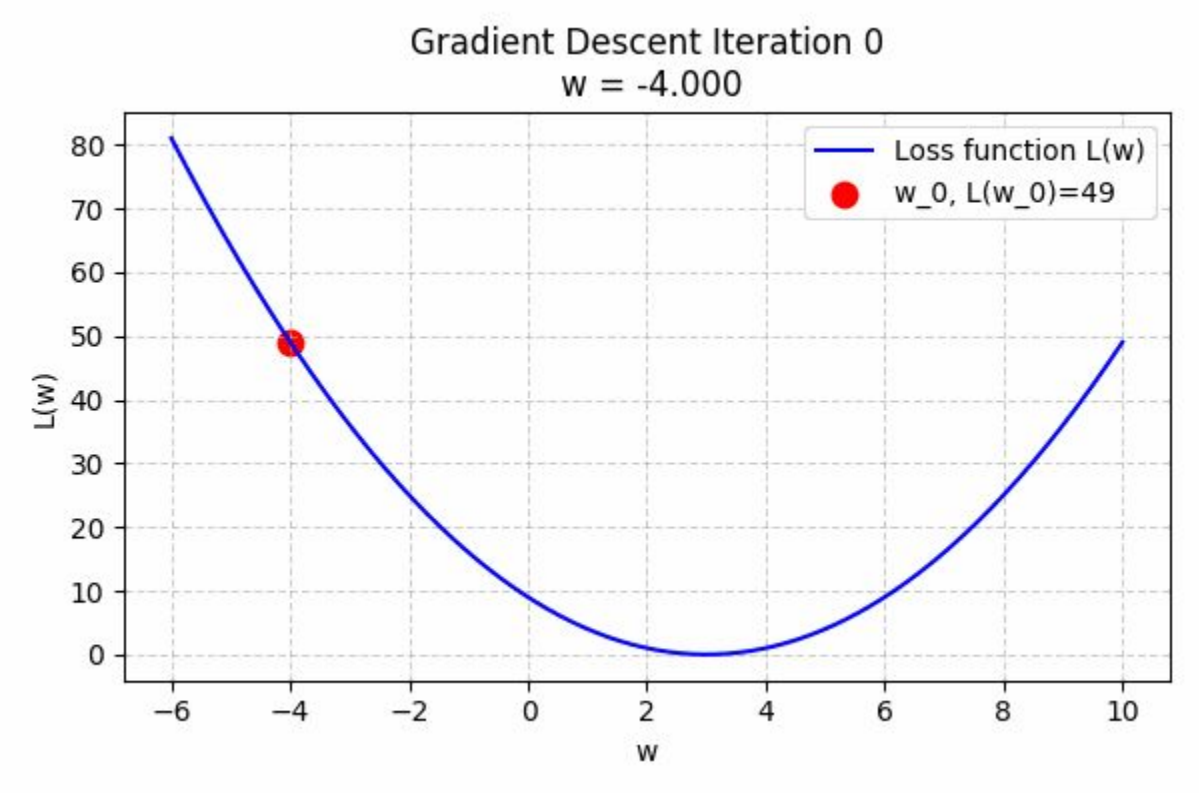
Gradient Descent (GD)



Gradient Descent (GD)



Gradient Descent (GD)



Gradient Descent (GD), vector case

For Neural networks, we have

$$\mathcal{W} \in \mathbb{R}^D, \quad D \gg d$$

$$\mathcal{L}(\mathcal{D}, \mathcal{W}) = \frac{1}{N} \sum_{i=1}^{N} l(x_i, y_i, \mathcal{W})$$

Here we consider the gradient (one derivative per parameter):

Gradient Descent (GD), vector case

For Neural networks, we have

$$\mathcal{W} \in \mathbb{R}^D, \quad D \gg d$$

$$\mathcal{L}(\mathcal{D}, \mathcal{W}) = \frac{1}{N} \sum_{i=1}^{N} l(x_i, y_i, \mathcal{W})$$

Here we consider the gradient (one derivative per parameter):

$$\nabla_{\mathcal{W}} \mathcal{L}(\mathcal{D}, \mathcal{W}) = \begin{bmatrix} \frac{\partial \mathcal{L}(\mathcal{D}, \mathcal{W})}{w_{1,1}^{(1)}} \\ \frac{\partial \mathcal{L}(\mathcal{D}, \mathcal{W})}{w_{1,2}^{(1)}} \\ \vdots \\ \frac{\partial \mathcal{L}(\mathcal{D}, \mathcal{W})}{w_{1,1}^{(L)}} \\ \vdots \\ \frac{\partial \mathcal{L}(\mathcal{D}, \mathcal{W})}{b^{(L)}} \end{bmatrix} \in \mathbb{R}^D$$

$$\mathcal{W}_t = \mathcal{W}_{t-1} - \eta_t \nabla_{\mathcal{W}} \mathcal{L}(\mathcal{D}, \mathcal{W}) \Big|_{\mathcal{W}=\mathcal{W}_{t-1}}$$

NEXT: how to compute the gradient of these “beasts” and other cool stuffs