ECE 1508S2: Applied Deep Learning

Chapter 1: Preliminaries

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Winter 2024

Deep Learning More than Ever

Deep learning is turning into an *unbeatable champion*

- It has achieved accuracy levels in image classification, restoration, and segmentation that surpass human capabilities
- It can create human-like conversational dialogues and respond to scientific and technical questions
- It can beat world-class chess players
- It has solved problems that had not been effectively addressed for a long time

Deep Learning More than Ever

Deep learning is turning into an *unbeatable champion*

- It has achieved accuracy levels in image classification, restoration, and segmentation that surpass human capabilities
- It can create human-like conversational dialogues and respond to scientific and technical questions
- It can beat world-class chess players
- It has solved problems that had not been effectively addressed for a long time

We all have items to add to this list!

The Unbeatable: *AlexNet* $\rightsquigarrow$ *ResNet*

Image recognition has been one of the first problems in AI

we want the machine to recognize what in an image is

The Unbeatable: *AlexNet* ⇝ *ResNet*

Image recognition has been one of the first problems in AI

we want the machine to recognize what in an image is

- Yann LeCun proposed LeNet in 1998
- In 2010, ImageNet started the annual contest

  ImageNet Large Scale Visual Recognition Challenge

  ↳ AlexNet (2012) ⇝ VGG and GoogLeNet (2014) ⇝ ResNet (2015)
- Currently, we have much *deeper* and advanced neural networks

We will get what these networks are and do in this course

The Unbeatable: *AlexNet* ⇝ *ResNet*

In February 2015, first Microsoft and then Google announced that

their deep neural networks beat human accuracy in image recognition

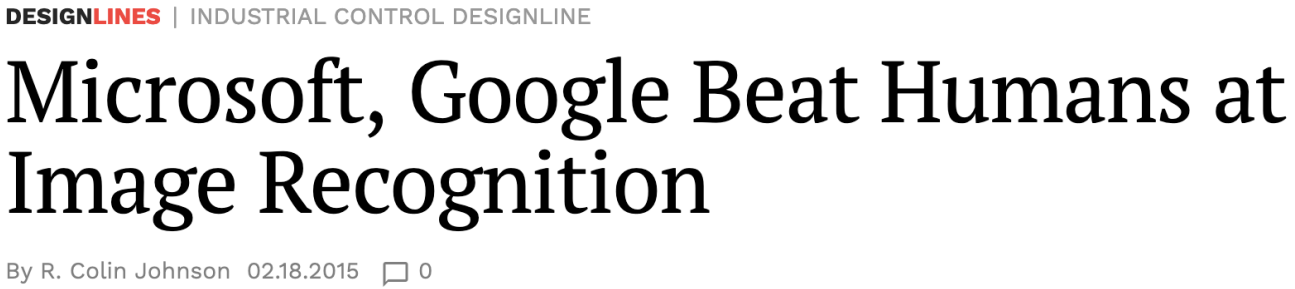
FORBES > TECH

Microsoft's Deep Learning Project Outperforms Humans In Image Recognition

Michael Thomsen Former Contributor ⓘ

I write about tech, video games, science and culture.

Feb 19, 2015, 01:06pm EST



DESIGNLINES | INDUSTRIAL CONTROL DESIGNLINE

Microsoft, Google Beat Humans at Image Recognition

By R. Colin Johnson 02.18.2015 0

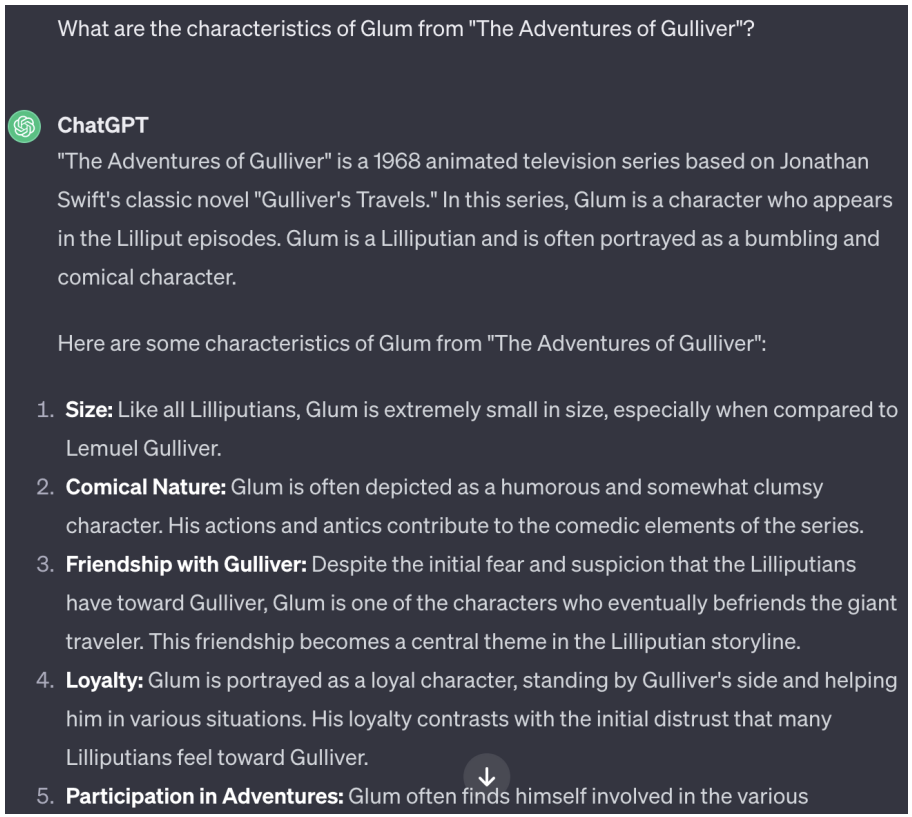
The Unbeatable: *ChatGPT*

Making a human-like dialogues had been another fundamental problem in AI

The Unbeatable: *ChatGPT*

Making a human-like dialogues had been another fundamental problem in AI

↳ ChatGPT broke the record of fastest-growing software in history



The Unbeatable: *AlphaZero*

Computers playing chess: a long-going goal in AI

- *1951:* Alan Turing designed a program that plays chess

The Unbeatable: *AlphaZero*

Computers playing chess: a long-going goal in AI

- *1951:* Alan Turing designed a program that plays chess
- *1989:* Gary Kasparov defeated IBM's Deep Thought
- *1996:* Gary Kasparov defeated IBM's Deep Blue

The Unbeatable: *AlphaZero*

Computers playing chess: a long-going goal in AI

- *1951:* Alan Turing designed a program that plays chess
- *1989:* Gary Kasparov defeated IBM's Deep Thought
- *1996:* Gary Kasparov defeated IBM's Deep Blue
- *1997:* IBM's Deep Blue defeated Gary Kasparov
- *2005:* Ruslan Ponomariov was the last person beating AI chess player

The Unbeatable: *AlphaZero*

Computers playing chess: *a long-going goal in AI*

- *1951*: Alan Turing designed a program that plays chess
- *1989*: Gary Kasparov defeated IBM's Deep Thought
- *1996*: Gary Kasparov defeated IBM's Deep Blue
- *1997*: IBM's Deep Blue defeated Gary Kasparov
- *2005*: Ruslan Ponomariov was the last person beating AI chess player
- *2017*: DeepMind released AlphaZero algorithm
  - ↳ AlphaGo ⇝ AlphaGo Zero ⇝ AlphaZero
  - ↳ After 24 hours of training it defeated Stockfish chess engine
- Even newer AI chess players, e.g., *Leela Chess Zero (2019)*

The Unbeatable: *Neural Machine Translation*

Most translator machines were unreliable until AI kicked in!

- Computer-based translation is a research topic since 1950s

The Unbeatable: *Neural Machine Translation*

Most translator machines were unreliable until AI kicked in!

- Computer-based translation is a research topic since 1950s
- In 2000s, statistical machine translation (SMT) became dominant

The Unbeatable: *Neural Machine Translation*

Most translator machines were unreliable until AI kicked in!

- Computer-based translation is a research topic since 1950s
- In 2000s, statistical machine translation (SMT) became dominant
- In 2010s, neural machine translation (NMT) showed great potentials
  - ↳ *Baidu launched first large scale NMT in 2015*
  - ↳ *Google introduced its NMT called GNMT in 2016*
  - ↳ *DeepL was launched in 2017*

The Unbeatable: *Neural Machine Translation*

Most translator machines were unreliable until AI kicked in!

- Computer-based translation is a research topic since 1950s
- In 2000s, statistical machine translation (SMT) became dominant
- In 2010s, neural machine translation (NMT) showed great potentials
  - ↳ *Baidu launched first large scale NMT in 2015*
  - ↳ *Google introduced its NMT called GNMT in 2016*
  - ↳ *DeepL was launched in 2017*
- *And again: lots of NMT nowadays are getting developed!*

Notion of Deep Learning

Glum starts this conversation

Notion of Deep Learning

Glum starts this conversation

+ *Well! Nice examples! But, what exactly is this Deep Learning?*
- Deep Learning refers to the subset of Machine Learning that uses Deep Neural Networks to execute a given learning task

Notion of Deep Learning

Glum starts this conversation

+ *Well! Nice examples! But, what exactly is this Deep Learning?*
- Deep Learning refers to the subset of Machine Learning that uses Deep Neural Networks to execute a given learning task
+ *Wait a moment! The only thing that I understood is that Deep Learning is a form of Machine Learning! But,*
  - *what is Machine Learning itself?*
  - *what are Neural Networks in the first place? Before we talk about their "deep" version*
  - *what do you mean by a "learning task"?*

Notion of Deep Learning

Glum starts this conversation

+ *Well! Nice examples! But, what exactly is this Deep Learning?*
- Deep Learning refers to the subset of Machine Learning that uses Deep Neural Networks to execute a given learning task
+ *Wait a moment! The only thing that I understood is that Deep Learning is a form of Machine Learning! But,*
  - *what is Machine Learning itself?*
  - *what are Neural Networks in the first place? Before we talk about their "deep" version*
  - *what do you mean by a "learning task"?*
- We are going to understand all of them!
  - First, we start with the concept of Machine Learning

Machine Learning (ML)

Goodfellow et al. *informally* define ML as *". . . a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions. . ."*

Machine Learning (ML)

Goodfellow et al. *informally* define ML as *". . . a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions. . ."*

Though good, this definition is still unclear! Let's put it in simple words

For most problems, there are two approaches to get to a solution

- through analytic derivation
- using data-driven algorithms

ML develops efficient data-driven algorithms for *complicated problems* whose analysis is either *infeasible* or *too complicated*

Machine Learning (ML)

Goodfellow et al. *informally* define ML as *". . . a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions. . ."*

Though good, this definition is still unclear! Let's put it in simple words

For most problems, there are two approaches to get to a solution

- through analytic derivation
- using data-driven algorithms

ML develops efficient data-driven algorithms for *complicated problems* whose analysis is either *infeasible* or *too complicated*

Let's try a dummy example!

Motion of a Projectile

A projectile, located at height h, is initiated by horizontal velocity v. We are asked to determine horizontal distance d at which *it hits the ground*

Motion of a Projectile

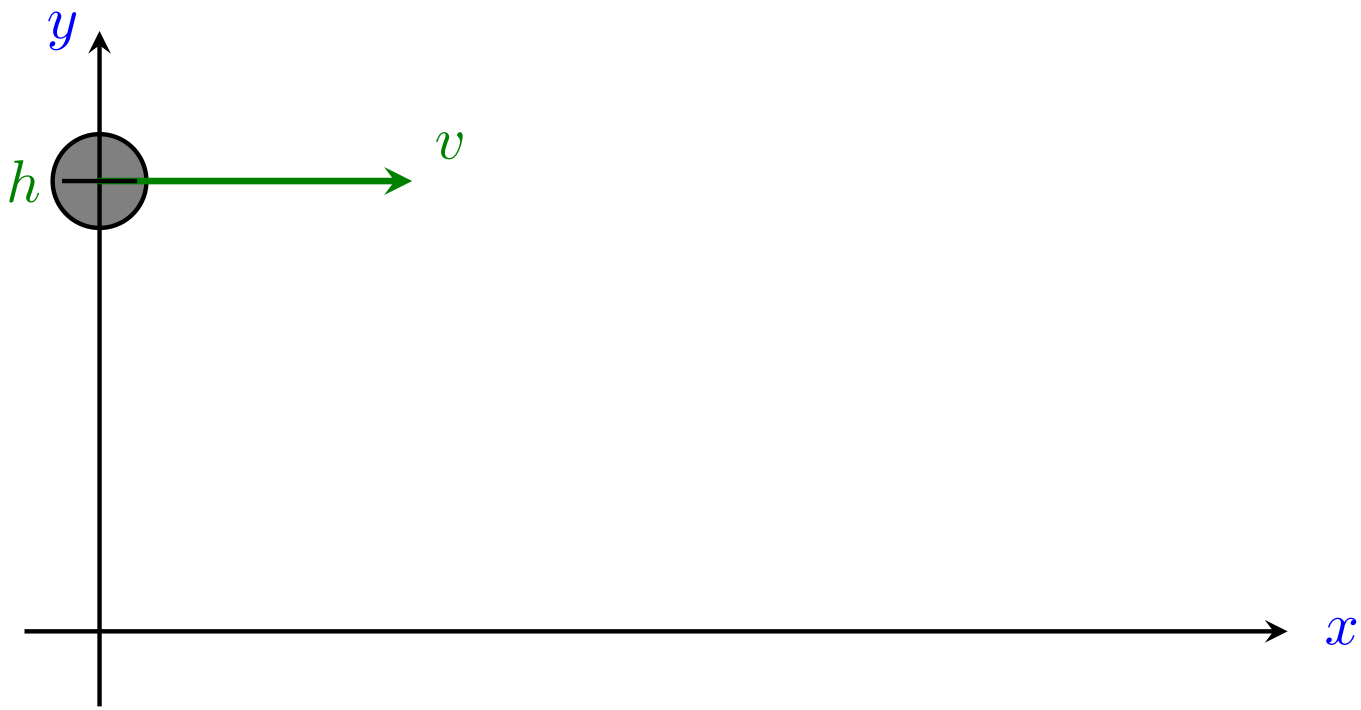
A projectile, located at height h, is initiated by horizontal velocity v. We are asked to determine horizontal distance d at which *it hits the ground*

Motion of a Projectile

A projectile, located at height h, is initiated by horizontal velocity v. We are asked to determine horizontal distance d at which *it hits the ground*

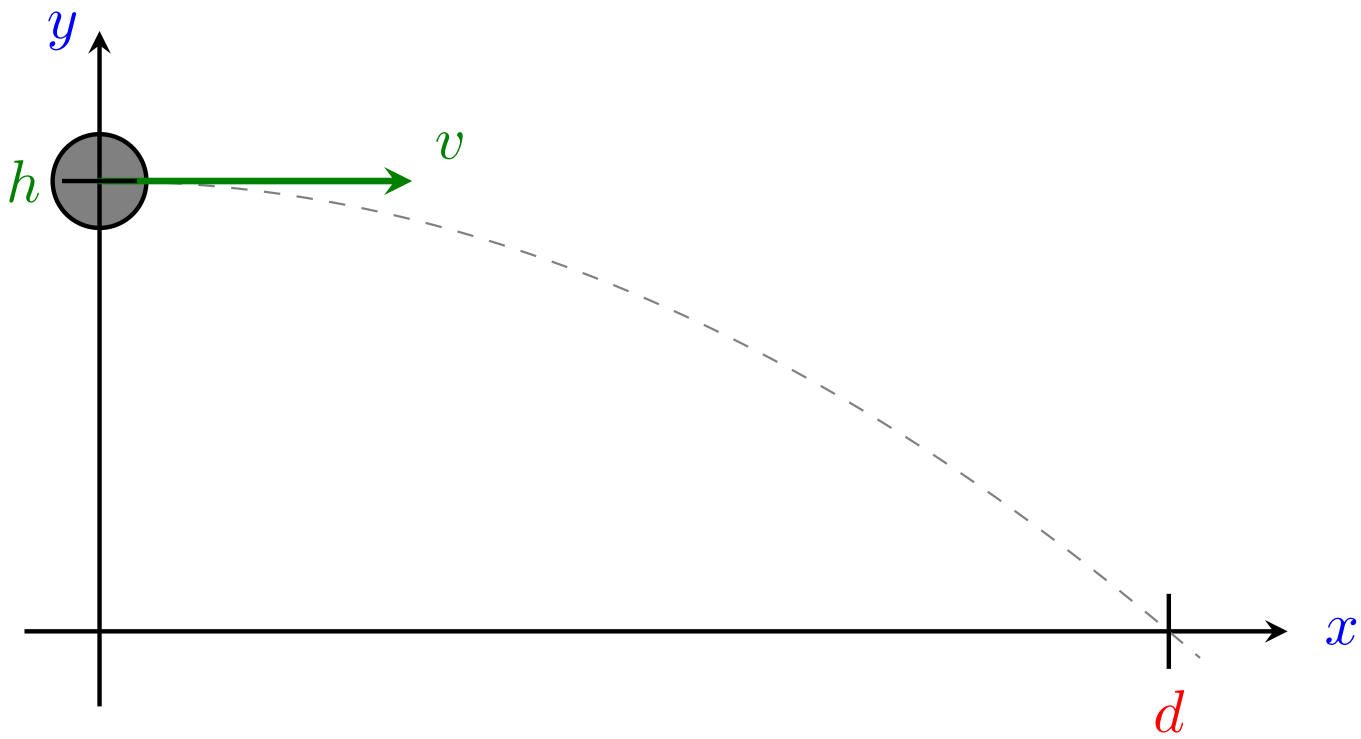
Motion of a Projectile

A projectile, located at height h, is initiated by horizontal velocity v. We are asked to determine horizontal distance d at which *it hits the ground*



Motion of a Projectile

A projectile, located at height h, is initiated by horizontal velocity v. We are asked to determine horizontal distance d at which *it hits the ground*



Motion of a Projectile: *Analytic Solution*

You may remember the analytic solution from high school or general physics

At time t, the location of the projectile (x, y) is

$$x = vt \qquad\qquad y = -\frac{g}{2}t^2 + h$$

with $g \approx 9.8$ m/s^2 being the gravitational acceleration.

Motion of a Projectile: *Analytic Solution*

You may remember the analytic solution from high school or general physics

At time t, the location of the projectile (x, y) is

$$x = vt \qquad\qquad y = -\frac{g}{2}t^2 + h$$

with $g \approx 9.8$ m/s^2 being the gravitational acceleration. When it hits the ground, we have $y = 0$; thus, we can find the hitting time as

$$t_0 = \sqrt{\frac{2h}{g}}$$

Motion of a Projectile: *Analytic Solution*

You may remember the analytic solution from high school or general physics

At time t, the location of the projectile (x, y) is

$$x = vt \qquad\qquad y = -\frac{g}{2}t^2 + h$$

with $g \approx 9.8$ m/s^2 being the gravitational acceleration. When it hits the ground, we have $y = 0$; thus, we can find the hitting time as

$$t_0 = \sqrt{\frac{2h}{g}}$$

At $t = t_0$, we have $x = d = vt_0$. Defining $\gamma = \sqrt{2/g} \approx 0.45$, we have

$$d = \gamma v \sqrt{h}$$

Motion of a Projectile: *Analytic Solution*

The projectile hits the ground at the horizontal distance

$$d = \gamma v \sqrt{h}$$

Motion of a Projectile: *Analytic Solution*

The projectile hits the ground at the horizontal distance

$$d = \gamma v \sqrt{h}$$

To derive this analytic result, we have used two facts

- We knew Newton's laws that describe the motion
- We could solve the equations for d analytically

Motion of a Projectile: *Analytic Solution*

The projectile hits the ground at the horizontal distance

$$d = \gamma v \sqrt{h}$$

To derive this analytic result, we have used two facts

- We knew Newton's laws that describe the motion
- We could solve the equations for d analytically

Well! This is not the case in all problems! In fact, there might be

- no analytic law known that describes the relations, or
- a complicated law whose analysis is computationally infeasible

Motion of a Projectile: *Analytic Solution*

The projectile hits the ground at the horizontal distance

$$d = \gamma v \sqrt{h}$$

To derive this analytic result, we have used two facts

- We knew Newton's laws that describe the motion
- We could solve the equations for d analytically

Well! This is not the case in all problems! In fact, there might be

- no analytic law known that describes the relations, or
- a complicated law whose analysis is computationally infeasible

Let's now look at the ML approach!

Motion of a Projectile: *ML Approach*

A computer scientist could well have forgotten general physics

We conduct the experiment for I different times: in try $i = 1, \ldots, I$

- we initiate with different velocity v_i and height h_i
- we measure the horizontal distance d_i

Motion of a Projectile: *ML Approach*

A computer scientist could well have forgotten general physics

We conduct the experiment for I different times: in try $i = 1, \ldots, I$

- we initiate with different velocity v_i and height h_i
- we measure the horizontal distance d_i

We then assume that d_i and $[v_i, h_i]$ are related via a pre-defined function;

Motion of a Projectile: *ML Approach*

A computer scientist could well have forgotten general physics

We conduct the experiment for I different times: in try $i = 1, \ldots, I$

- we initiate with different velocity v_i and height h_i
- we measure the horizontal distance d_i

We then assume that d_i and $[v_i, h_i]$ are related via a pre-defined function; for instance, we assume d_i and $[v_i, h_i]$ are related as

$$d_i = w_0 v_i + w_1 h_i$$

Motion of a Projectile: *ML Approach*

A computer scientist could well have forgotten general physics

We conduct the experiment for I different times: in try $i = 1, \ldots, I$

- we initiate with different velocity v_i and height h_i
- we measure the horizontal distance d_i

We then assume that d_i and $[v_i, h_i]$ are related via a pre-defined function; for instance, we assume d_i and $[v_i, h_i]$ are related as

$$d_i = w_0 v_i + w_1 h_i$$

We then try to find the values $w_0 = w_0^\star$ and $w_1 = w_1^\star$, such that this pre-defined function closely matches our experimental results

Motion of a Projectile: *ML Approach*

How can we find $w_0^\star$ and $w_1^\star$?

The function should match our experimental data, i.e., for any i

$$d_i \overset{!}{=} w_0^\star v_i + w_1^\star h_i \rightsquigarrow (d_i - w_0^\star v_i - w_1^\star h_i)^2 \overset{!}{=} 0$$

By $\overset{!}{=}$, we mean that we intend to have this identity holding

Motion of a Projectile: *ML Approach*

How can we find $w_0^\star$ and $w_1^\star$?

The function should match our experimental data, i.e., for any i

$$d_i \overset{!}{=} w_0^\star v_i + w_1^\star h_i \rightsquigarrow (d_i - w_0^\star v_i - w_1^\star h_i)^2 \overset{!}{=} 0$$

By $\overset{!}{=}$, we mean that we intend to have this identity holding

The last identity is equivalent to write

$$\sum_{i=1}^{I} (d_i - w_0^\star v_i - w_1^\star h_i)^2 \overset{!}{=} 0$$

Motion of a Projectile: *ML Approach*

How can we find $w_0^\star$ and $w_1^\star$?

The function should match our experimental data, i.e., for any i

$$d_i \overset{!}{=} w_0^\star v_i + w_1^\star h_i \rightsquigarrow (d_i - w_0^\star v_i - w_1^\star h_i)^2 \overset{!}{=} 0$$

By $\overset{!}{=}$, we mean that we intend to have this identity holding

The last identity is equivalent to write

$$\sum_{i=1}^{I} (d_i - w_0^\star v_i - w_1^\star h_i)^2 \overset{!}{=} 0$$

But, such $w_0^\star$ and $w_1^\star$ do not necessarily exist if $I \geqslant 3$!

Don't worry if you don't see it right away! You'll see it in an assignment!

Motion of a Projectile: *ML Approach*

How can we find $w_0^\star$ *and* $w_1^\star$*?*

Motion of a Projectile: *ML Approach*

How can we find $w_0^\star$ *and* $w_1^\star$*?* We find $w_0^\star$ and $w_1^\star$ such that

$$\sum_{i=1}^{I} \left(d_i - w_0^\star v_i - w_1^\star h_i\right)^2$$

is as small as possible

Motion of a Projectile: *ML Approach*

How can we find $w_0^\star$ *and* $w_1^\star$*?* We find $w_0^\star$ and $w_1^\star$ such that

$$\sum_{i=1}^{I} \left(d_i - w_0^\star v_i - w_1^\star h_i\right)^2$$

is as small as possible

Let us define the loss function $\mathcal{L}\left(w_0, w_1\right)$ as

$$\mathcal{L}\left(w_0, w_1\right) = \sum_{i=1}^{I} \left(d_i - w_0 v_i - w_1 h_i\right)^2$$

We then find $w_0^\star$ and $w_1^\star$ that minimize the loss

$$\left(w_0^\star, w_1^\star\right) = \underset{w_0, w_1}{\operatorname{argmin}} \mathcal{L}\left(w_0, w_1\right)$$

Motion of a Projectile: *ML Approach*

For new given v and h, we find the horizontal distance as

$$d = w_0^\star v + w_1^\star h$$

Motion of a Projectile: *ML Approach*

For new given v and h, we find the horizontal distance as

$$d = w_0^\star v + w_1^\star h$$

We could intuitively say that

- The ML-derived distance is not as accurate as the analytic one
  - The assumed relation between d and $[v, h]$ is not exact
- The ML approach gets better as we increase the number of trial I

Motion of a Projectile: *ML Approach*

For new given v and h, we find the horizontal distance as

$$d = w_0^\star v + w_1^\star h$$

We could intuitively say that

- The ML-derived distance is not as accurate as the analytic one
  - The assumed relation between d and $[v, h]$ is not exact
- The ML approach gets better as we increase the number of trial I

In the first assignment, we will program this dummy example!

Motion of a Projectile: *ML Approach*

The ML approach has three main components:

Motion of a Projectile: *ML Approach*

The ML approach has three main components:

1. *Dataset:* For our example, we collected dataset $\mathbb{D}$

$$\mathbb{D} = \{([v_i, h_i], d_i) : i = 1, \ldots, I\}$$

Motion of a Projectile: *ML Approach*

The ML approach has three main components:

1. *Dataset:* For our example, we collected dataset $\mathbb{D}$

$$\mathbb{D} = \{([v_i, h_i], d_i) : i = 1, \ldots, I\}$$

2. *Model:* We assumed d and $[v, h]$ are related via a *linear* model

$$d = w_0 v + w_1 h$$

Motion of a Projectile: *ML Approach*

The ML approach has three main components:

1. *Dataset:* For our example, we collected dataset $\mathbb{D}$

$$\mathbb{D} = \{([v_i, h_i], d_i) : i = 1, \ldots, I\}$$

2. *Model:* We assumed d and $[v, h]$ are related via a *linear* model

$$d = w_0 v + w_1 h$$

3. *Loss:* We evaluated the loss of our model for given w_0 and w_1 as

$$\mathcal{L}(w_0, w_1) = \sum_{i=1}^{I} (d_i - w_0 v_i + w_1 h_i)^2$$

We now take a deeper look into each component

Recap: Vectors and Matrices

We're going to use frequently linear algebra! So, let's recall some basics

$\boldsymbol{x} \in \mathbb{R}^N$ is an N-dimensional *column*-vector with N *real* entries, i.e.,

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \rightsquigarrow \boldsymbol{x}^\mathsf{T} = \begin{bmatrix} x_1, \ldots, x_N \end{bmatrix}$$

If we want to make a a *row*-vector, we transpose it, i.e., use $\boldsymbol{x}^\mathsf{T}$

Recap: Vectors and Matrices

We're going to use frequently linear algebra! So, let's recall some basics

$\boldsymbol{x} \in \mathbb{R}^N$ is an N-dimensional *column*-vector with N *real* entries, i.e.,

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \rightsquigarrow \boldsymbol{x}^\mathsf{T} = \begin{bmatrix} x_1, \ldots, x_N \end{bmatrix}$$

If we want to make a a *row*-vector, we transpose it, i.e., use $\boldsymbol{x}^\mathsf{T}$

Notation

We show vectors with **bold-face** small letters and drop *column*/*row*

- A vector is by default a *column*-vector
- If we need a *row*-vector, we transpose its *column* version

Recap: Vectors and Matrices

Matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$ can be seen as

- either as the *collection of M column-vectors of dimension N*

$$\mathbf{A} = \left[\mathbf{a}_1, \ldots, \mathbf{a}_M\right]$$

with $\mathbf{a}_m \in \mathbb{R}^N$ for $m = 1, \ldots, M$

Recap: Vectors and Matrices

Matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$ can be seen as

- either as the *collection of M column-vectors of dimension N*

$$\mathbf{A} = \left[\mathbf{a}_1, \ldots, \mathbf{a}_M\right]$$

with $\mathbf{a}_m \in \mathbb{R}^N$ for $m = 1, \ldots, M$

- or as the *collection of N row-vectors of dimension M*

$$\mathbf{A} = \begin{bmatrix} \mathbf{b}_1^\mathsf{T} \\ \vdots \\ \mathbf{b}_N^\mathsf{T} \end{bmatrix}$$

with $\mathbf{b}_n \in \mathbb{R}^M$ for $n = 1, \ldots, N$

Recap: Vectors and Matrices

Notation

We show matrices with **bold-face** capital letters

Two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^N$ of same dimension N are inner-multiplied as

$$\boldsymbol{x}^\mathsf{T}\boldsymbol{y} = \boldsymbol{y}^\mathsf{T}\boldsymbol{x} = \sum_{n=1}^{N} x_n y_n$$

Recap: Vectors and Matrices

Notation

We show matrices with **bold-face** capital letters

Two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^N$ of same dimension N are inner-multiplied as

$$\boldsymbol{x}^\mathsf{T}\boldsymbol{y} = \boldsymbol{y}^\mathsf{T}\boldsymbol{x} = \sum_{n=1}^{N} x_n y_n$$

They can further outer-multiplied as

$$\boldsymbol{x}\boldsymbol{y}^\mathsf{T} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \begin{bmatrix} y_1, \ldots, y_N \end{bmatrix} = \begin{bmatrix} x_1 y_1 & \ldots & x_1 y_N \\ \vdots & \ldots & \vdots \\ x_N y_1 & \ldots & x_N y_N \end{bmatrix}$$

Recap: Vectors and Matrices

Notation

We show matrices with **bold-face** capital letters

Two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^N$ of same dimension N are inner-multiplied as

$$\boldsymbol{x}^\mathsf{T}\boldsymbol{y} = \boldsymbol{y}^\mathsf{T}\boldsymbol{x} = \sum_{n=1}^{N} x_n y_n$$

They can further outer-multiplied as

$$\boldsymbol{x}\boldsymbol{y}^\mathsf{T} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \begin{bmatrix} y_1, \ldots, y_N \end{bmatrix} = \begin{bmatrix} x_1 y_1 & \ldots & x_1 y_N \\ \vdots & \ldots & \vdots \\ x_N y_1 & \ldots & x_N y_N \end{bmatrix} = \left(\boldsymbol{y}^\mathsf{T}\boldsymbol{x}\right)^\mathsf{T}$$

Recap: Vectors and Matrices

Multiplying matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$ with $\boldsymbol{x} \in \mathbb{R}^{M}$ can be seen as

- either as the *linear combination of column-vectors in* $\mathbf{A}$

$$\mathbf{A}\boldsymbol{x} = \left[\mathbf{a}_1, \ldots, \mathbf{a}_M\right] \begin{bmatrix} x_1 \\ \vdots \\ x_M \end{bmatrix} = \sum_{m=1}^{M} x_m \mathbf{a}_m \in \mathbb{R}^N$$

Recap: Vectors and Matrices

Multiplying matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$ with $\boldsymbol{x} \in \mathbb{R}^{M}$ can be seen as

- either as the *linear combination of column-vectors in* $\mathbf{A}$

$$\mathbf{A}\boldsymbol{x} = \begin{bmatrix} \mathbf{a}_1, \ldots, \mathbf{a}_M \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_M \end{bmatrix} = \sum_{m=1}^{M} x_m \mathbf{a}_m \in \mathbb{R}^N$$

- or as the *collection of inner-products of row-vectors with* $\boldsymbol{x}$

$$\mathbf{A}\boldsymbol{x} = \begin{bmatrix} \bar{\mathbf{a}}_1^\mathsf{T} \\ \vdots \\ \bar{\mathbf{a}}_N^\mathsf{T} \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} \bar{\mathbf{a}}_1^\mathsf{T} \boldsymbol{x} \\ \vdots \\ \bar{\mathbf{a}}_N^\mathsf{T} \boldsymbol{x} \end{bmatrix} \in \mathbb{R}^N$$

ML Components: *Dataset*

Dataset is the collection of data-points; in our example, the dataset was

$$\mathbb{D} = \{([v_i, h_i], d_i) : i = 1, \ldots, I\}$$

which has I data-points $([v_i, h_i], d_i)$ for $i = 1, \ldots, I$

ML Components: *Dataset*

Dataset is the collection of data-points; in our example, the dataset was

$$\mathbb{D} = \{([v_i, h_i], d_i) : i = 1, \ldots, I\}$$

which has I data-points $([v_i, h_i], d_i)$ for $i = 1, \ldots, I$

This is an example of a labeled dataset: a dataset whose data-points contain both the inputs and their corresponding labels (outputs)

ML Components: *Dataset*

Dataset is the collection of data-points; in our example, the dataset was

$$\mathbb{D} = \{([v_i, h_i], d_i) : i = 1, \ldots, I\}$$

which has I data-points $([v_i, h_i], d_i)$ for $i = 1, \ldots, I$

> This is an example of a labeled dataset: a dataset whose data-points contain both the inputs and their corresponding labels (outputs)

More general, a labeled dataset with I data-points is

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

- $\boldsymbol{x}_i \in \mathbb{R}^N$ is an input vector with N entries
- y_i is the label of $\boldsymbol{x}_i$ that is a scalar

ML Components: *Supervised Learning*

When the dataset is labeled, our *task* is clear:

- We observe I inputs to a function with their outputs (labels)
- We try to find out (learn) this function

ML Components: *Supervised Learning*

When the dataset is labeled, our *task* is clear:

- We observe I inputs to a function with their outputs (labels)
- We try to find out (learn) this function

Our task is a *learning task*, because we are trying to

learn the unknown relation between inputs and labels

ML Components: *Supervised Learning*

When the dataset is labeled, our *task* is clear:

- We observe I inputs to a function with their outputs (labels)
- We try to find out (learn) this function

Our task is a *learning task*, because we are trying to

learn the unknown relation between inputs and labels

It describes a *supervised learning problem*, since

a supervisor has filled our dataset with labels

In other words, for the sample inputs in our hand, we know the outputs

ML Components: *Supervised Learning*

When the dataset is labeled, our *task* is clear:

- We observe I inputs to a function with their outputs (labels)
- We try to find out (learn) this function

Our task is a *learning task*, because we are trying to

learn the unknown relation between inputs and labels

It describes a *supervised learning problem*, since

a supervisor has filled our dataset with labels

In other words, for the sample inputs in our hand, we know the outputs

In this course, we are mainly focused on supervised learning

ML Components: *Unsupervised Learning*

+ *But can a dataset be unlabeled?*
– Yes! This is the case in *unsupervised learning* in which we are to learn features of a data-point x by investigating a set of its samples

ML Components: *Unsupervised Learning*

+ *But can a dataset be unlabeled?*
- Yes! This is the case in *unsupervised learning* in which we are to learn features of a data-point $\boldsymbol{x}$ by investigating a set of its samples

In this case, the dataset is of the form

$$\mathbb{D} = \{\boldsymbol{x}_i : i = 1, \ldots, I\}$$

which is unlabeled. Our *learning task* is further *unsupervised*

ML Components: *Unsupervised Learning*

+ *But can a dataset be unlabeled?*
- Yes! This is the case in *unsupervised learning* in which we are to learn features of a data-point $\boldsymbol{x}$ by investigating a set of its samples

In this case, the dataset is of the form

$$\mathbb{D} = \{\boldsymbol{x}_i : i = 1, \ldots, I\}$$

which is unlabeled. Our *learning task* is further *unsupervised*

Let's make it crystal clear via an example!

ML Components: *Example of Unsupervised Learning*

An N-dimensional data-point $\boldsymbol{x}$ is coming from a natural process, e.g., it contains the pixel values of an image taken from body.

ML Components: *Example of Unsupervised Learning*

An N-dimensional data-point $\boldsymbol{x}$ is coming from a natural process, e.g., it contains the pixel values of an image taken from body. It is known that $\boldsymbol{x}$ is a linear combination of $Q \ll N$ *principal vectors*:

ML Components: *Example of Unsupervised Learning*

An N-dimensional data-point $\boldsymbol{x}$ is coming from a natural process, e.g., it contains the pixel values of an image taken from body. It is known that $\boldsymbol{x}$ is a linear combination of $Q \ll N$ *principal vectors*: any $\boldsymbol{x}$ is

$$\boldsymbol{x} = \sum_{q=1}^{Q} a_q \boldsymbol{v}_q$$

ML Components: *Example of Unsupervised Learning*

An N-dimensional data-point $\boldsymbol{x}$ is coming from a natural process, e.g., it contains the pixel values of an image taken from body. It is known that $\boldsymbol{x}$ is a linear combination of $Q \ll N$ *principal vectors*: any $\boldsymbol{x}$ is

$$\boldsymbol{x} = \sum_{q=1}^{Q} a_q \boldsymbol{v}_q = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_Q] \begin{bmatrix} a_1 \\ \vdots \\ a_Q \end{bmatrix}$$

ML Components: *Example of Unsupervised Learning*

An N-dimensional data-point $\boldsymbol{x}$ is coming from a natural process, e.g., it contains the pixel values of an image taken from body. It is known that $\boldsymbol{x}$ is a linear combination of $Q \ll N$ *principal vectors*: any $\boldsymbol{x}$ is

$$\boldsymbol{x} = \sum_{q=1}^{Q} a_q \boldsymbol{v}_q = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_Q] \begin{bmatrix} a_1 \\ \vdots \\ a_Q \end{bmatrix} = \mathbf{V}\mathbf{a}$$

with $\mathbf{a} \in \mathbb{R}^Q$ and $\mathbf{V} \in \mathbb{R}^{N \times Q}$.

ML Components: *Example of Unsupervised Learning*

An N-dimensional data-point $\boldsymbol{x}$ is coming from a natural process, e.g., it contains the pixel values of an image taken from body. It is known that $\boldsymbol{x}$ is a linear combination of $Q \ll N$ *principal vectors*: any $\boldsymbol{x}$ is

$$\boldsymbol{x} = \sum_{q=1}^{Q} a_q \boldsymbol{v}_q = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_Q] \begin{bmatrix} a_1 \\ \vdots \\ a_Q \end{bmatrix} = \mathbf{V}\mathbf{a}$$

with $\mathbf{a} \in \mathbb{R}^Q$ and $\mathbf{V} \in \mathbb{R}^{N \times Q}$. Nevertheless, we do not now the principle vectors, i.e., matrix $\mathbf{V}$ is unknown to us.

ML Components: *Example of Unsupervised Learning*

An N-dimensional data-point $\boldsymbol{x}$ is coming from a natural process, e.g., it contains the pixel values of an image taken from body. It is known that $\boldsymbol{x}$ is a linear combination of $Q \ll N$ *principal vectors*: any $\boldsymbol{x}$ is

$$\boldsymbol{x} = \sum_{q=1}^{Q} a_q \boldsymbol{v}_q = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_Q] \begin{bmatrix} a_1 \\ \vdots \\ a_Q \end{bmatrix} = \mathbf{V}\mathbf{a}$$

with $\mathbf{a} \in \mathbb{R}^Q$ and $\mathbf{V} \in \mathbb{R}^{N \times Q}$. Nevertheless, we do not now the principle vectors, i.e., matrix $\mathbf{V}$ is unknown to us.

Our learning task is to

find out what $\mathbf{V}$ is by investigating I samples of $\boldsymbol{x}$

ML Components: *Example of Unsupervised Learning*

In this problem, our dataset is of the form

$$\mathbb{D} = \{\boldsymbol{x}_i : i = 1, \ldots, I\}$$

which is unlabled.

ML Components: *Example of Unsupervised Learning*

In this problem, our dataset is of the form

$$\mathbb{D} = \{\boldsymbol{x}_i : i = 1, \ldots, I\}$$

which is unlabled. We know that these samples are of the form

$$\boldsymbol{x}_i = \mathbf{V}\mathbf{a}_i$$

but we know *neither* $\mathbf{V}$ *nor* $\mathbf{a}_i$. This is an unsupervised learning problem

ML Components: *Example of Unsupervised Learning*

In this problem, our dataset is of the form

$$\mathbb{D} = \{\boldsymbol{x}_i : i = 1, \ldots, I\}$$

which is unlabled. We know that these samples are of the form

$$\boldsymbol{x}_i = \mathbf{V}\mathbf{a}_i$$

but we know *neither* $\mathbf{V}$ *nor* $\mathbf{a}_i$. This is an unsupervised learning problem

This is the well-known problem of *dimensionality reduction*

- we get a large dimensional vector $\boldsymbol{x}$
- we derive a feature out of it, i.e., $\mathbf{a}$, which is of lower dimention

The classical solution is Principal Component Analysis (PCA)

ML Components: *Unsupervised* $\rightsquigarrow$ *Supervised*

Supervised and unsupervised are not only divisions in terms of dataset:

- We may deal with a semi-supervised learning task
  - Dataset contains both labeled and unlabeled data-points

ML Components: *Unsupervised* $\rightsquigarrow$ *Supervised*

Supervised and unsupervised are not only divisions in terms of dataset:

- We may deal with a semi-supervised learning task
  - Dataset contains both labeled and unlabeled data-points
- We may deal with a reinforcement learning task
  - A client is to learn a set of actions *each relying on the others*
  - Client's dataset grows through interactions with the environment
  - Best example is the design of a machine that *learns to play chess*

ML Components: *Unsupervised* ⇝ *Supervised*

Supervised and unsupervised are not only divisions in terms of dataset:

- We may deal with a semi-supervised learning task
  - Dataset contains both labeled and unlabeled data-points
- We may deal with a reinforcement learning task
  - A client is to learn a set of actions *each relying on the others*
  - Client's dataset grows through interactions with the environment
  - Best example is the design of a machine that *learns to play chess*

As mentioned, the main focus of this course is on supervised learning

We will also discuss unsupervised later in the course

Reinforcement learning is beyond the scope of this course

There will be a separate course on this topic next semester

Stay tuned for that!

ML Components: *Model*

Now, we know what supervised learning and labeled dataset are.

Let's assume we are given by the labeled dataset

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

and are to learn the function that relates an input $\boldsymbol{x}$ to its label y

ML Components: *Model*

Now, we know what supervised learning and labeled dataset are.

Let's assume we are given by the labeled dataset

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

and are to learn the function that relates an input $\boldsymbol{x}$ to its label y

For this task we need to consider a Model

Model is a *parameterized* function that is used to describe the relation between the input $\boldsymbol{x}$ and its label y

ML Components: *Model*

Now, we know what supervised learning and labeled dataset are.

Let's assume we are given by the labeled dataset

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

and are to learn the function that relates an input $\boldsymbol{x}$ to its label y

For this task we need to consider a Model

Model is a *parameterized* function that is used to describe the relation between the input $\boldsymbol{x}$ and its label y

In our dummy example, the model was linear

$$\text{label} = d = w_0 v + w_1 h$$

ML Components: *Model*

Now, we know what supervised learning and labeled dataset are.

Let's assume we are given by the labeled dataset

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

and are to learn the function that relates an input $\boldsymbol{x}$ to its label y

For this task we need to consider a Model

Model is a *parameterized* function that is used to describe the relation between the input $\boldsymbol{x}$ and its label y

In our dummy example, the model was linear

$$\text{label} = d = w_0 v + w_1 h = [w_0, w_1] \begin{bmatrix} v \\ h \end{bmatrix}$$

ML Components: *Model*

Now, we know what supervised learning and labeled dataset are.

Let's assume we are given by the labeled dataset

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

and are to learn the function that relates an input $\boldsymbol{x}$ to its label y

For this task we need to consider a Model

Model is a *parameterized* function that is used to describe the relation between the input $\boldsymbol{x}$ and its label y

In our dummy example, the model was linear

$$\text{label} = d = w_0 v + w_1 h = [w_0, w_1] \begin{bmatrix} v \\ h \end{bmatrix} = \mathbf{w}^\mathsf{T} \boldsymbol{x}$$

Recap: Linear versus Affine Function

Linear Function

Scalar $y \in \mathbb{R}$ is a linear function of vector $\boldsymbol{x} \in \mathbb{R}^N$ if

$$y = \mathbf{w}^\mathsf{T} \boldsymbol{x}$$

for some constant vector $\mathbf{w} \in \mathbb{R}^N$

Recap: Linear versus Affine Function

Linear Function

Scalar $y \in \mathbb{R}$ is a linear function of vector $\boldsymbol{x} \in \mathbb{R}^N$ if

$$y = \mathbf{w}^\mathsf{T} \boldsymbol{x}$$

for some constant vector $\mathbf{w} \in \mathbb{R}^N$

Affine Function

Scalar $y \in \mathbb{R}$ is an affine function of vector $\boldsymbol{x} \in \mathbb{R}^N$ if

$$y = \mathbf{w}^\mathsf{T} \boldsymbol{x} + b$$

for some constant vector $\mathbf{w} \in \mathbb{R}^N$ and scalar $b \neq 0$

Recap: Linear versus Affine Function

Key difference: If $\boldsymbol{x} = \mathbf{0}_N$ is the vector of all zeros

- Linear function returns zero: *it passes through the origin*
- Affine function returns non-zero: *it does not pass through the origin*

Recap: Linear versus Affine Function

Key difference: If $\boldsymbol{x} = \mathbf{0}_N$ is the vector of all zeros

- Linear function returns zero: *it passes through the origin*
- Affine function returns non-zero: *it does not pass through the origin*

We can simply extend the definition to a vector-valued functions

Let $\mathbf{A} \in \mathbb{R}^{M \times N}$ and $\mathbf{b} \in \mathbb{R}^{M}$; then,

$$\boldsymbol{y} = \mathbf{A}\boldsymbol{x}$$

is a linear vector-valued function of $\boldsymbol{x}$ and

$$\boldsymbol{y} = \mathbf{A}\boldsymbol{x} + \mathbf{b}$$

is an affine vector-valued function

ML Components: *Model*

In our dummy example, we considered a linear model, but

- we could have considered an affine model

$$\text{label} = b + \mathbf{w}^\mathsf{T} \boldsymbol{x}$$

ML Components: *Model*

In our dummy example, we considered a linear model, but

- we could have considered an affine model

$$\text{label} = b + \mathbf{w}^\mathsf{T} \boldsymbol{x}$$

- or a polynomial model

$$\text{label} = b + \mathbf{w}_1^\mathsf{T} \boldsymbol{x} + \mathbf{w}_2^\mathsf{T} \boldsymbol{x}^2 + \ldots + \mathbf{w}_P^\mathsf{T} \boldsymbol{x}^P$$

ML Components: *Model*

In our dummy example, we considered a linear model, but

- we could have considered an affine model

$$\text{label} = b + \mathbf{w}^{\mathsf{T}} \boldsymbol{x}$$

- or a polynomial model

$$\text{label} = b + \mathbf{w}_1^{\mathsf{T}} \boldsymbol{x} + \mathbf{w}_2^{\mathsf{T}} \boldsymbol{x}^2 + \ldots + \mathbf{w}_P^{\mathsf{T}} \boldsymbol{x}^P$$

Notation

By $f(\boldsymbol{x})$ we refer to entry-wise function operation

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \rightsquigarrow \boldsymbol{x}^2 = \begin{bmatrix} {x_1}^p \\ \vdots \\ {x_N}^p \end{bmatrix} \text{ or } \sqrt{\boldsymbol{x}} = \begin{bmatrix} \sqrt{x_1} \\ \vdots \\ \sqrt{x_N} \end{bmatrix}$$

ML Components: *Model*

In our dummy example, we considered a linear model, but

- we could have considered even a wired model

$$\text{label} = \left(\mathbf{w}_1^{\mathsf{T}} \boldsymbol{x}\right) \left(\mathbf{w}_2^{\mathsf{T}} \sqrt{\boldsymbol{x}}\right)$$

ML Components: *Model*

In our dummy example, we considered a linear model, but

- we could have considered even a wired model

$$\text{label} = \left(\mathbf{w}_1^\mathsf{T} \boldsymbol{x}\right) \left(\mathbf{w}_2^\mathsf{T} \sqrt{\boldsymbol{x}}\right)$$

Models are always parameterized meaning that they contain some parameters that are to be tuned; for example, $\mathbf{w}$ *in the linear model*, $\mathbf{w}_p$*'s and* b *in polynomial model, or* $\mathbf{w}_1$ *and* $\mathbf{w}_2$ *in our latter wired model*

ML Components: *Model*

In our dummy example, we considered a linear model, but

- we could have considered even a wired model

$$\text{label} = \left(\mathbf{w}_1^{\mathsf{T}} \boldsymbol{x}\right) \left(\mathbf{w}_2^{\mathsf{T}} \sqrt{\boldsymbol{x}}\right)$$

Models are always parameterized meaning that they contain some parameters that are to be tuned; for example, $\mathbf{w}$ *in the linear model*, $\mathbf{w}_p$*'s and* b *in polynomial model, or* $\mathbf{w}_1$ *and* $\mathbf{w}_2$ *in our latter wired model*

Model parameters are of two types:

- Hyperparameters
- Learnable parameters

ML Components: *Model*

Hyperparameters

Parameters that are required to specify the model *explicitly*

ML Components: *Model*

Hyperparameters

Parameters that are required to specify the model *explicitly*

Best example is the order P in the polynomial model: *we need to know* P *in order to write down the model explicitly.*

ML Components: *Model*

Hyperparameters

Parameters that are required to specify the model *explicitly*

Best example is the order P in the polynomial model: *we need to know* P *in order to write down the model explicitly. If we know* $P = 2$*; then, we know that our model is*

$$\text{label} = b + \mathbf{w}_1^\mathsf{T} \boldsymbol{x} + \mathbf{w}_2^\mathsf{T} \boldsymbol{x}^2$$

ML Components: *Model*

Hyperparameters

Parameters that are required to specify the model *explicitly*

Best example is the order P in the polynomial model: *we need to know* P *in order to write down the model explicitly. If we know* $P = 2$*; then, we know that our model is*

$$\text{label} = b + \mathbf{w}_1^\mathsf{T} \boldsymbol{x} + \mathbf{w}_2^\mathsf{T} \boldsymbol{x}^2$$

Hyperparameters are specified prior to the start of learning process

ML Components: *Model*

Hyperparameters

Parameters that are required to specify the model *explicitly*

Best example is the order P in the polynomial model: *we need to know* P *in order to write down the model explicitly. If we know* $P = 2$*; then, we know that our model is*

$$\text{label} = b + \mathbf{w}_1^\mathsf{T} \boldsymbol{x} + \mathbf{w}_2^\mathsf{T} \boldsymbol{x}^2$$

Hyperparameters are specified prior to the start of learning process

+ *How can we tune them?*
– We will discuss it in detail in this course! For the moment, assume that they are given to us by some genie

ML Components: *Model*

Learnable Parameters

Once the hyperparameters are set, we have a model with some parameters that are to be learned, such that the model fits our dataset

ML Components: *Model*

Learnable Parameters

Once the hyperparameters are set, we have a model with some parameters that are to be learned, such that the model fits our dataset

In our polynomial example after we set $P = 2$, we get the model

$$\text{label} = b + \mathbf{w}_1^\mathsf{T} \boldsymbol{x} + \mathbf{w}_2^\mathsf{T} \boldsymbol{x}^2$$

Now we need to *learn* $\mathbf{w}_1$, $\mathbf{w}_2$, *and* b:

ML Components: *Model*

Learnable Parameters

Once the hyperparameters are set, we have a model with some parameters that are to be learned, such that the model fits our dataset

In our polynomial example after we set $P = 2$, we get the model

$$\text{label} = b + \mathbf{w}_1^\mathsf{T} \boldsymbol{x} + \mathbf{w}_2^\mathsf{T} \boldsymbol{x}^2$$

Now we need to *learn* $\mathbf{w}_1$, $\mathbf{w}_2$, *and* b: let our dataset be

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

we need to find (learn) values $\mathbf{w}_1^\star$, $\mathbf{w}_2^\star$, and $b^\star$ such that

$$y_i \approx b^\star + \mathbf{w}_1^{\star\mathsf{T}} \boldsymbol{x}_i + \mathbf{w}_2^{\star\mathsf{T}} \boldsymbol{x}_i{}^2$$

ML Components: *Model*

Learnable Parameters

Once the hyperparameters are set, we have a model with some parameters that are to be learned, such that the model fits our dataset

In our polynomial example with $P = 2$: we learn $\mathbf{w}_1^\star$, $\mathbf{w}_2^\star$, and $b^\star$ such that

$$y_i \approx b^\star + \mathbf{w}_1^{\star\mathsf{T}} \boldsymbol{x}_i + \mathbf{w}_2^{\star\mathsf{T}} {\boldsymbol{x}_i}^2$$

ML Components: *Model*

Learnable Parameters

Once the hyperparameters are set, we have a model with some parameters that are to be learned, such that the model fits our dataset

In our polynomial example with $P = 2$: we learn $\mathbf{w}_1^\star$, $\mathbf{w}_2^\star$, and $b^\star$ such that

$$y_i \approx b^\star + \mathbf{w}_1^{\star\mathsf{T}} \boldsymbol{x}_i + \mathbf{w}_2^{\star\mathsf{T}} {\boldsymbol{x}_i}^2$$

+ *But what does this* $\approx$ *mean?*
– This approximation needs to be quantified!

This is what the loss function does for us

+ *How do we learn the learnable parameters?*
– We answer this after we understand the loss function

ML Components: *Loss*

Loss Function

Loss function quantifies the difference between the output of the model and the true label

ML Components: *Loss*

Loss Function

Loss function quantifies the difference between the output of the model and the true label

Back to our polynomial example with dataset $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$

- We set the hyperparameter P to $P = 2$
- We now set the learnable parameters to

$$\mathbf{w}_1 = \mathbf{w}_1^{(0)} \qquad \mathbf{w}_2 = \mathbf{w}_2^{(0)} \qquad b = b^{(0)}$$

ML Components: *Loss*

Loss Function

Loss function quantifies the difference between the output of the model and the true label

Back to our polynomial example with dataset $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$

- We set the hyperparameter P to $P = 2$
- We now set the learnable parameters to

$$\mathbf{w}_1 = \mathbf{w}_1^{(0)} \qquad \mathbf{w}_2 = \mathbf{w}_2^{(0)} \qquad b = b^{(0)}$$

If we now give the data-point $\boldsymbol{x}_i$ to this model as input, it returns

$$z_i = b^{(0)} + \mathbf{w}_1^{(0)} \boldsymbol{x}_i + \mathbf{w}_2^{(0)} {\boldsymbol{x}_i}^2$$

ML Components: *Loss*

Back to our polynomial example with dataset $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$

$$z_i = b^{(0)} + \mathbf{w}_1^{(0)} \boldsymbol{x}_i + \mathbf{w}_2^{(0)} {\boldsymbol{x}_i}^2$$

ML Components: *Loss*

Back to our polynomial example with dataset $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$

$$z_i = b^{(0)} + \mathbf{w}_1^{(0)} \boldsymbol{x}_i + \mathbf{w}_2^{(0)} {\boldsymbol{x}_i}^2$$

Loss determines the difference between z_i and true label y_i in $\mathbb{D}$

$$\mathcal{L}(z_i, y_i) = \ell_i \in \mathbb{R}$$

ML Components: *Loss*

Back to our polynomial example with dataset $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$

$$z_i = b^{(0)} + \mathbf{w}_1^{(0)} \boldsymbol{x}_i + \mathbf{w}_2^{(0)} {\boldsymbol{x}_i}^2$$

Loss determines the difference between z_i and true label y_i in $\mathbb{D}$

$$\mathcal{L}\left(z_i, y_i\right) = \ell_i \in \mathbb{R}$$

Let's see few examples of loss function:

We can calculate the *squared error*

$$\mathcal{L}\left(z_i, y_i\right) = \left(z_i - y_i\right)^2$$

which intuitively determines the energy of the difference

ML Components: *Loss*

Back to our polynomial example with dataset $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$

$$z_i = b^{(0)} + \mathbf{w}_1^{(0)} \boldsymbol{x}_i + \mathbf{w}_2^{(0)} {\boldsymbol{x}_i}^2$$

Loss determines the difference between z_i and true label y_i in $\mathbb{D}$

$$\mathcal{L}\left(z_i, y_i\right) = \ell_i \in \mathbb{R}$$

Let's see few examples of loss function:

We can calculate the *indicator error*

$$\mathcal{L}\left(z_i, y_i\right) = \mathbb{1}\left(z_i \neq y_i\right) = \begin{cases} 1 & z_i \neq y_i \\ 0 & z_i = y_i \end{cases}$$

which indicates the occurrence of error

ML Components: *A Quick Wrap-up*

Any ML problem has three components:

1. *Dataset* which is the collection of samples

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

2. *Model* that describes the relation between the input and label

$$z = f_{\mathbf{h}}\left(\boldsymbol{x}|\mathbf{w}\right)$$

3. *Loss* quantifies the difference between the *model's output* and *true label*

$$\mathcal{L}\left(z_i, y_i\right)$$

ML Components: *A Quick Wrap-up*

Any ML problem has three components:

1. *Dataset* which is the collection of samples

   supervised learning $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$

2. *Model* that describes the relation between the input and label

$$z = f_{\mathbf{h}}(\boldsymbol{x}|\mathbf{w})$$

3. *Loss* quantifies the difference between the *model's output* and *true label*

$$\mathcal{L}(z_i, y_i)$$

ML Components: *A Quick Wrap-up*

Any ML problem has three components:

1. *Dataset* which is the collection of samples

   supervised learning $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$

2. *Model* that describes the relation between the input and label

   $\mathbf{h}$: Hyperparameters $z = f_{\mathbf{h}}(\boldsymbol{x}|\mathbf{w})$ $\mathbf{w}$: Learnable Parameters

3. *Loss* quantifies the difference between the *model's output* and *true label*

$$\mathcal{L}(z_i, y_i)$$

What's Next?

Now that we know the components of an ML problem, we need to answer the following question:

+ *How can we solve the problem? Or, speaking in the language of ML people, how can we address the learning task?*

What's Next?

Now that we know the components of an ML problem, we need to answer the following question:

+ *How can we solve the problem? Or, speaking in the language of ML people, how can we address the learning task?*
  - Recall that the learning task is to tune the learnable parameters ($\mathbf{w}$ in the last slide): once we tune them, the problem is over

We approximate the lable of a new input $\boldsymbol{x}_{\text{new}}$ as $y_{\text{new}} = f_{\mathbf{h}}\left(\boldsymbol{x}_{\text{new}} \middle| \mathbf{w}\right)$

The process of tuning the learnable parameters is called

training of the model

What's Next?

Now that we know the components of an ML problem, we need to answer the following question:

+ *How can we solve the problem? Or, speaking in the language of ML people, how can we address the learning task?*
  - Recall that the learning task is to tune the learnable parameters ($\mathbf{w}$ in the last slide): once we tune them, the problem is over

We approximate the lable of a new input $\boldsymbol{x}_{\text{new}}$ as $y_{\text{new}} = f_{\mathbf{h}}\left(\boldsymbol{x}_{\text{new}} \middle| \mathbf{w}\right)$

The process of tuning the learnable parameters is called

training of the model

+ *How do we do the training?*
  - We see it very shortly, but first we need a more serious example!

Classification

Classification is a supervised learning problem in which

labels belong to a discrete set: $y_i \in \{c_1, \ldots, c_J\}$

the label y_i represents the class to which the input $\boldsymbol{x}_i$ belongs

Classification

Classification is a supervised learning problem in which

labels belong to a discrete set: $y_i \in \{c_1, \ldots, c_J\}$

the label y_i represents the class to which the input $\boldsymbol{x}_i$ belongs

Best example is image classification:

- The dataset contains some images and their labels

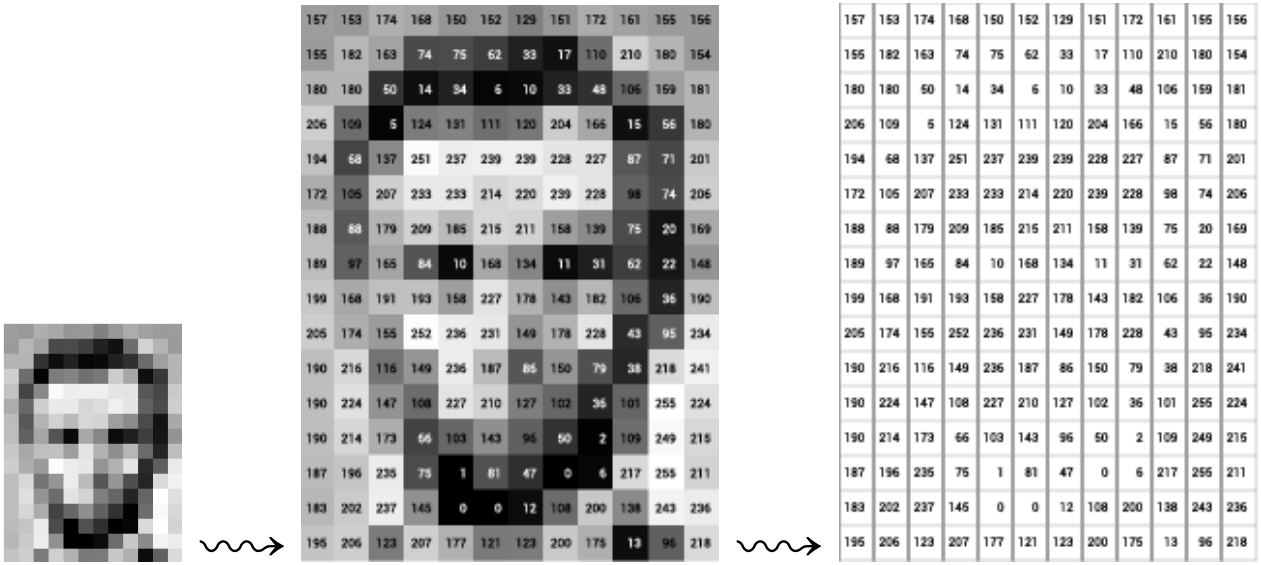
$$\mathbb{D} = \{(\text{image } 1, \text{dog}), (\text{image } 2, \text{cat}), \ldots, (\text{image } I, \text{cat})\}$$

- The images are either *dog* or *cat*

We can simply convert $\mathbb{D}$ *into a numerical dataset*

Classification: *Image Classification*

Any image is nothing but *a collection of pixel values*



We will work with images a lot in this course, so we will learn the meaning of pixel values very well. For now, the important thing is that

for each image, we can make a vector containing its pixel values

Thus, image i in $\mathbb{D}$ can be shown by a pixel vector $\boldsymbol{x}_i$

Classification: *Image Classification*

The labels can further be marked by integer numbers as classes

In our example, we have *two classes of dog and cat*: so, we could say

$$\textit{label of dog images} = 0 \qquad \text{and} \qquad \textit{label of cat images} = 1$$

Classification: *Image Classification*

The labels can further be marked by integer numbers as classes

In our example, we have *two classes of dog and cat*: so, we could say

label of dog images $= 0$ and *label of cat images* $= 1$

So, a collection of I images like

$$\mathbb{D} = \{(\text{image } 1, \text{dog}), (\text{image } 2, \text{cat}), \ldots, (\text{image } I, \text{cat})\}$$

can be represented by the numerical dataset

$$\mathbb{D} = \{(\boldsymbol{x}_1, 0), (\boldsymbol{x}_2, 1), \ldots, (\boldsymbol{x}_I, 1)\}$$

Classification: *Image Classification*

The labels can further be marked by integer numbers as classes

In our example, we have *two classes of dog and cat*: so, we could say

label of dog images $= 0$ and *label of cat images* $= 1$

So, a collection of I images like

$$\mathbb{D} = \{(\text{image } 1, \text{dog}), (\text{image } 2, \text{cat}), \ldots, (\text{image } I, \text{cat})\}$$

can be represented by the numerical dataset

$$\mathbb{D} = \{(\boldsymbol{x}_1, 0), (\boldsymbol{x}_2, 1), \ldots, (\boldsymbol{x}_I, 1)\}$$

pixel vector of image 1

Classification: *Image Classification*

The labels can further be marked by integer numbers as classes

In our example, we have *two classes of dog and cat*: so, we could say

label of dog images $= 0$ and *label of cat images* $= 1$

So, a collection of I images like

$$\mathbb{D} = \{(\text{image } 1, \text{dog}), (\text{image } 2, \text{cat}), \ldots, (\text{image } I, \text{cat})\}$$

can be represented by the numerical dataset

$$\mathbb{D} = \{(\boldsymbol{x}_1, 0), (\boldsymbol{x}_2, 1), \ldots, (\boldsymbol{x}_I, 1)\}$$

pixel vector of image 1

it's an image of a dog

Classification: *Binary Classification*

We now start with the basic case of binary classification

Binary Classification

A classification problem with only two classes, i.e., $y_i \in \{0, 1\}$

Our example was a binary classification with dog:0 and cat:1

Classification: *Binary Classification*

We now start with the basic case of binary classification

Binary Classification

A classification problem with only two classes, i.e., $y_i \in \{0, 1\}$

Our example was a binary classification with dog:0 and cat:1

Binary classification is very fundamental, since

- it is one of the very first problems investigated in ML
- other classification problems are reduced into a series of binary classifications: *say we want to classify an image as dog, cat or car*

Classification: *Binary Classification*

We now start with the basic case of binary classification

Binary Classification

A classification problem with only two classes, i.e., $y_i \in \{0, 1\}$

Our example was a binary classification with dog:0 and cat:1

Binary classification is very fundamental, since

- it is one of the very first problems investigated in ML
- other classification problems are reduced into a series of binary classifications: *say we want to classify an image as dog, cat or car*
  - *Binary Classification 1: Is it class 0: dog or class 1: {cat, car}?*
    - ↳ *If class 0 ⇝ classification ended*
    - ↳ *If class 1 ⇝ Binary Classification 2: Is it class 0: cat or class 1: car?*

Classification: *Binary Classification*

We now start with the basic case of binary classification

Binary Classification

A classification problem with only two classes, i.e., $y_i \in \{0, 1\}$

Our example was a binary classification with dog:0 and cat:1

Binary classification is very fundamental, since

- it is one of the very first problems investigated in ML
- other classification problems are reduced into a series of binary classifications: *say we want to classify an image as dog, cat or car*
  - *Binary Classification 1: Is it class 0: dog or class 1: {cat, car}?*
    - ↳ *If class 0 ⇝ classification ended*
    - ↳ *If class 1 ⇝ Binary Classification 2: Is it class 0: cat or class 1: car?*

Let's build the main components of this ML problem

Binary Classification: *Dataset*

Dataset is similar to what we had in our dog or cat example

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

- $\boldsymbol{x}_i \in \mathbb{R}^N$ is a real-valued N-dimensional vector
  - *For instance it contains the pixel values of an image with N pixels*
- $y_i \in \{0, 1\}$ is a binary label

Binary Classification: *Model*

+ *What should our model be?*
– The model gets N real numbers and return a binary number

Binary Classification: *Model*

+ *What should our model be?*
- The model gets N real numbers and return a binary number

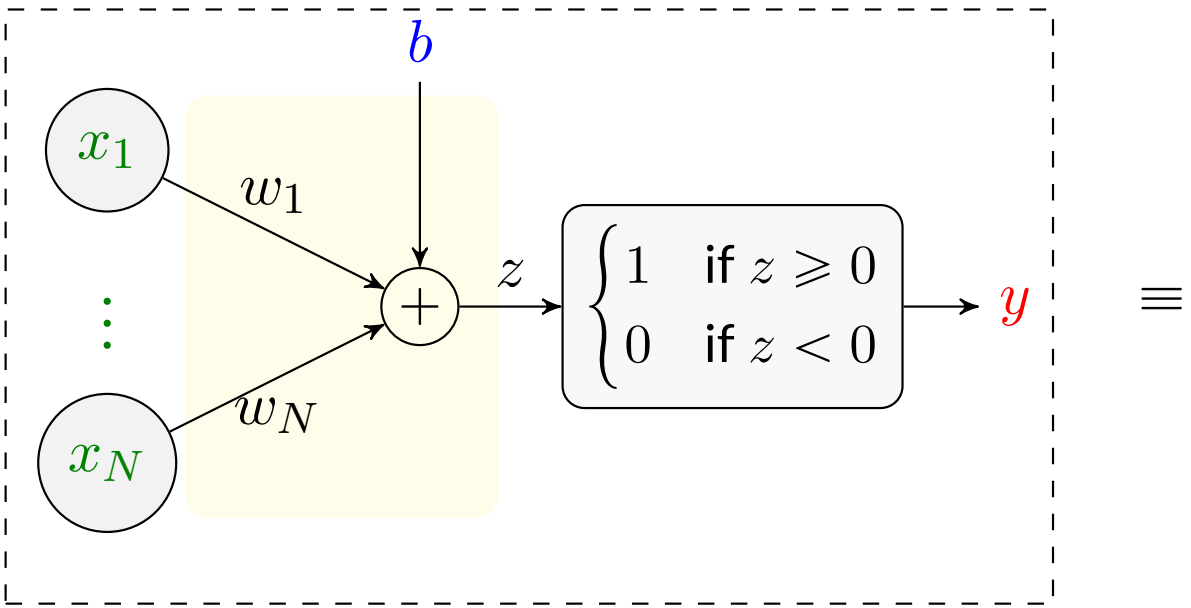
Classification models are also called *Classifiers*

Binary Classification: *Model*

+ *What should our model be?*
- The model gets N real numbers and return a binary number

Classification models are also called *Classifiers*

Let's start with perceptron who is the father of neural networks

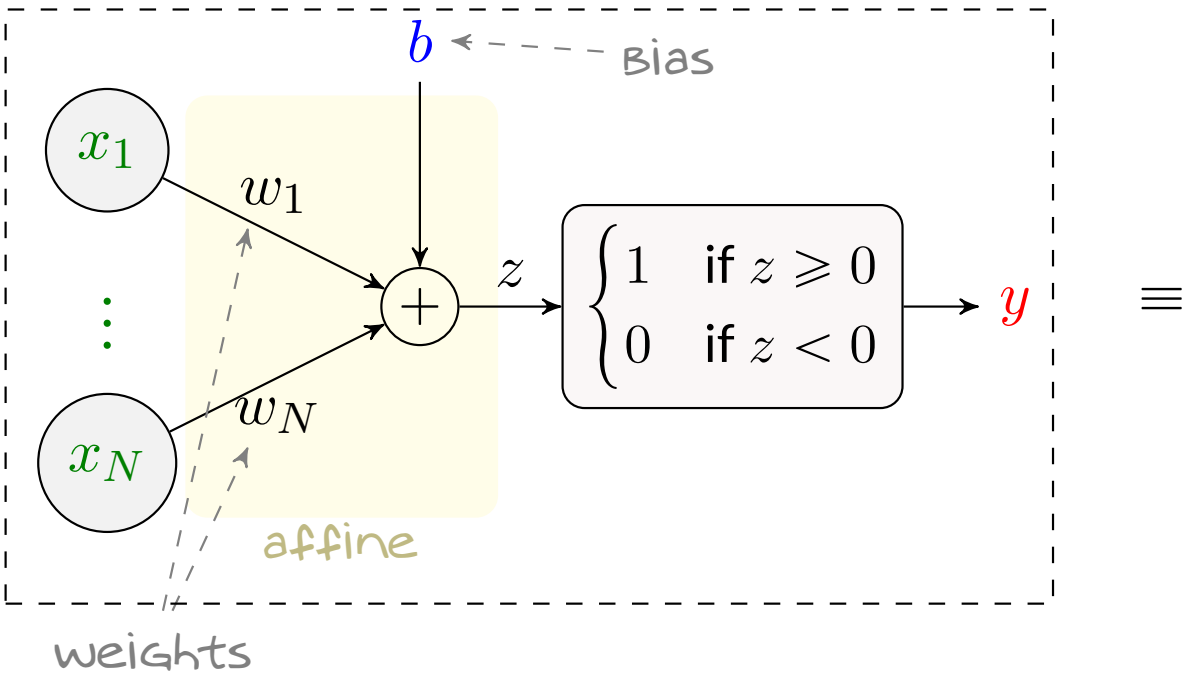


Binary Classification: *Model*

+ *What should our model be?*
- The model gets N real numbers and return a binary number

Classification models are also called *Classifiers*

Let's start with perceptron who is the father of neural networks



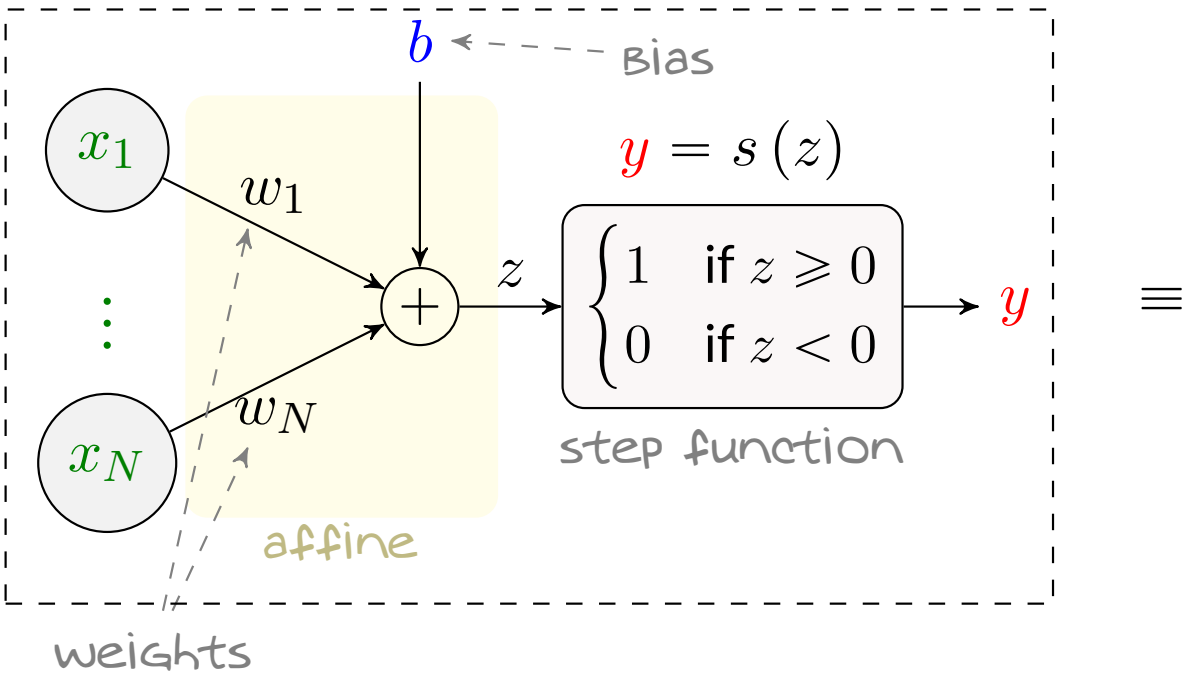
$\equiv$

Binary Classification: *Model*

+ *What should our model be?*
- The model gets N real numbers and return a binary number

Classification models are also called *Classifiers*

Let's start with perceptron who is the father of neural networks



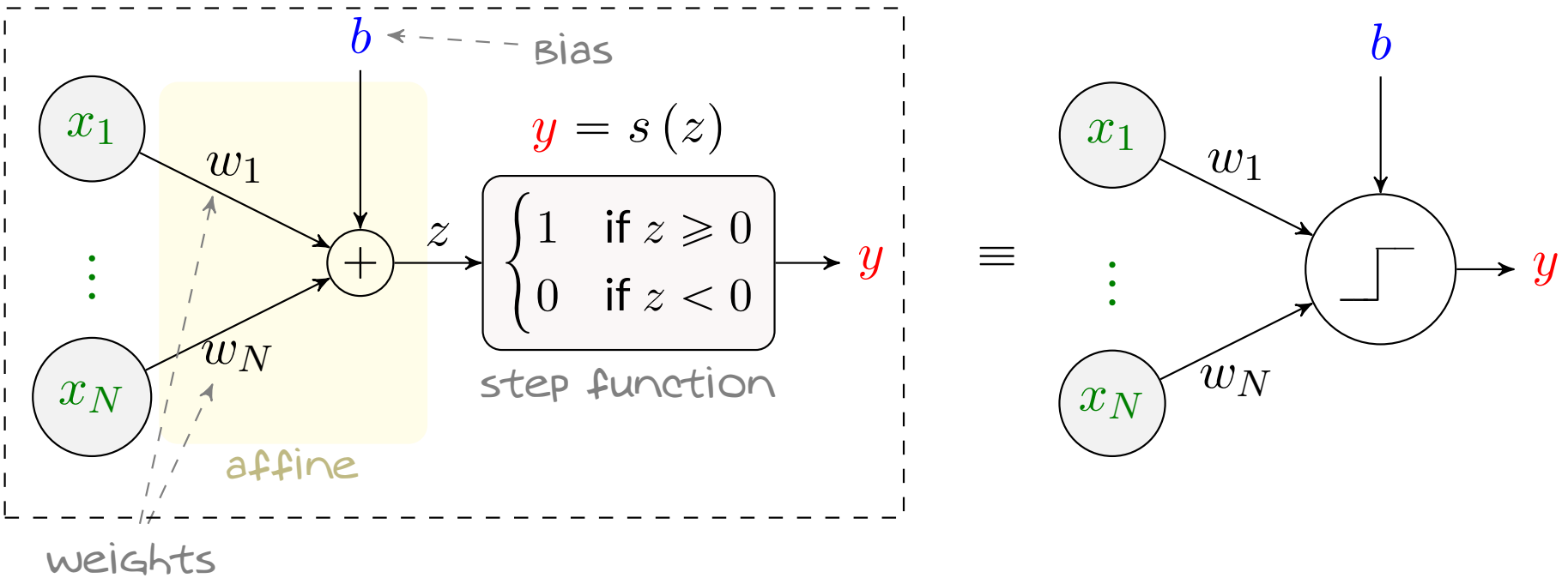
$\equiv$

Binary Classification: *Model*

+ *What should our model be?*
- The model gets N real numbers and return a binary number

Classification models are also called *Classifiers*

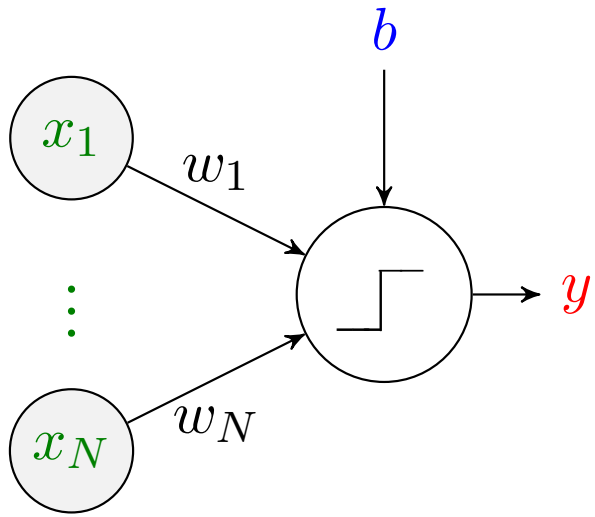
Let's start with perceptron who is the father of neural networks



Binary Classification: *Perceptron*

Perceptron is a *linear classifier*

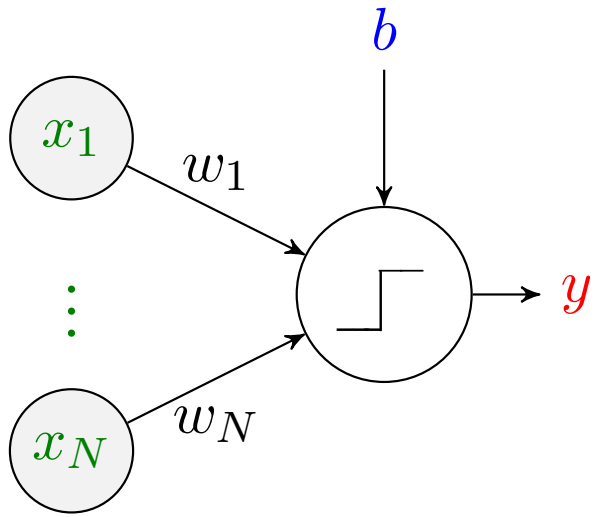
- it determines an affine transform of $\boldsymbol{x}$
- it classifies using the sign of this transform



Binary Classification: *Perceptron*

Perceptron is a *linear classifier*

- it determines an affine transform of $\boldsymbol{x}$
- it classifies using the sign of this transform

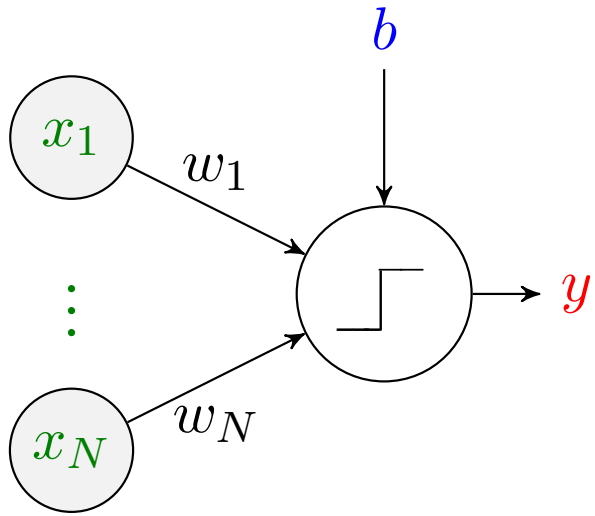


Mathematically, the perceptron is described by

$$y = s\Big(\sum_{n=1}^{N} w_n x_n + b\Big) = s\Big(\underbrace{[w_1, \ldots, w_N]}_{\mathbf{w}^\mathsf{T}} \underbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix}}_{\boldsymbol{x}} + b\Big) = s(\mathbf{w}^\mathsf{T}\boldsymbol{x} + b)$$

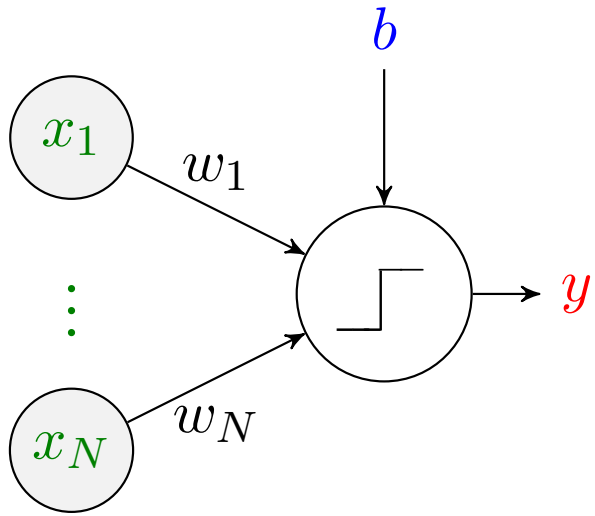
Binary Classification: *Perceptron*

$$y = s(\mathbf{w}^\mathsf{T}\boldsymbol{x} + b)$$



Binary Classification: *Perceptron*

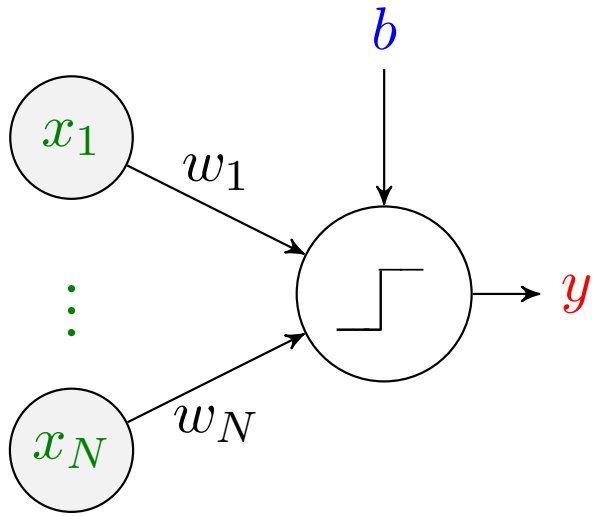
$$y = s(\mathbf{w}^\mathsf{T}\boldsymbol{x} + b)$$



+ *Is there any hyperparameter in this model?*
– We could look at step function as a hyperparameter: *we could have chosen another function to map the affine transform to a binary label*

Binary Classification: *Perceptron*

$$y = s(\mathbf{w}^\mathsf{T}\boldsymbol{x} + b)$$

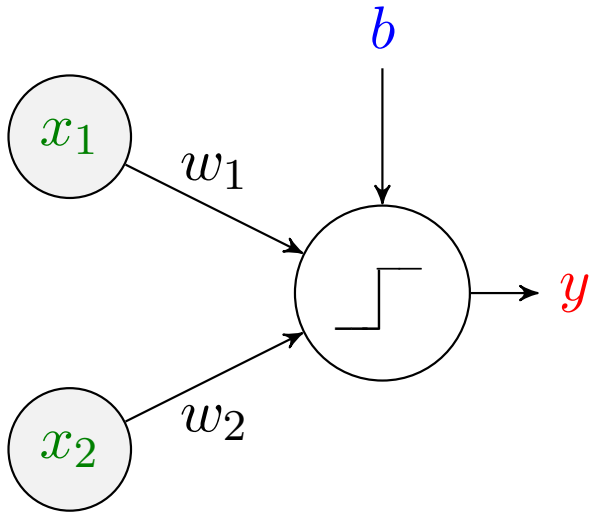


+ *Is there any hyperparameter in this model?*
– We could look at step function as a hyperparameter: *we could have chosen another function to map the affine transform to a binary label*
+ *Is there any learnable parameter in this model?*
– Of course! $\mathbf{w}$ and b are the learnable parameters

Perceptron: *Geometrical Interpretation*

Perceptron geometrically realizes a linear division of $\mathbb{R}^N$

To see this, let's focus on the two-dimensional case, i.e., $N = 2$



$$y = s(\mathbf{w}^\mathsf{T}\boldsymbol{x} + b) = s(w_1x_1 + w_2x_2 + b) = \begin{cases} 1 & \text{if } w_1x_1 + w_2x_2 + b \geqslant 0 \\ 0 & \text{if } w_1x_1 + w_2x_2 + b < 0 \end{cases}$$

Perceptron: *Geometrical Interpretation*

Let's go to the two-dimensional space with one dimension denoting x_1 *and the other* x_2 *and plot the line*

$$z = w_1 x_1 + w_2 x_2 + b = 0$$

for some $w_1, w_2 > 0$ *and* $b < 0$

Perceptron: *Geometrical Interpretation*

Let's go to the two-dimensional space with one dimension denoting x_1 *and the other* x_2 *and plot the line*

$$z = w_1 x_1 + w_2 x_2 + b = 0$$

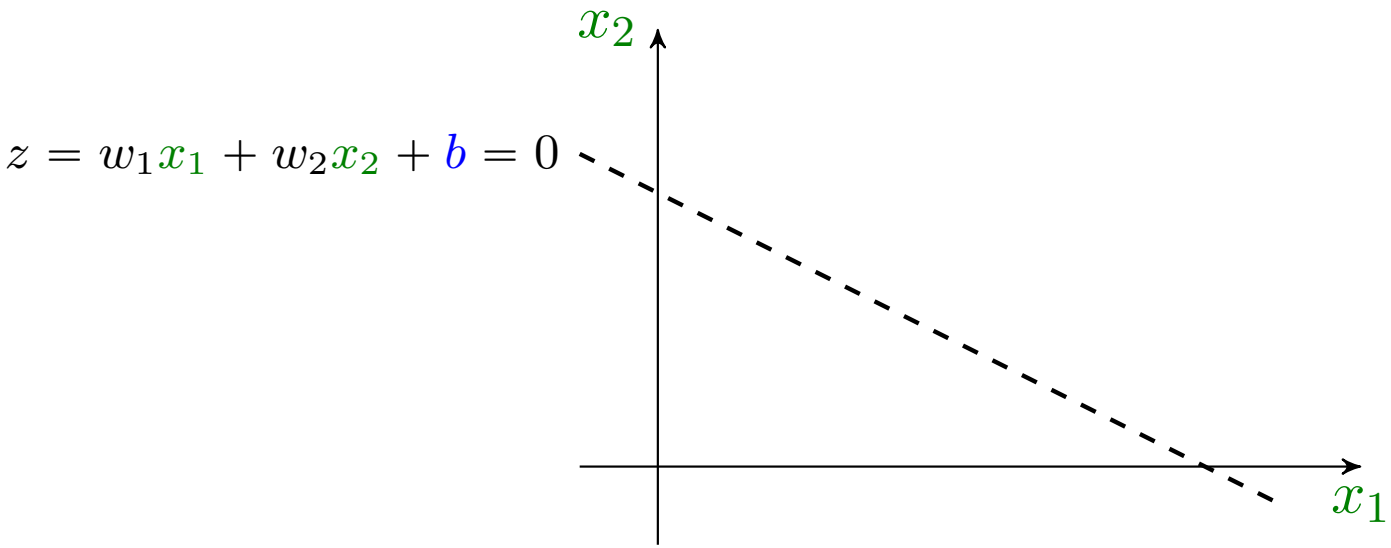
for some $w_1, w_2 > 0$ *and* $b < 0$

Perceptron: *Geometrical Interpretation*

Let's go to the two-dimensional space with one dimension denoting x_1 *and the other* x_2 *and plot the line*

$$z = w_1 x_1 + w_2 x_2 + b = 0$$

for some $w_1, w_2 > 0$ *and* $b < 0$

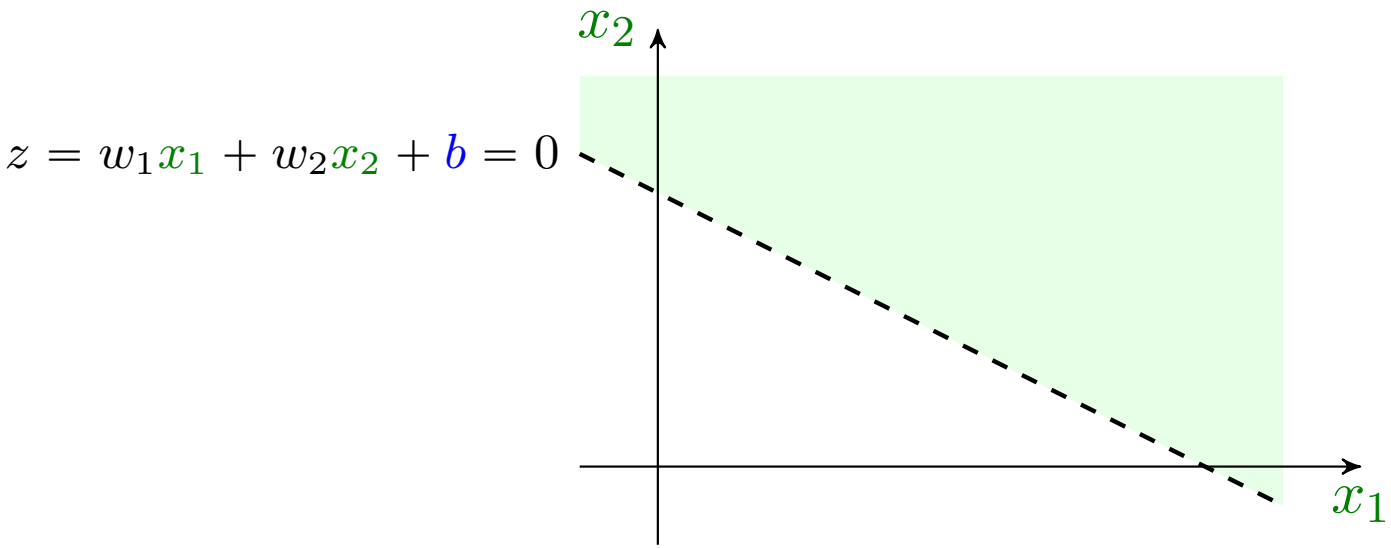


Perceptron: *Geometrical Interpretation*

Let's go to the two-dimensional space with one dimension denoting x_1 *and the other* x_2 *and plot the line*

$$z = w_1 x_1 + w_2 x_2 + b = 0$$

for some $w_1, w_2 > 0$ *and* $b < 0$

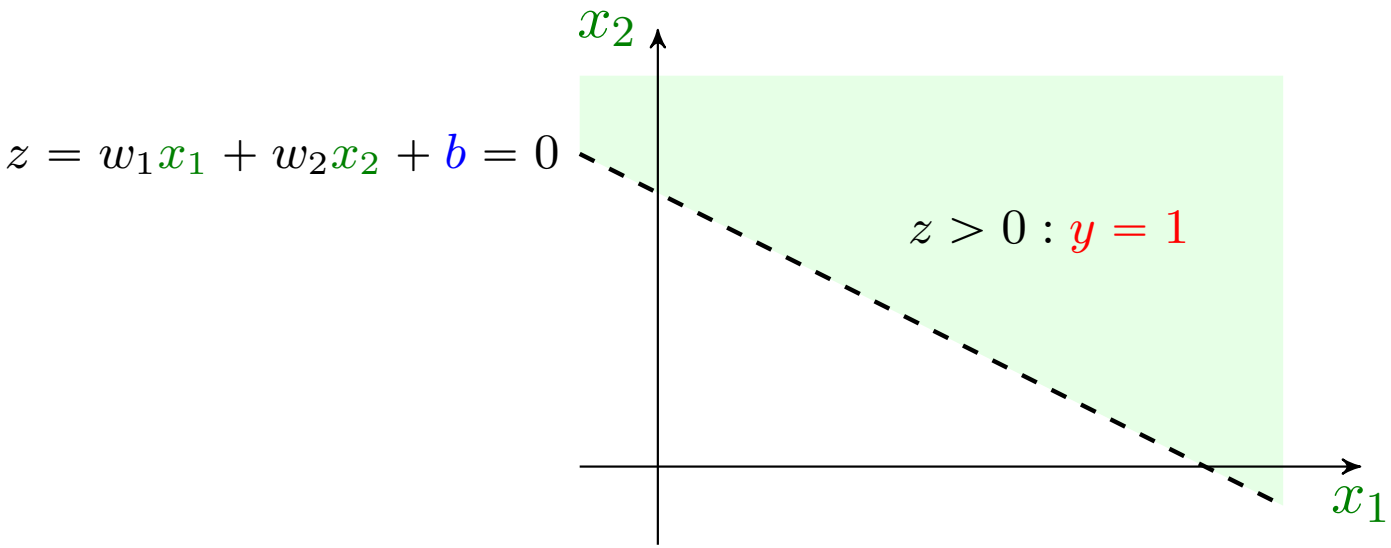


Perceptron: *Geometrical Interpretation*

Let's go to the two-dimensional space with one dimension denoting x_1 *and the other* x_2 *and plot the line*

$$z = w_1 x_1 + w_2 x_2 + b = 0$$

for some $w_1, w_2 > 0$ *and* $b < 0$

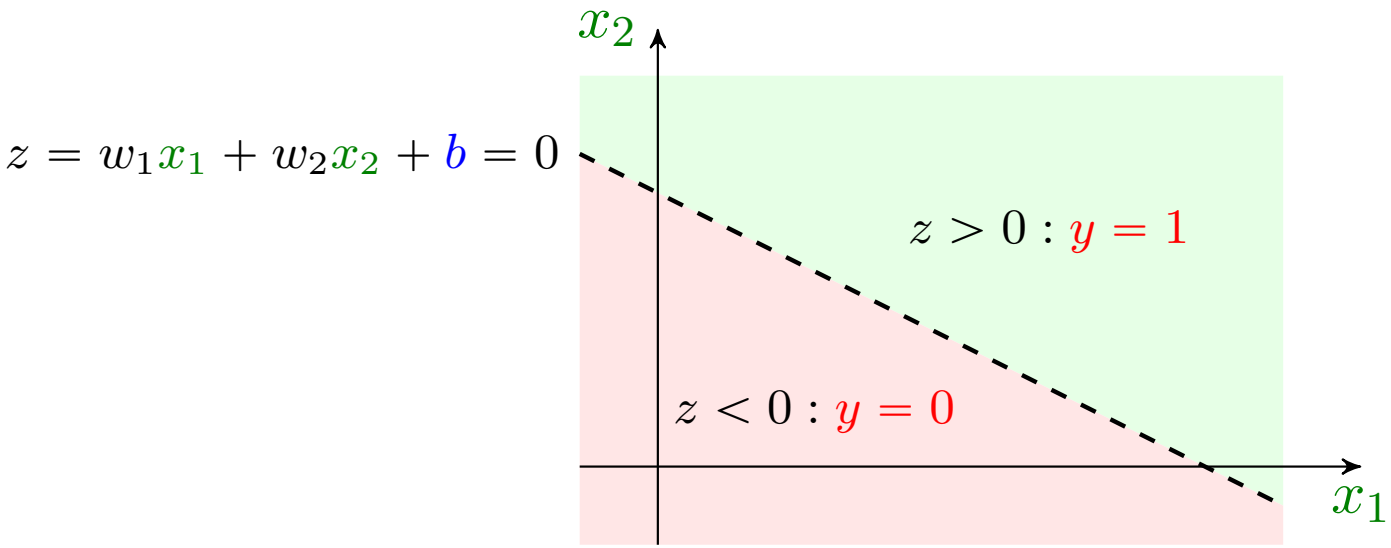


Perceptron: *Geometrical Interpretation*

Let's go to the two-dimensional space with one dimension denoting x_1 and the other x_2 and plot the line

$$z = w_1 x_1 + w_2 x_2 + b = 0$$

for some $w_1, w_2 > 0$ and $b < 0$

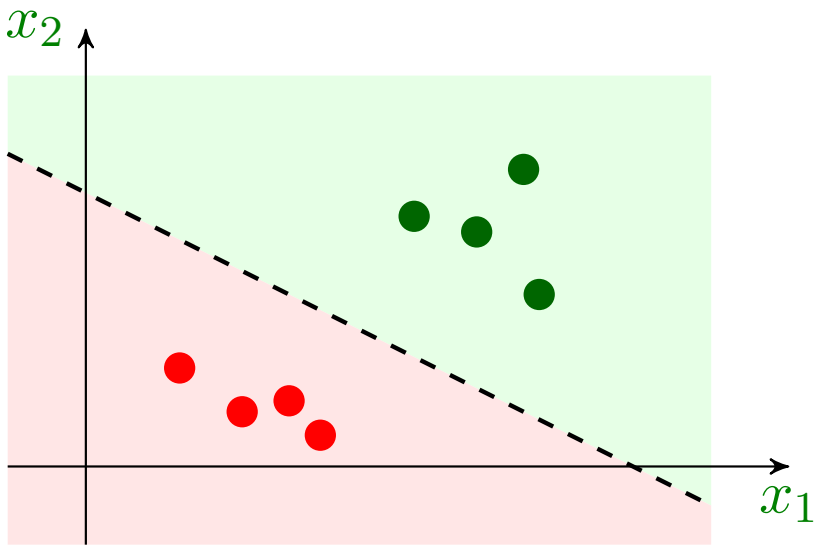


Perceptron draws a line, and then classifies every point above it with label 1 and every point below it with label 0

Perceptron: *Geometrical Interpretation*

So, if we have a perceptron with learnable parameters $w_1, w_2 > 0$, *and* $b < 0$

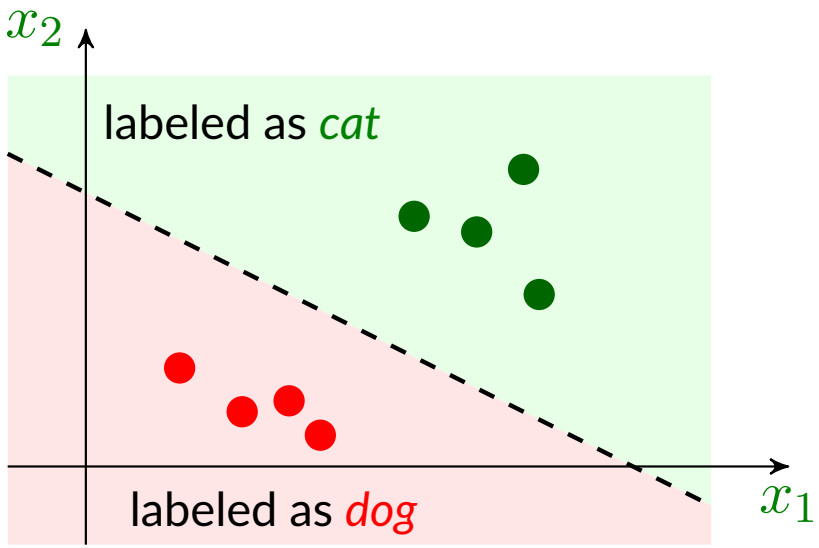
$$z = w_1 x_1 + w_2 x_2 + b = 0$$



Perceptron: *Geometrical Interpretation*

So, if we have a perceptron with learnable parameters $w_1, w_2 > 0$, *and* $b < 0$

$$z = w_1 x_1 + w_2 x_2 + b = 0$$



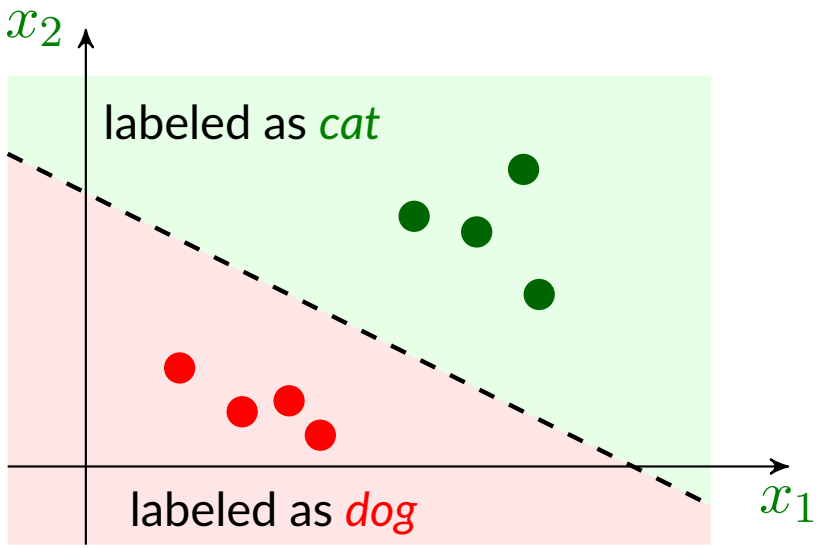
For instance in the example of image classification

- *Every • represents a data-point* $\equiv$ *pixel vector: we should think of a two-pixel image* ☺
- *Perceptron gets the values of the two pixels and predicts whether it is an image of a dog or a cat*

Perceptron: *Geometrical Interpretation*

So, if we have a perceptron with learnable parameters $w_1, w_2 > 0$, *and* $b < 0$

$$z = w_1 x_1 + w_2 x_2 + b = 0$$



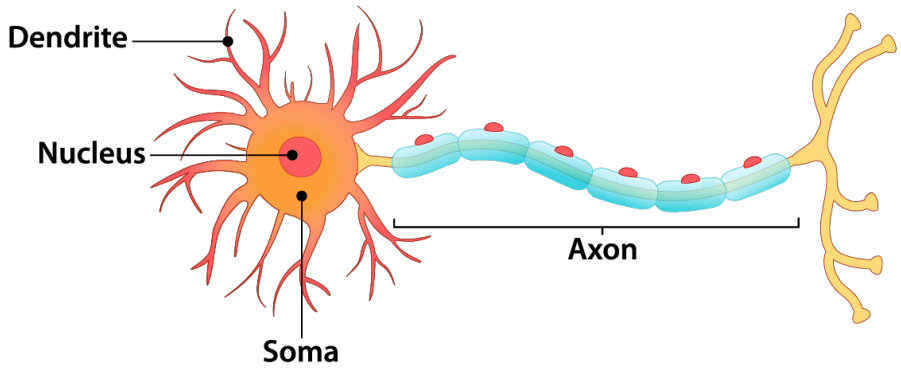
For instance in the example of image classification

- *Every • represents a data-point* $\equiv$ *pixel vector: we should think of a two-pixel image* ☺
- *Perceptron gets the values of the two pixels and predicts whether it is an image of a dog or a cat*

In realistic problems, this viewpoint extends to large-dimensional space

Perceptron: *A Bit of History*

- Perceptron machine was implemented by Frank Rosenblatt in 1957
- He wrote a paper in 1958 illustrating the machine and its algorithm
- The original idea is however older than that
  - It was proposed in 1943 by Warren McCulloch and Walter Pitts
  - They introduced it to abstractly describe biological neurons
  - This was why perceptron is also called McCulloch-Pitts neuron



Perceptron was a breakthrough in the long-time ongoing attempt to understand the functionality of brain; however,

for us, perceptron is simply a linear classification model

Components of Binary Classification

Back to binary classification: we have the first two components

1. A dataset with N-dimensional inputs $\boldsymbol{x}_i$ and binary labels $y_i \in \{0, 1\}$

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

2. Perceptron as the model

$$y = s(\mathbf{w}^\mathsf{T}\boldsymbol{x} + b)$$

We are now looking for the third component, i.e., the loss function

Binary Classification: *Loss*

In general, we can use any loss function

- *squared error*
- *Kullback-Leibler divergence* ⇜ we are going to learn it soon
- error indicator
- . . .

Let's use the one initially used in Rosenblatt's machine, i.e., error indicator[1]

For two binary variables y and $\hat{y}$, the loss function is

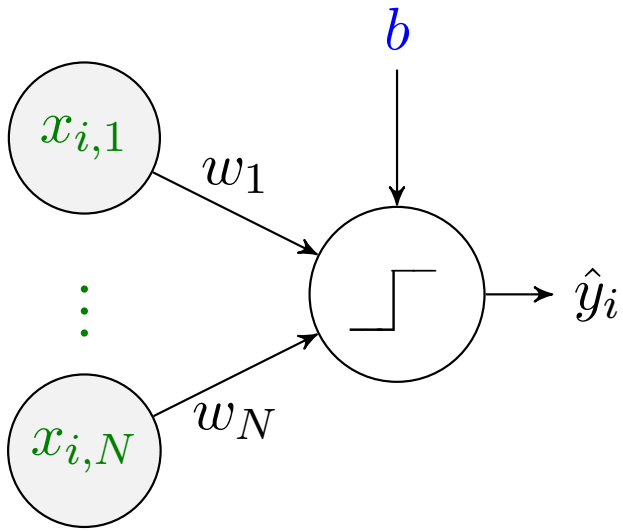
$$\mathcal{L}(\hat{y}, y) = \mathbb{1}(\hat{y} \neq y) = \begin{cases} 1 & \hat{y} \neq y \\ 0 & \hat{y} = y \end{cases}$$

[1] We will realize later that this is not a good choice

Binary Classification: *Loss*

How does this loss look like when we classify via perceptron?

Let $(\boldsymbol{x}_i, y_i)$ be a data-point in dataset. We give $\boldsymbol{x}_i$ to the perceptron



and get $\hat{y}_i$. The loss says whether we got the same label (0) or not (1)

$$\mathcal{L}\left(\hat{y}_i, y_i\right) = \mathbb{1}\left(\hat{y}_i \neq y_i\right) = \begin{cases} 1 & \hat{y}_i \neq y_i \\ 0 & \hat{y}_i = y_i \end{cases}$$

Training a Model

Up to now, we have learned the main components of ML problems and seen a classical example, i.e., classification. We now want to understand how we can tune the learnable parameters of a model for our problem.

As mentioned, tuning the learnable parameters of a model is called

training of the model

Before starting with training, let's review some basics of probability theory

Recap: Probability Theory

Discrete Random Variables

A discrete random variable x is described by *probability mass function* $P(x)$

$$\Pr\{x = x_0\} = P(x_0)$$

Assuming that $x \in \mathbb{X}$, we have

$$\sum_{x \in \mathbb{X}} P(x) = 1$$

Recap: Probability Theory

Discrete Random Variables

A discrete random variable x is described by *probability mass function* $P(x)$

$$\Pr\{x = x_0\} = P(x_0)$$

Assuming that $x \in \mathbb{X}$, we have

$$\sum_{x \in \mathbb{X}} P(x) = 1$$

Continuous Random Variables

A continuous random variable x is described by *probability density function* $P(x)$

$$\Pr\{a < x \leqslant b\} = \int_a^b P(x)\mathrm{d}x$$

We have in this case

$$\int_{-\infty}^{+\infty} P(x)\mathrm{d}x = 1$$

When we talk about a general random variable, we call $P(x)$ the distribution

Recap: Probability Theory

A random vector $\boldsymbol{x}$ is a vector of random variables and its distribution

$$P(\boldsymbol{x}) = P(x_1, \ldots, x_N)$$

is the *joint distribution of the entries*

Recap: Probability Theory

A random vector $\boldsymbol{x}$ is a vector of random variables and its distribution

$$P(\boldsymbol{x}) = P(x_1, \ldots, x_N)$$

is the *joint distribution of the entries*

Assume $\boldsymbol{x} \in \mathbb{R}^N$ is a vector of random variables, its *expectation* is

Discrete: $\mathbb{E}\{\boldsymbol{x}\} = \sum_{\boldsymbol{x} \in \mathbb{X}^N} \boldsymbol{x} P(\boldsymbol{x})$ Continuous: $\mathbb{E}\{\boldsymbol{x}\} = \int \boldsymbol{x} P(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}$

and we can extend the definition to any function of $\boldsymbol{x}$*, i.e.,*

$$\mathbb{E}\{f(\boldsymbol{x})\} = \int f(\boldsymbol{x}) P(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}$$

Recap: Law of Large Numbers

Consider a random sequence $x_1, \ldots, x_I$: *we call it independent and identically distributed (i.i.d.) with* $x \sim P(x)$ *if* x_i*'s are generated independently all with the same distribution* $P(x)$

Law of Large Numbers

Assume $x_1, \ldots, x_I$ is an i.i.d. sequence with $x \sim P(x)$; then, we have[2]

$$\frac{1}{I} \sum x_i \to \mathbb{E}\{x\}$$

as $I \to \infty$

In simple words: if we *arithmetically average* too many samples of a random process, we get a value very close to its *expectation*

[2] Of course under some conditions which we assume holding

Training a Model

Let us now use the probability theory to derive a meaningful approach of model training: for sake of simplicity, let's assume that we have dataset

$$\mathbb{D} = \{(x_i, y_i) : i = 1, \ldots, I\}$$

with scalar (one-dimensional) inputs, i.e., x_i's are real numbers.

Let us further denote our model as

$$y = f\left(x|\mathbf{w}\right)$$

where in this notation

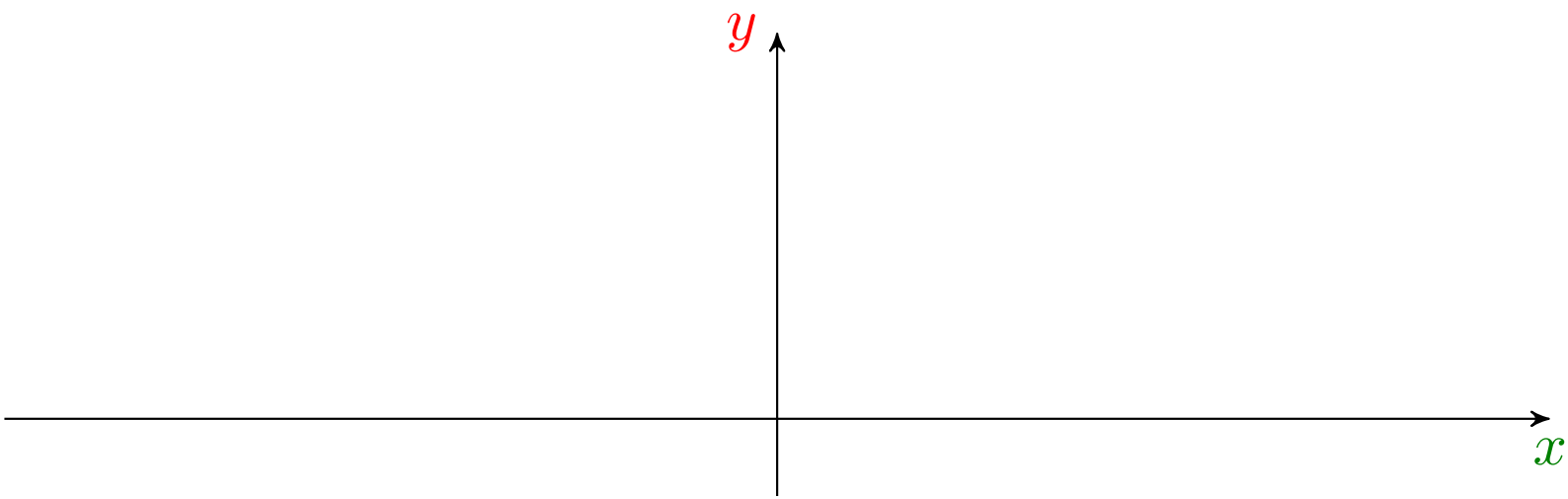
- $\mathbf{w}$ contains all learnable parameters that are to be learned through the training procedure
- we drop the hyperparameters, since at this point we assume *they are fixed by a genie*

Training a Model

The one-dimensional assumption helps us visualize the model

$$y = f\left(x|\mathbf{w}\right)$$

With scalar input, we can visualize the model as



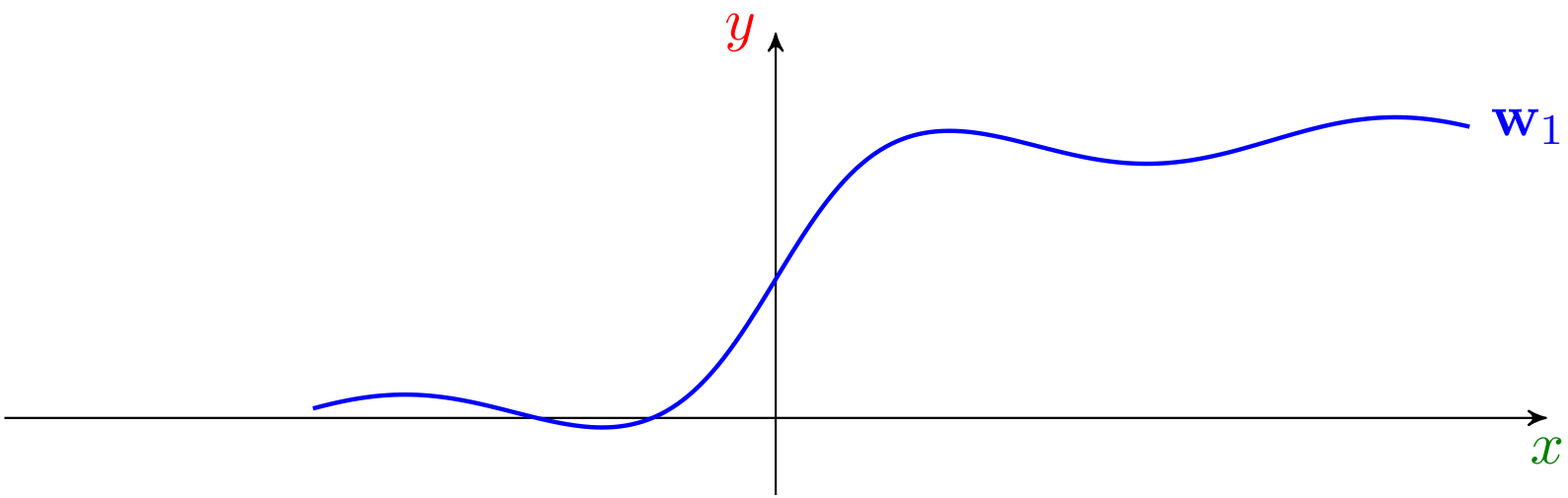
As learnable parameters change, the model represents different functions

Training a Model

The one-dimensional assumption helps us visualize the model

$$y = f\left(x|\mathbf{w}\right)$$

With scalar input, we can visualize the model as



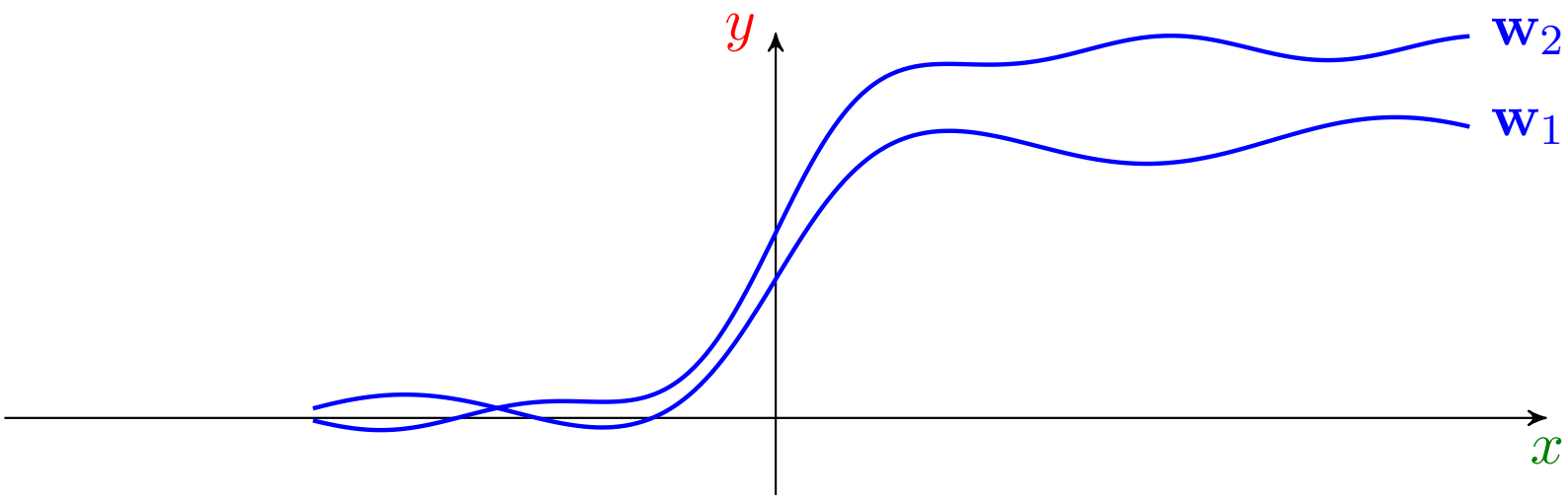
As learnable parameters change, the model represents different functions

Training a Model

The one-dimensional assumption helps us visualize the model

$$y = f\left(x|\mathbf{w}\right)$$

With scalar input, we can visualize the model as



As learnable parameters change, the model represents different functions

Training a Model

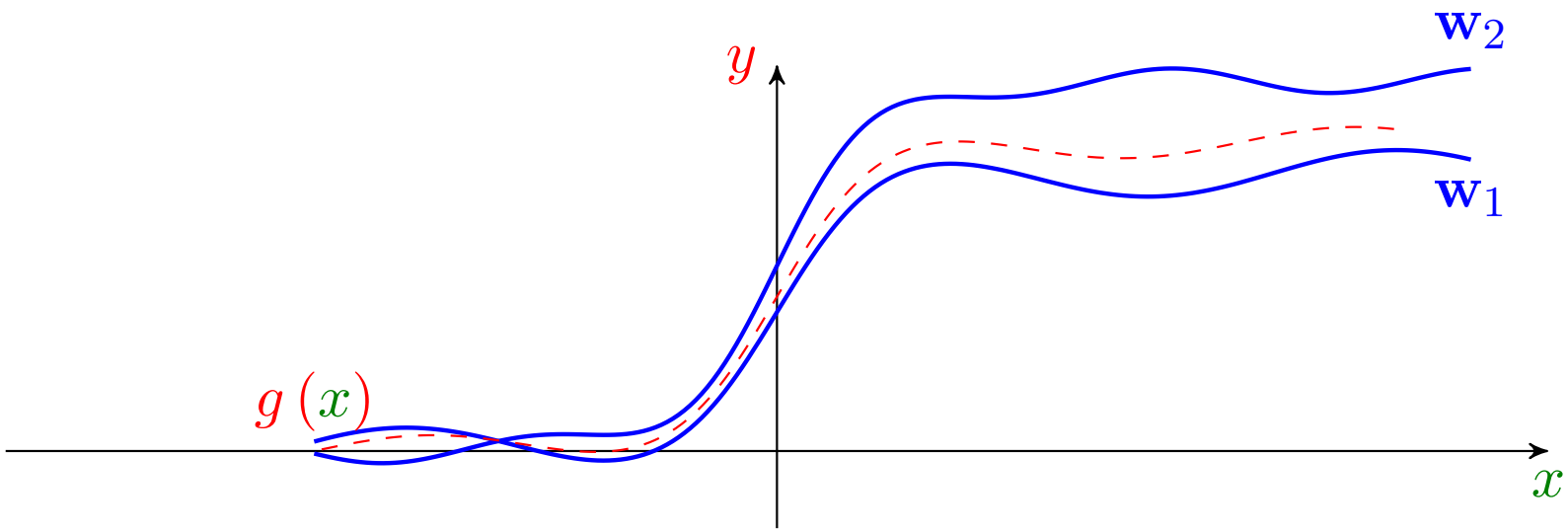
In reality, the labels and inputs are related through a function $g(\cdot)$

$$y = g(x)$$

The function $g(\cdot)$ is however unknown to us

In fact, our whole learning task is to learn it from the dataset!

+ *What if we knew function* $g(\cdot)$*? How would have trained our model?*
– Well! We would have tuned $\mathbf{w}$ until the model matches $g(\cdot)$



Training a Model

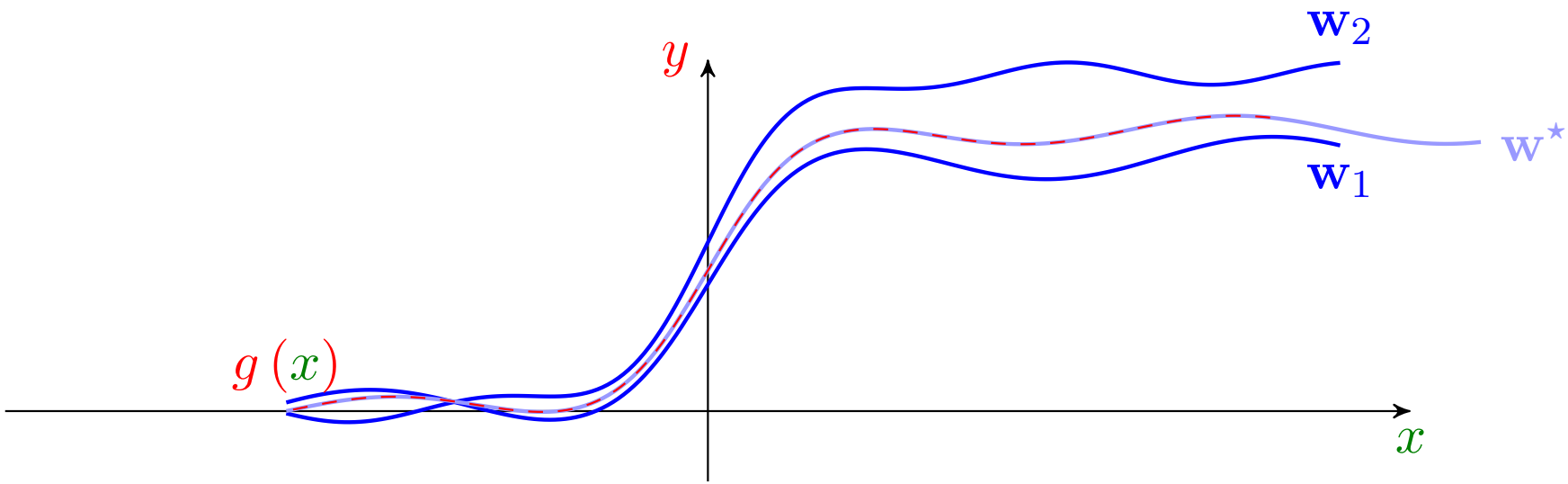
In reality, the labels and inputs are related through a function $g(\cdot)$

$$y = g(x)$$

The function $g(\cdot)$ is however unknown to us

In fact, our whole learning task is to learn it from the dataset!

+ *What if we knew function* $g(\cdot)$*? How would have trained our model?*
– Well! We would have tuned $\mathbf{w}$ until the model matches $g(\cdot)$



Training a Model

We could represent such $\mathbf{w}^\star$ *using the loss*: let $\mathcal{L}$ be the loss function, and assume that we have an input x_0. The model with learnable parameter $\mathbf{w}$ gives us the label $\hat{y}_0 = f\left(x_0|\mathbf{w}\right)$ that may be different from the true label $y_0 = g\left(x_0\right)$. The loss between the two is given by

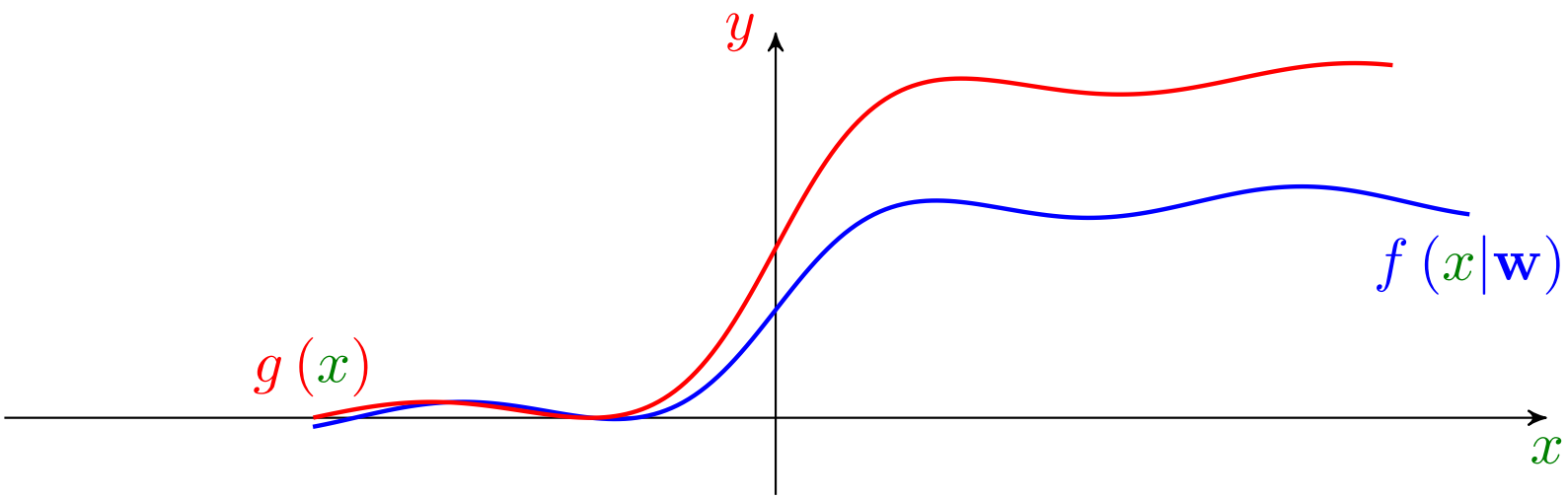
$$\ell\left(x_0|\mathbf{w}\right) = \mathcal{L}\left(\hat{y}_0, y_0\right) = \mathcal{L}\left(f\left(x_0|\mathbf{w}\right), g\left(x_0\right)\right)$$

Training a Model

We could represent such $\mathbf{w}^\star$ *using the loss*: let $\mathcal{L}$ be the loss function, and assume that we have an input x_0. The model with learnable parameter $\mathbf{w}$ gives us the label $\hat{y}_0 = f\left(x_0|\mathbf{w}\right)$ that may be different from the true label $y_0 = g\left(x_0\right)$. The loss between the two is given by

$$\ell\left(x_0|\mathbf{w}\right) = \mathcal{L}\left(\hat{y}_0, y_0\right) = \mathcal{L}\left(f\left(x_0|\mathbf{w}\right), g\left(x_0\right)\right)$$

Let's visualize this loss

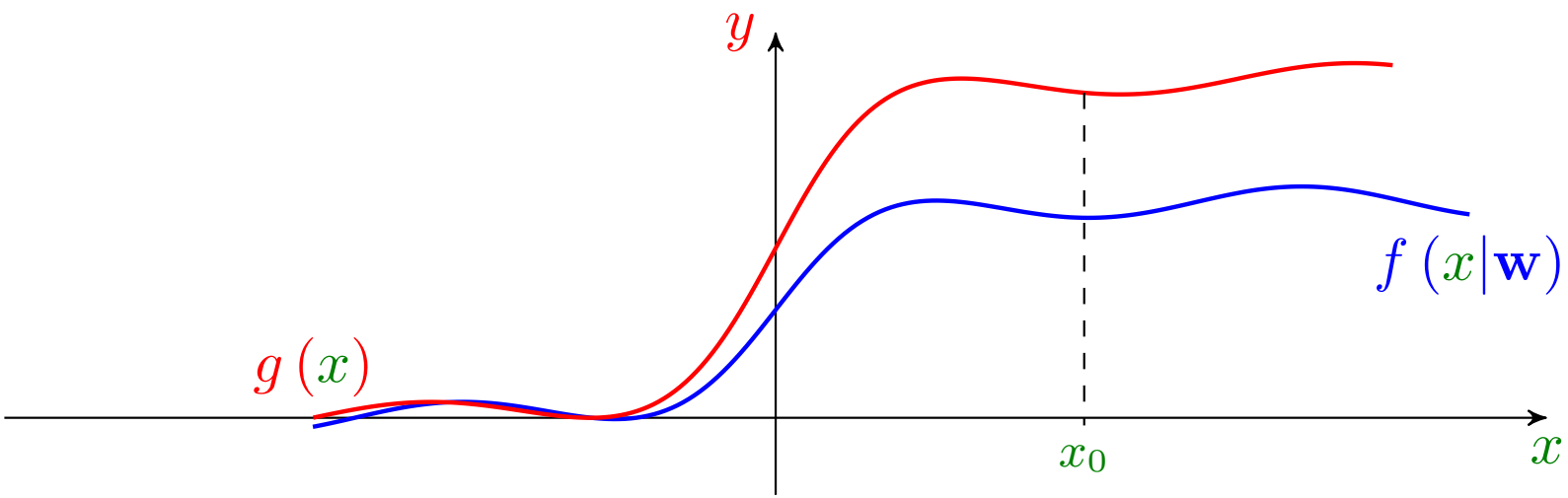


Training a Model

We could represent such $\mathbf{w}^\star$ *using the loss*: let $\mathcal{L}$ be the loss function, and assume that we have an input x_0. The model with learnable parameter $\mathbf{w}$ gives us the label $\hat{y}_0 = f\left(x_0|\mathbf{w}\right)$ that may be different from the true label $y_0 = g\left(x_0\right)$. The loss between the two is given by

$$\ell\left(x_0|\mathbf{w}\right) = \mathcal{L}\left(\hat{y}_0, y_0\right) = \mathcal{L}\left(f\left(x_0|\mathbf{w}\right), g\left(x_0\right)\right)$$

Let's visualize this loss

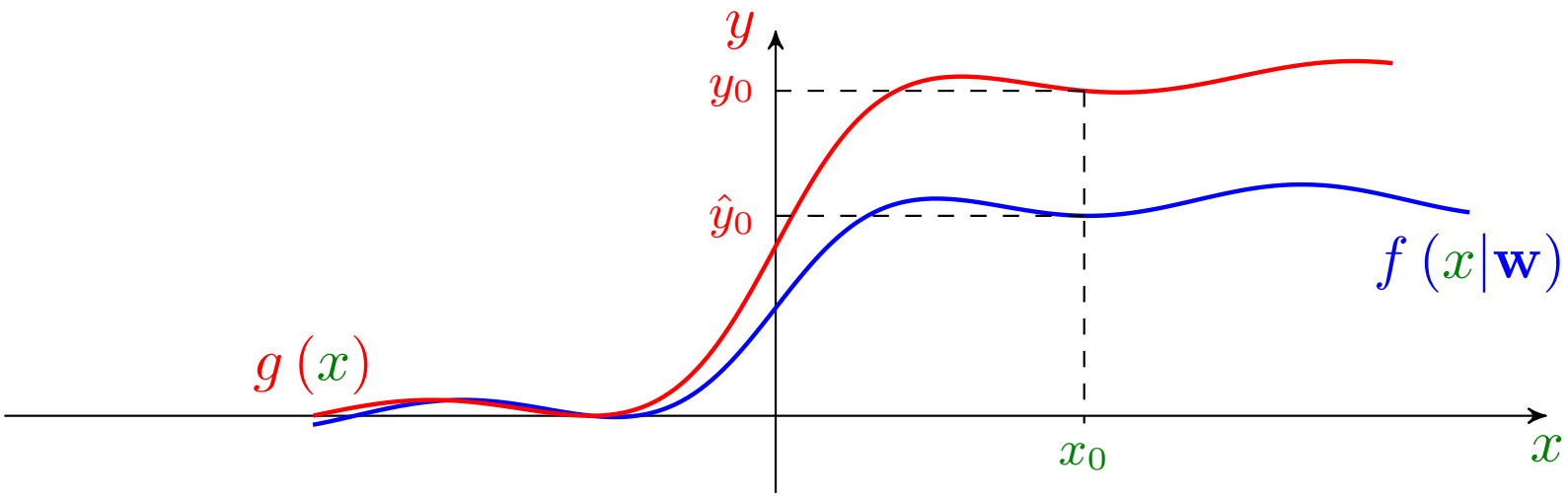


Training a Model

We could represent such $\mathbf{w}^\star$ using the loss: let $\mathcal{L}$ be the loss function, and assume that we have an input x_0. The model with learnable parameter $\mathbf{w}$ gives us the label $\hat{y}_0 = f\left(x_0|\mathbf{w}\right)$ that may be different from the true label $y_0 = g\left(x_0\right)$. The loss between the two is given by

$$\ell\left(x_0|\mathbf{w}\right) = \mathcal{L}\left(\hat{y}_0, y_0\right) = \mathcal{L}\left(f\left(x_0|\mathbf{w}\right), g\left(x_0\right)\right)$$

Let's visualize this loss

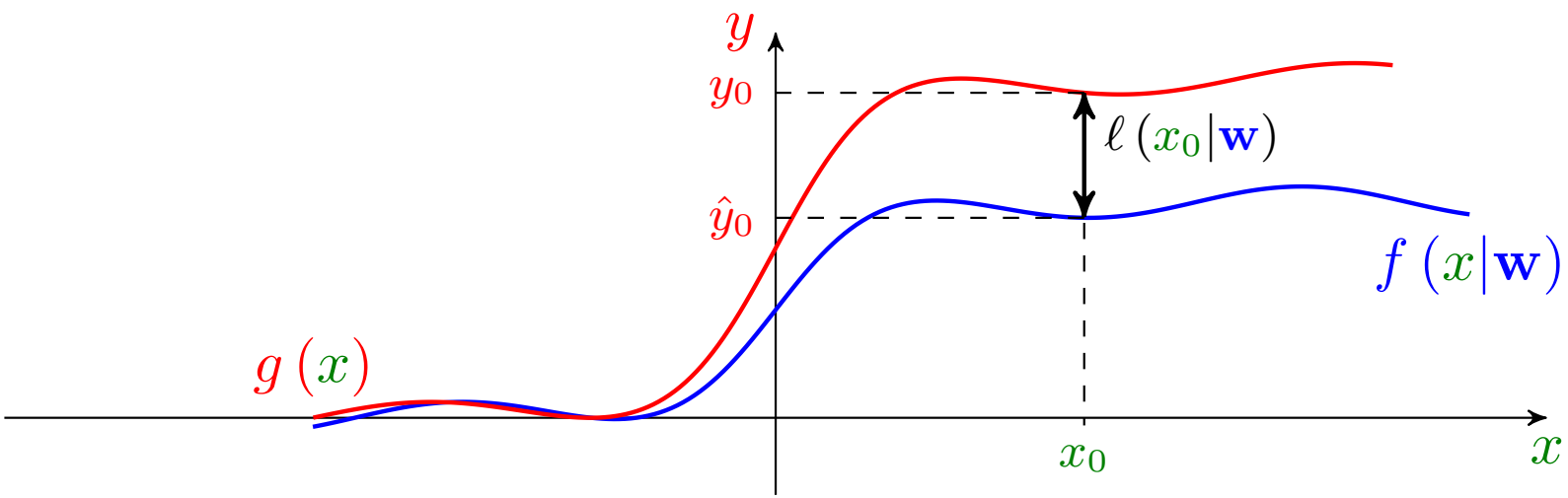


Training a Model

We could represent such $\mathbf{w}^\star$ using the loss: let $\mathcal{L}$ be the loss function, and assume that we have an input x_0. The model with learnable parameter $\mathbf{w}$ gives us the label $\hat{y}_0 = f\left(x_0|\mathbf{w}\right)$ that may be different from the true label $y_0 = g\left(x_0\right)$. The loss between the two is given by

$$\ell\left(x_0|\mathbf{w}\right) = \mathcal{L}\left(\hat{y}_0, y_0\right) = \mathcal{L}\left(f\left(x_0|\mathbf{w}\right), g\left(x_0\right)\right)$$

Let's visualize this loss



Training a Model: *Risk*

We would like our model to recover the true label as closely as possible: so, the best option at the particular point x_0 is to find the choice of $\mathbf{w}$ that minimizes $\ell\left(x_0|\mathbf{w}\right)$.

+ *But x_0 is not fixed! We cannot find $\mathbf{w}$ for a single value of x_0!*
– Right! We should learn $\mathbf{w}$ for any x_0
+ *How can we do it?*
– We treat x_0 as a random variable with some distribution $P\left(x_0\right)$, and minimize the *expexted loss* often called risk

Training a Model: *Risk*

We would like our model to recover the true label as closely as possible: so, the best option at the particular point x_0 is to find the choice of $\mathbf{w}$ that minimizes $\ell\left(x_0|\mathbf{w}\right)$.

+ *But x_0 is not fixed! We cannot find $\mathbf{w}$ for a single value of x_0!*
– Right! We should learn $\mathbf{w}$ for any x_0
+ *How can we do it?*
– We treat x_0 as a random variable with some distribution $P\left(x_0\right)$, and minimize the *expexted loss* often called risk

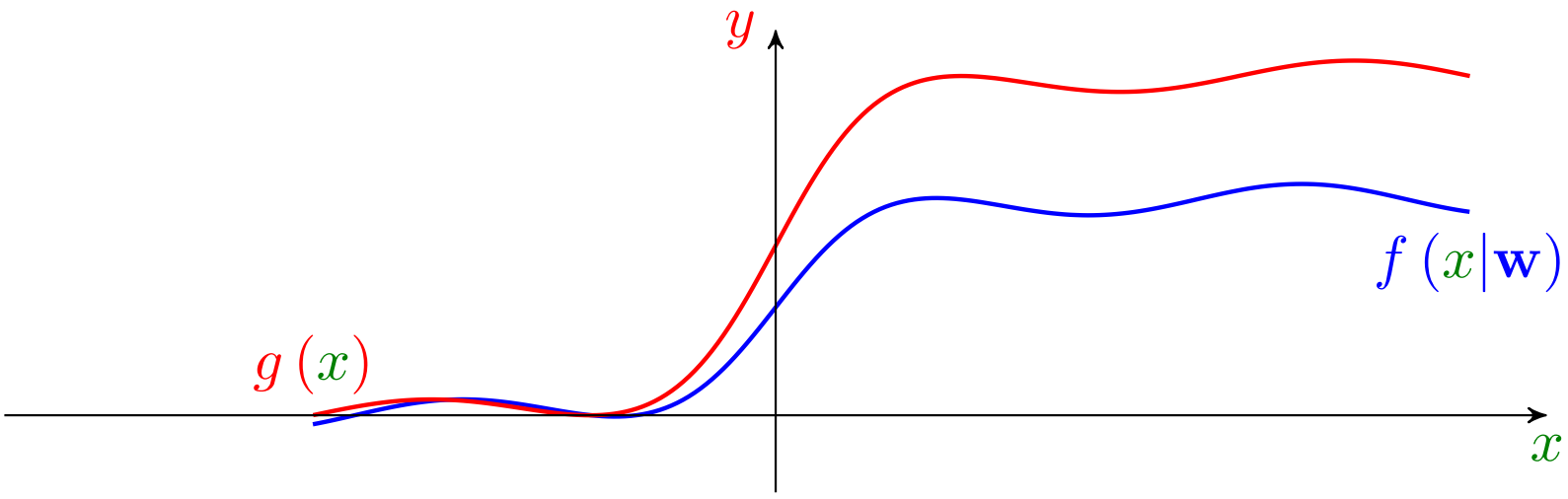
Risk

For given learnable parameter $\mathbf{w}$, the risk is defined as

$$R\left(\mathbf{w}\right) = \mathbb{E}\left\{\ell\left(x_0|\mathbf{w}\right)\right\} = \int \ell\left(x_0|\mathbf{w}\right) P\left(x_0\right) \mathrm{d}x_0$$

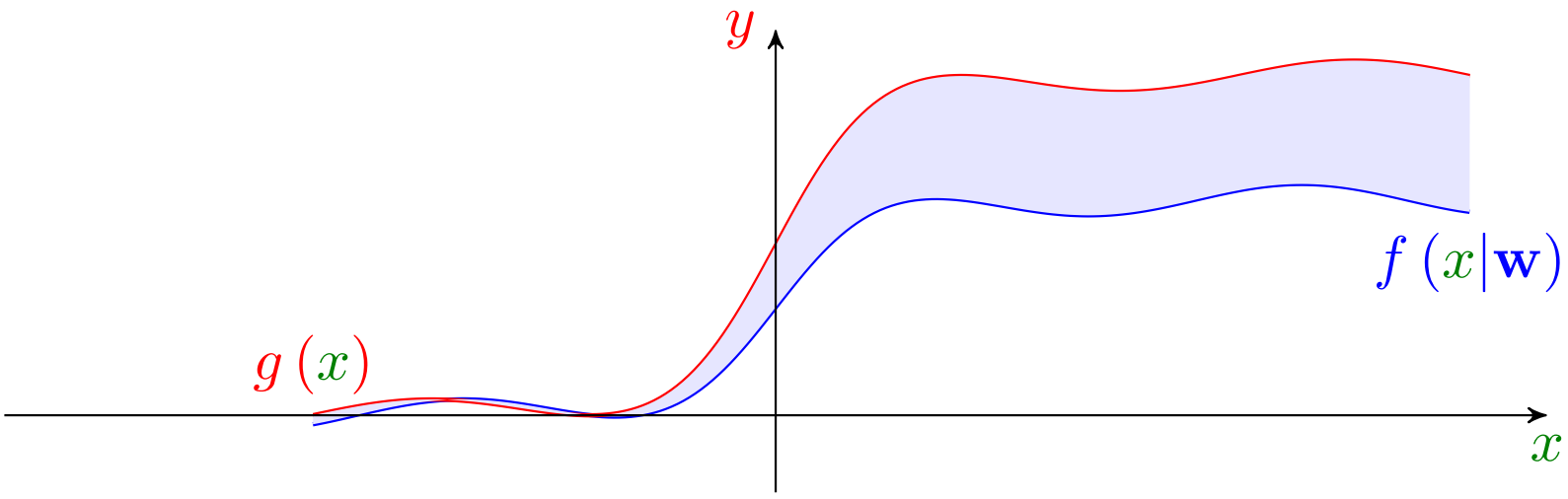
Training a Model: *Risk Minimization*

Let's visualize the risk



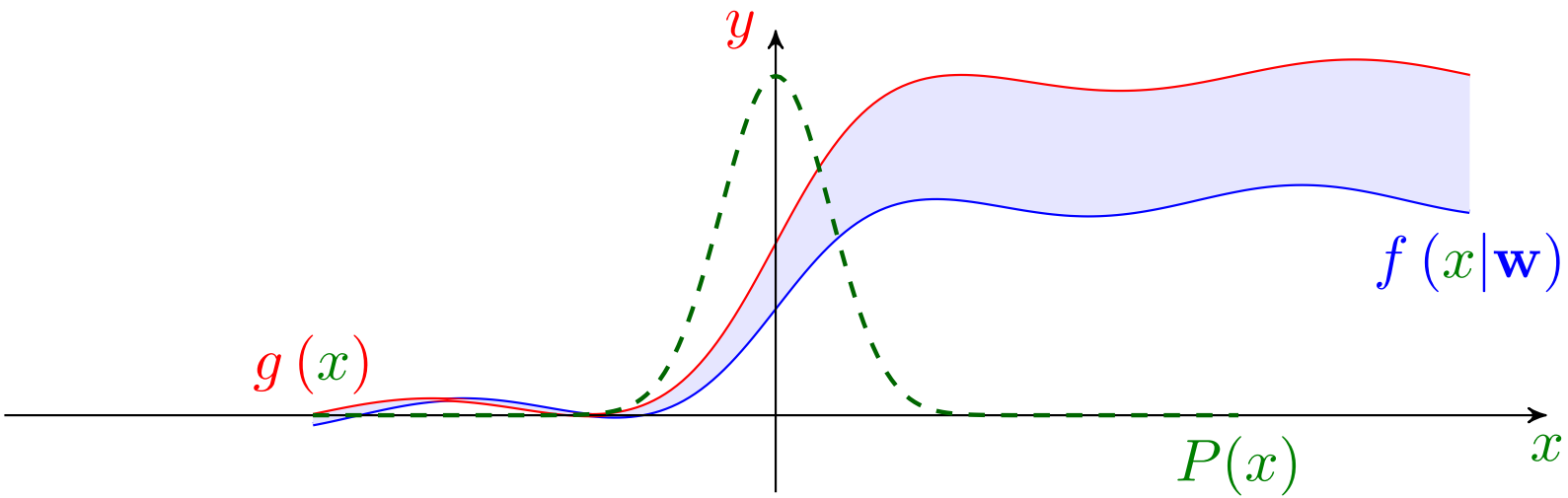
Training a Model: *Risk Minimization*

Let's visualize the risk



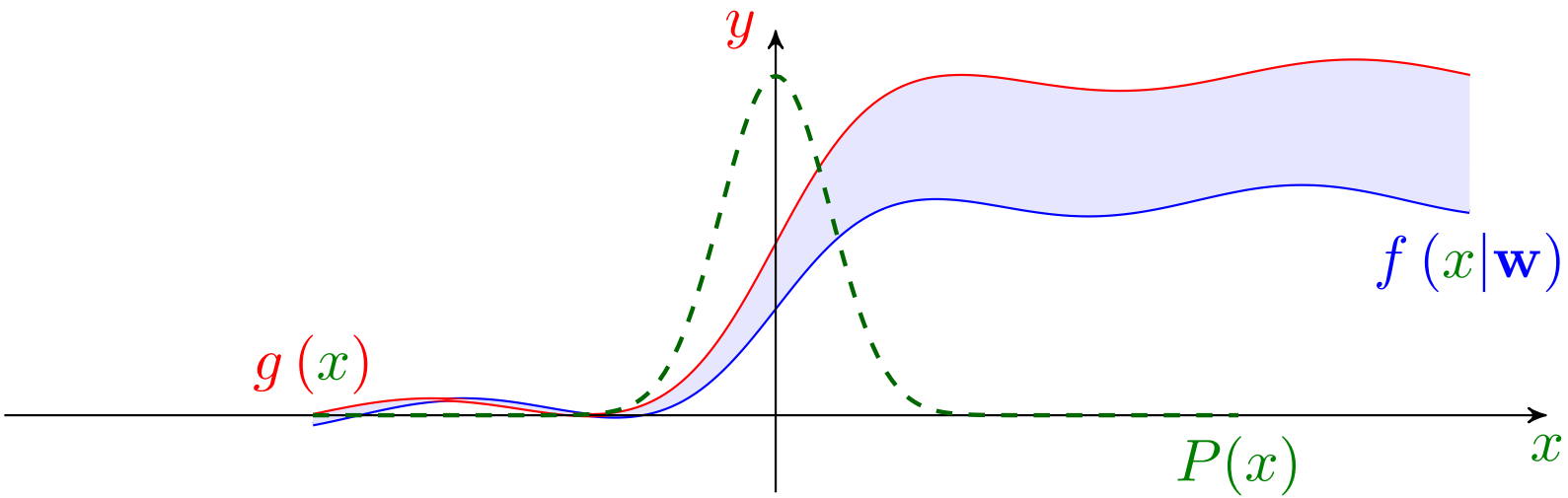
Training a Model: *Risk Minimization*

Let's visualize the risk



Training a Model: *Risk Minimization*

Let's visualize the risk



The training is then formulated as the minimization of risk

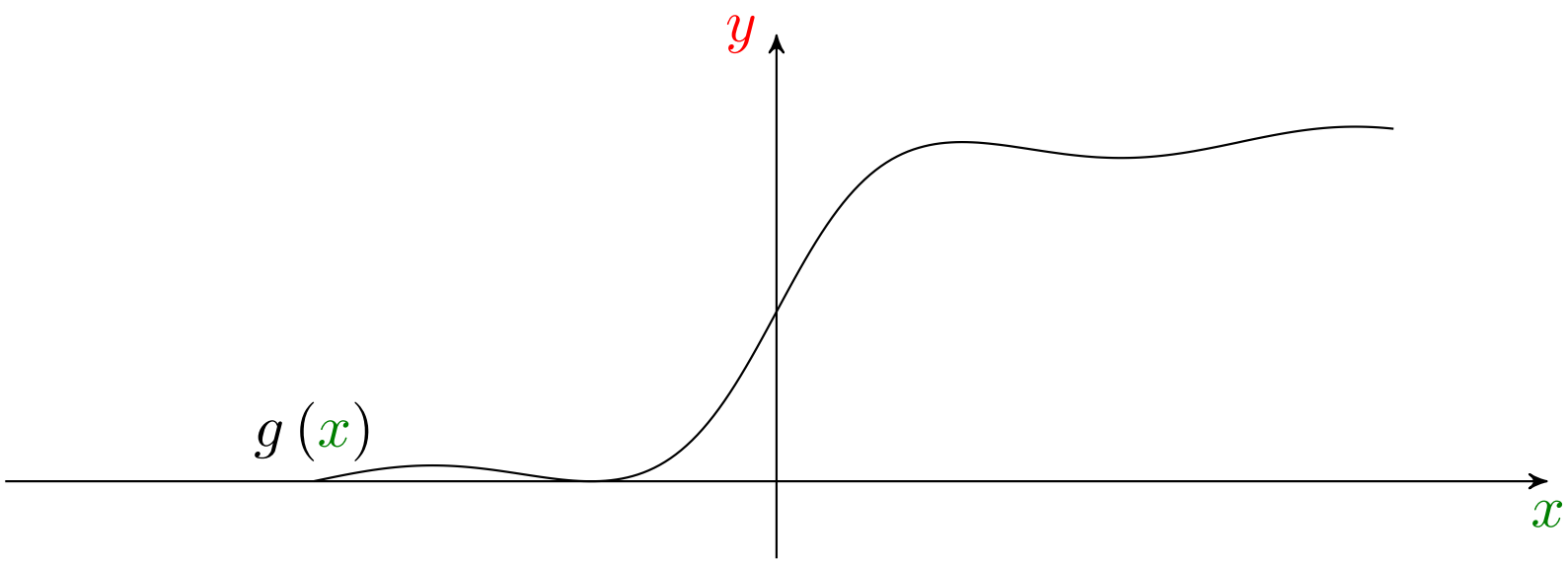
$$\mathbf{w}^{\star} = \underset{\mathbf{w}}{\operatorname{argmin}}\, R\left(\mathbf{w}\right)$$

Training a Model: *Empirical Risk*

+ *Bravo! But, the training seems infeasible, since we have neither the true function $g(\cdot)$, nor the distribution $P(x)$!*
– Right! But, we could handle this approximately using the LLN

Let's look at what we have: *the dataset which contains samples of $g(\cdot)$*

$$\mathbb{D} = \{(x_i, y_i) : i = 1, \ldots, I\}$$

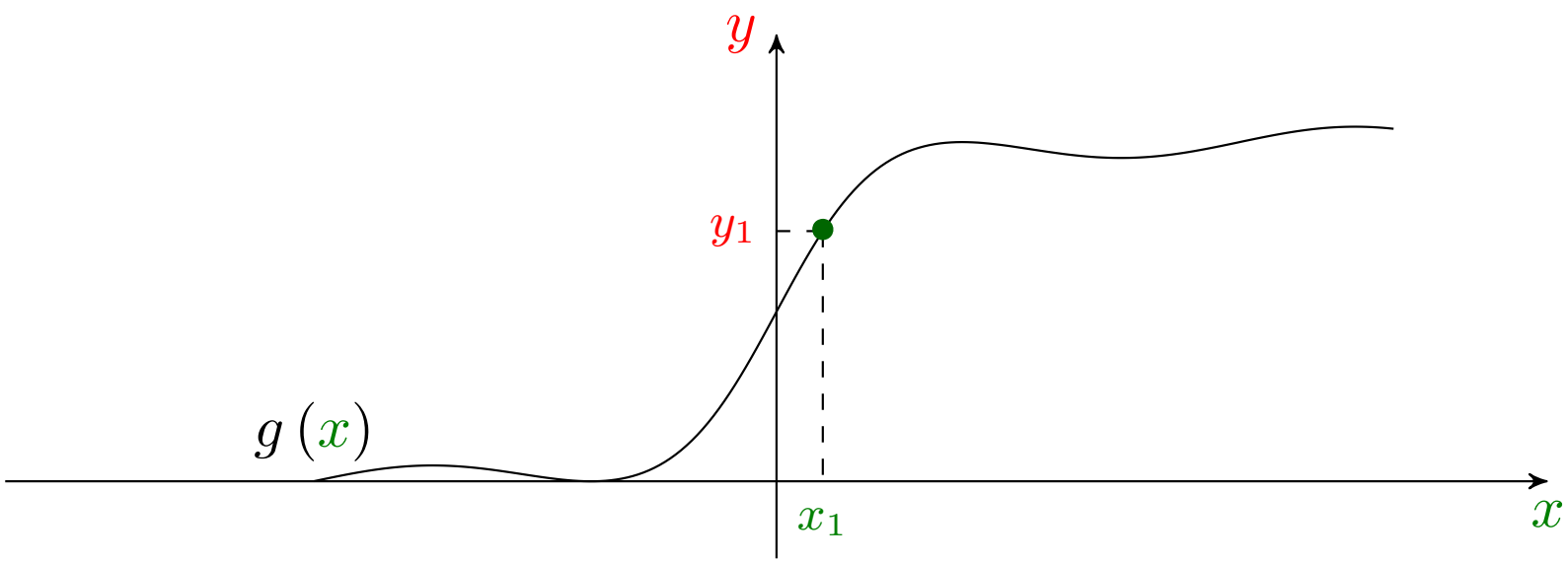


Training a Model: *Empirical Risk*

+ *Bravo! But, the training seems infeasible, since we have neither the true function $g(\cdot)$, nor the distribution $P(x)$!*
– Right! But, we could handle this approximately using the LLN

Let's look at what we have: *the dataset which contains samples of $g(\cdot)$*

$$\mathbb{D} = \{(x_i, y_i) : i = 1, \ldots, I\}$$

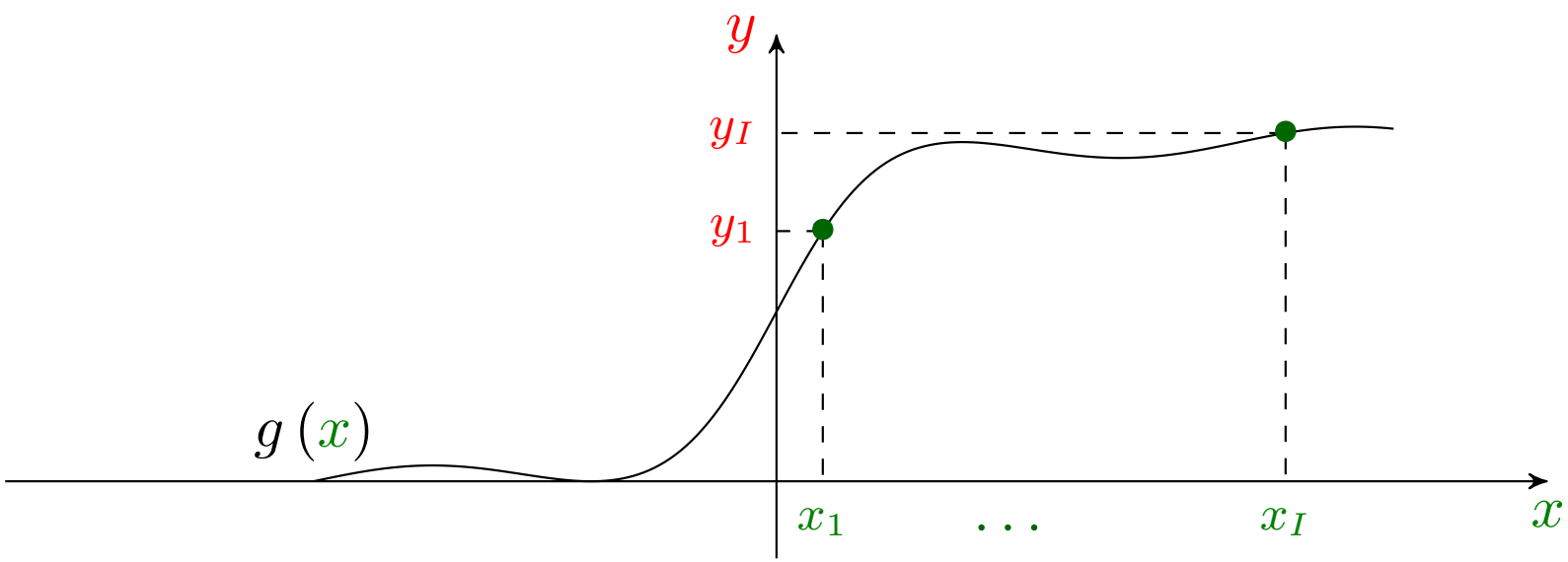


Training a Model: *Empirical Risk*

+ *Bravo! But, the training seems infeasible, since we have neither the true function $g(\cdot)$, nor the distribution $P(x)$!*
– Right! But, we could handle this approximately using the LLN

Let's look at what we have: *the dataset which contains samples of $g(\cdot)$*

$$\mathbb{D} = \{(x_i, y_i) : i = 1, \ldots, I\}$$



Training a Model: *Empirical Risk*

At these data-points, we can determine the loss

$$\ell\left(x_i|\mathbf{w}\right) = \mathcal{L}\left(f\left(x_i|\mathbf{w}\right), g\left(x_i\right)\right) = \mathcal{L}\left(f\left(x_i|\mathbf{w}\right), y_i\right)$$

Training a Model: *Empirical Risk*

At these data-points, we can determine the loss

$$\ell\left(x_i|\mathbf{w}\right) = \mathcal{L}\left(f\left(x_i|\mathbf{w}\right), g\left(x_i\right)\right) = \mathcal{L}\left(f\left(x_i|\mathbf{w}\right), y_i\right)$$

Now, we assume that the data-points are *independently drawn from the distribution* $P\left(x\right)$: the LLN then suggests that *the arithmetic average of losses converges to the risk* when we have a *large enough number of data-points*, i.e., as we increase I

$$\frac{1}{I}\sum_{i=1}^{I}\ell\left(x_i|\mathbf{w}\right) \rightarrow \mathbb{E}\left\{\ell\left(x|\mathbf{w}\right)\right\} = R\left(\mathbf{w}\right)$$

We call this arithmetic average the empirical risk

Empirical risk is the best estimate of risk that we get from our dataset

Training a Model: *Empirical Risk Minimization*

Empirical Risk

Let $\mathbf{w}$ includes all learnable parameters of the model, and the dataset be

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

for loss function $\mathcal{L}$, the empirical risk is defined as

$$\hat{R}(\mathbf{w}) = \frac{1}{I} \sum_{i=1}^{I} \mathcal{L}\left(f\left(\boldsymbol{x}_i | \mathbf{w}\right), y_i\right)$$

Training a Model: *Empirical Risk Minimization*

Empirical Risk

Let $\mathbf{w}$ includes all learnable parameters of the model, and the dataset be

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

for loss function $\mathcal{L}$, the empirical risk is defined as

$$\hat{R}(\mathbf{w}) = \frac{1}{I} \sum_{i=1}^{I} \mathcal{L}\left(f\left(\boldsymbol{x}_i | \mathbf{w}\right), y_i\right)$$

The training is performed by minimizing the empirical risk

$$\mathbf{w}^{\star} = \underset{\mathbf{w}}{\operatorname{argmin}}\, \hat{R}(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}}\, \frac{1}{I} \sum_{i=1}^{I} \mathcal{L}\left(f\left(\boldsymbol{x}_i | \mathbf{w}\right), y_i\right) \qquad \text{(Training)}$$

Wrap-Up: *Universal Scheme for ML*

So, we know pretty much all the theory for supervised learning:

1. We build the main three components:
   1. Dataset
   2. Model
   3. Loss Function
2. We determine the empirical risk of the model using loss function
3. We train the model by minimizing the empirical risk

Wrap-Up: *Universal Scheme for ML*

So, we know pretty much all the theory for supervised learning:

1. We build the main three components:
   1. Dataset
   2. Model
   3. Loss Function
2. We determine the empirical risk of the model using loss function
3. We train the model by minimizing the empirical risk

We call this universal approach ML 1-2-3!

The theory is pretty short; however,

how to execute each step of ML 1-2-3 is a pretty long story

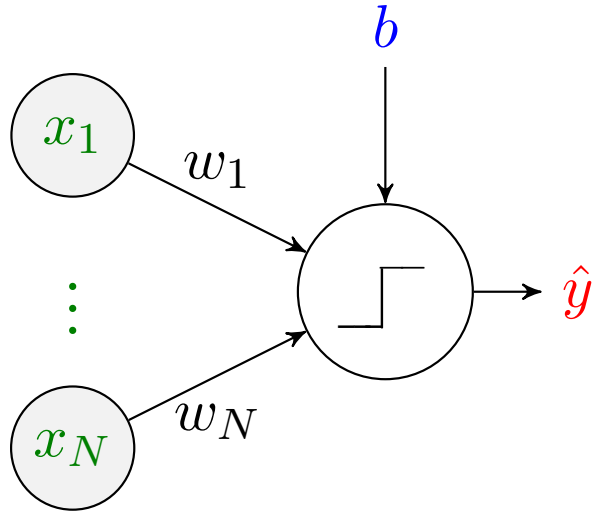
that we are going to learn in the remaining of this course!

Training Perceptron Machine: *ML 1*

Let's start our long story with binary classification via perceptron

1. Dataset $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$ with binary labels $y_i \in \{0, 1\}$
2. Perceptron as the model | Binary classifier |

$$\hat{y} = s(\mathbf{w}^\mathsf{T}\boldsymbol{x} + b) = \begin{cases} 1 & \text{if } \mathbf{w}^\mathsf{T}\boldsymbol{x} + b \geqslant 0 \\ 0 & \text{if } \mathbf{w}^\mathsf{T}\boldsymbol{x} + b < 0 \end{cases}$$



3. Error indicator as the loss function

$$\mathcal{L}(\hat{y}, y) = \mathbb{1}(\hat{y} \neq y) = \begin{cases} 1 & \hat{y} \neq y \\ 0 & \hat{y} = y \end{cases}$$

Training Perceptron Machine: *ML 2*

Let's write down the empirical risk

$$\begin{aligned}\hat{R}(\mathbf{w}, b) &= \frac{1}{I}\sum_{i=1}^{I}\mathcal{L}\left(f\left(\boldsymbol{x}_i|\mathbf{w}, b\right), y_i\right)\\ &= \frac{1}{I}\sum_{i=1}^{I}\mathbb{1}\left(s(\mathbf{w}^{\mathsf{T}}\boldsymbol{x}_i + b) \neq y_i\right)\\ &= \textit{Fraction of data-points mis-classified by perceptron}\\ &\equiv \textit{Error Rate}\end{aligned}$$

Training Perceptron Machine: *ML 3*

We should now minimize the empirical risk

$$\mathbf{w}^\star, b^\star = \underset{\mathbf{w}\in\mathbb{R}^N,\, b\,\in\mathbb{R}}{\operatorname{argmin}} \hat{R}\left(\mathbf{w}, b\right) = \underset{\mathbf{w}\in\mathbb{R}^N,\, b\,\in\mathbb{R}}{\operatorname{argmin}} \frac{1}{I}\sum_{i=1}^{I} \mathbb{1}\left(s(\mathbf{w}^\intercal \boldsymbol{x}_i + b) \neq y_i\right)$$

But, how can we solve this optimization? It doesn't look easy!

Training Perceptron Machine: *ML 3*

We should now minimize the empirical risk

$$\mathbf{w}^\star, b^\star = \underset{\mathbf{w}\in\mathbb{R}^N,\, b\,\in\mathbb{R}}{\operatorname{argmin}} \hat{R}\left(\mathbf{w}, b\right) = \underset{\mathbf{w}\in\mathbb{R}^N,\, b\,\in\mathbb{R}}{\operatorname{argmin}} \frac{1}{I}\sum_{i=1}^{I} \mathbb{1}\left(s(\mathbf{w}^\mathsf{T}\boldsymbol{x}_i + b) \neq y_i\right)$$

But, how can we solve this optimization? It doesn't look easy!

Let's see what Rosenblatt did: *Rosenblatt's perceptron had no bias, i.e.,* $b = 0$

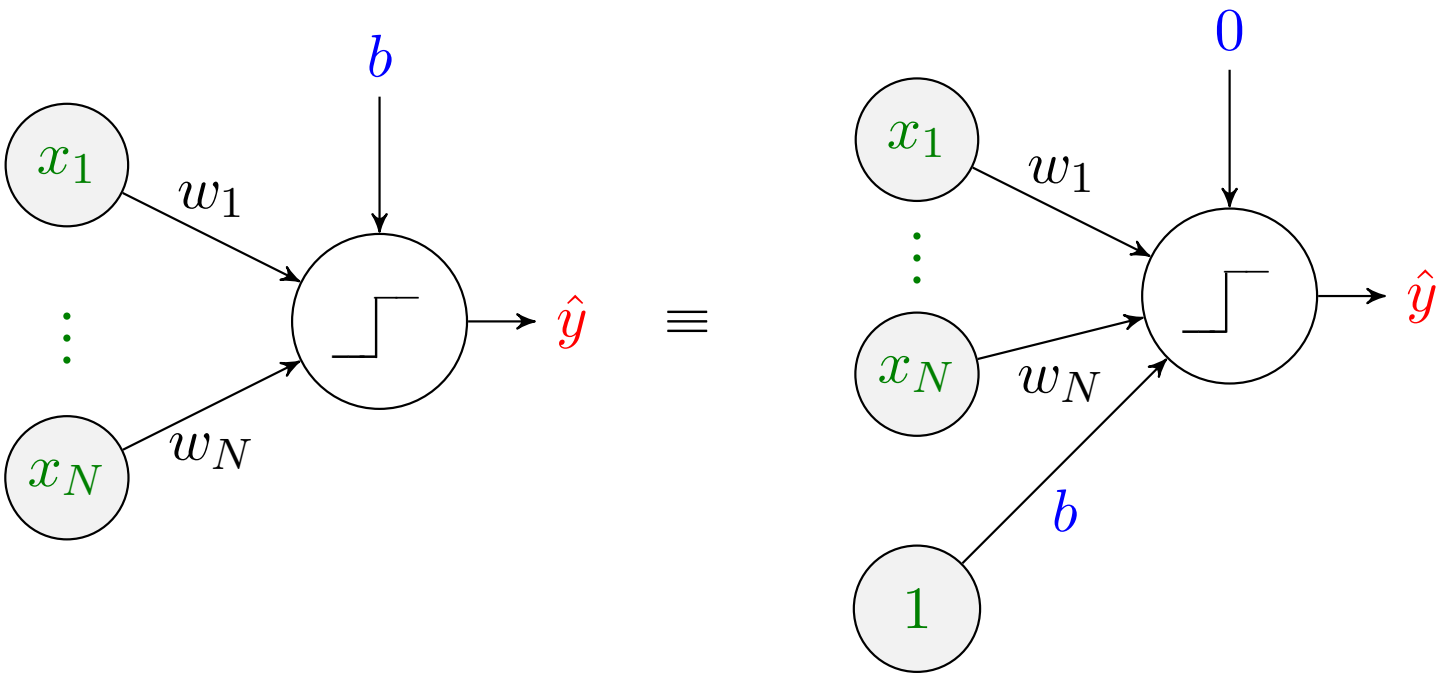
1: Start with $\mathbf{w} = \mathbf{0}$ or some small random initial value
2: **while** $\hat{R}\left(\mathbf{w}\right) \neq 0$ **do**
3: **for** $i = 1 : I$ **do**
4: Determine $z_i = \mathbf{w}^\mathsf{T}\boldsymbol{x}_i$
5: **if** $s(z_i) \neq y_i$ **then**
6: $\mathbf{w} \leftarrow \mathbf{w} - \text{sign}\left(z_i\right)\boldsymbol{x}_i$
7: **end if**
8: **end for**
9: **end while**

Training Perceptron Machine: *ML 3*

It's almost zero effort to extend Rosenblatt's algorithm to the case with bias

$$\mathbf{w}^\mathsf{T}\boldsymbol{x}_i + b = [\mathbf{w}^\mathsf{T}, b]\begin{bmatrix}\boldsymbol{x}_i \\ 1\end{bmatrix} = \tilde{\mathbf{w}}^\mathsf{T}\tilde{\boldsymbol{x}}_i$$

So, bias only adds one dimension to the data-point with value 1



Training Perceptron Machine: *Perceptron Algorithm*

So, we just need to replace $\mathbf{w}$ with $\begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}$ in Rosenblatt's algorithm

1: Start with $\mathbf{w} = \mathbf{0}$ and $b = 0$, or some small random initial value
2: **while** $\hat{R}(\mathbf{w}, b) \neq 0$ **do**
3: **for** $i = 1 : I$ **do**
4: Determine $z_i = \mathbf{w}^\mathsf{T} \boldsymbol{x}_i + b$
5: **if** $s(z_i) \neq y_i$ **then**
6: $\mathbf{w} \leftarrow \mathbf{w} - \text{sign}(z_i)\, \boldsymbol{x}_i$ and $b \leftarrow b - \text{sign}(z_i)$
7: **end if**
8: **end for**
9: **end while**

+ *Why should perceptron algorithm minimize the empirical risk?*
– Let's inspect the no-bias version step by step

Training Perceptron Machine: *Perceptron Algorithm*

1: Start with $\mathbf{w} = \mathbf{0}$ or some small random initial value
2: **while** $\hat{R}(\mathbf{w}) \neq 0$ **do**
3: **for** $i = 1 : I$ **do**
4: Determine $z_i = \mathbf{w}^\mathsf{T} \boldsymbol{x}_i$
5: **if** $s(z_i) \neq y_i$ **then**
6: $\mathbf{w} \leftarrow \mathbf{w} - \text{sign}(z_i)\, \boldsymbol{x}_i$
7: **end if**
8: **end for**
9: **end while**

- Outer loop stops only if $\hat{R}(\mathbf{w}) = 0$ which is minimum empirical risk
- In inner loop, let's say at iteration t error occurs for $\boldsymbol{x}_i$
  - ↳ either $z_i = \mathbf{w}_t^\mathsf{T} \boldsymbol{x}_i > 0$ and $y_i = 0$: *we update as* $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \boldsymbol{x}_i$

$$\mathbf{w}_{t+1}^\mathsf{T} \boldsymbol{x}_i = (\mathbf{w}_t - \boldsymbol{x}_i)^\mathsf{T} \boldsymbol{x}_i = \mathbf{w}_t^\mathsf{T} \boldsymbol{x}_i - \|\boldsymbol{x}_i\|^2 \leqslant \mathbf{w}_t^\mathsf{T} \boldsymbol{x}_i$$

so the algorithm pushes the weights somewhere that z_i *could get negative*

Training Perceptron Machine: *Perceptron Algorithm*

1: Start with $\mathbf{w} = \mathbf{0}$ or some small random initial value
2: **while** $\hat{R}(\mathbf{w}) \neq 0$ **do**
3: **for** $i = 1 : I$ **do**
4: Determine $z_i = \mathbf{w}^\mathsf{T} \boldsymbol{x}_i$
5: **if** $s(z_i) \neq y_i$ **then**
6: $\mathbf{w} \leftarrow \mathbf{w} - \text{sign}(z_i)\, \boldsymbol{x}_i$
7: **end if**
8: **end for**
9: **end while**

- Outer loop stops only if $\hat{R}(\mathbf{w}) = 0$ which is minimum empirical risk
- In inner loop, let's say at iteration t error occurs for $\boldsymbol{x}_i$
  - ↳ or $z_i = \mathbf{w}_t^\mathsf{T} \boldsymbol{x}_i < 0$ and $y_i = 1$: *we update as* $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \boldsymbol{x}_i$

$$\mathbf{w}_{t+1}^\mathsf{T} \boldsymbol{x}_i = (\mathbf{w}_t + \boldsymbol{x}_i)^\mathsf{T} \boldsymbol{x}_i = \mathbf{w}_t^\mathsf{T} \boldsymbol{x}_i + \|\boldsymbol{x}_i\|^2 \geqslant \mathbf{w}_t^\mathsf{T} \boldsymbol{x}_i$$

so the algorithm pushes the weights somewhere that z_i *could get positive*

Training Perceptron Machine: *Perceptron Algorithm*

1: Start with $\mathbf{w} = \mathbf{0}$ or some small random initial value
2: **while** $\hat{R}(\mathbf{w}) \neq 0$ **do**
3: **for** $i = 1 : I$ **do**
4: Determine $z_i = \mathbf{w}^{\mathsf{T}} \boldsymbol{x}_i$
5: **if** $s(z_i) \neq y_i$ **then**
6: $\mathbf{w} \leftarrow \mathbf{w} - \text{sign}(z_i)\, \boldsymbol{x}_i$
7: **end if**
8: **end for**
9: **end while**

+ *It makes sense that in each iteration* $\mathbf{w}$ *gets modified towards a right direction! But can we guarantee that this algorithm always converges? In other words, can't it get into an infinity loop?*
– Well! Let's try some examples

Perceptron Algorithm: *AND Operator*

Assume that we have the following dataset two-dimensional inputs

$$\mathbb{D} = \left\{ \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}, 1 \right) \right\}$$

We intend to train perceptron with this dataset via perceptron algorithm

Perceptron Algorithm: *AND Operator*

Assume that we have the following dataset two-dimensional inputs

$$\mathbb{D} = \left\{ \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}, 1 \right) \right\}$$

We intend to train perceptron with this dataset via perceptron algorithm

---

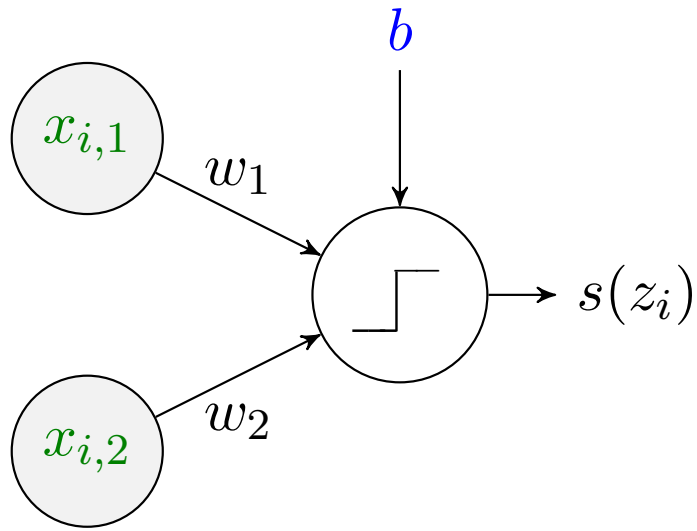
Before we start training, we note that this dataset represents the AND operator

$$\forall \left(\boldsymbol{x}_i, y_i \right) \in \mathbb{D} : y_i = x_{i,1} \wedge x_{i,2}$$

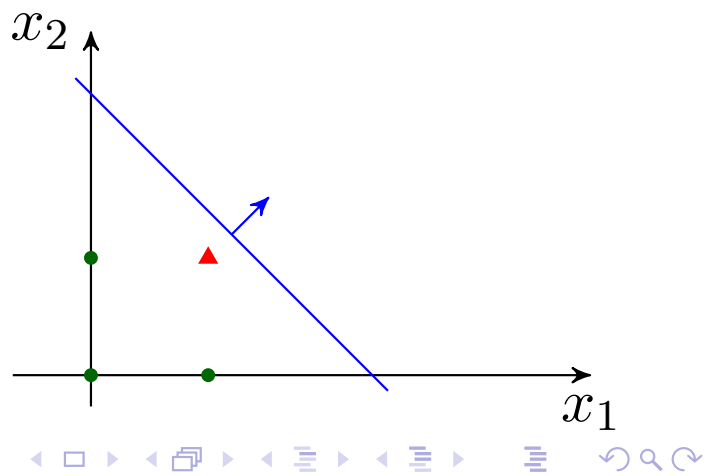
so, we basically want to see, if we could realize this operator via perceptron

Perceptron Algorithm: *AND Operator*

1: Start with $\mathbf{w} = [1.1, 1.1]^{\mathsf{T}}$ and $b = -2.25$: `blue`
2: **while** $\hat{R}(\mathbf{w}, b) \neq 0$ **do**
3: **for** $i = 1 : I$ **do**
4: Determine $z_i = \mathbf{w}^{\mathsf{T}} \boldsymbol{x}_i + b$
5: **if** $s(z_i) \neq y_i$ **then**
6: $\mathbf{w} \leftarrow \mathbf{w} - \text{sign}(z_i)\, \boldsymbol{x}_i$ and $b \leftarrow b - \text{sign}(z_i)$
7: **end if**
8: **end for**
9: **end while**

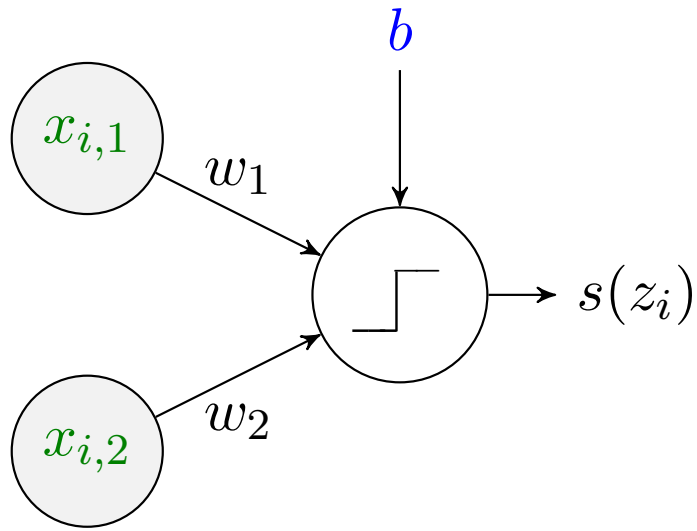


Let's show data-points with $y_i = 1$ by ▲ and those with $y_i = 0$ by ●



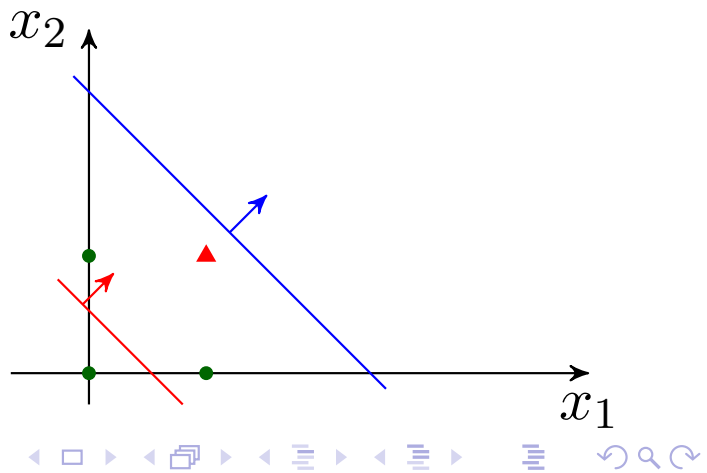
Perceptron Algorithm: *AND Operator*

1: Start with $\mathbf{w} = [1.1, 1.1]^\mathsf{T}$ and $b = -2.25$: `blue`
2: **while** $\hat{R}(\mathbf{w}, b) \neq 0$ **do**
3: **for** $i = 1 : I$ **do**
4: Determine $z_i = \mathbf{w}^\mathsf{T} \boldsymbol{x}_i + b$
5: **if** $s(z_i) \neq y_i$ **then**
6: $\mathbf{w} \leftarrow \mathbf{w} - \text{sign}(z_i)\, \boldsymbol{x}_i$ and $b \leftarrow b - \text{sign}(z_i)$
7: **end if**
8: **end for**
9: **end while**



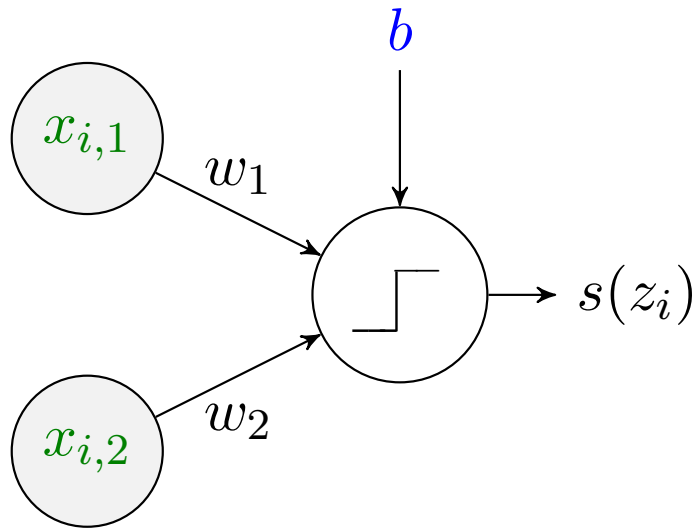
Let's show data-points with $y_i = 1$ by ▲ and those with $y_i = 0$ by ●

- Updated by $\boldsymbol{x}_i = [1, 1]^\mathsf{T}$
  - ↳ $\mathbf{w} = [2.1, 2.1]^\mathsf{T}$ and $b = -1.25$: `red`



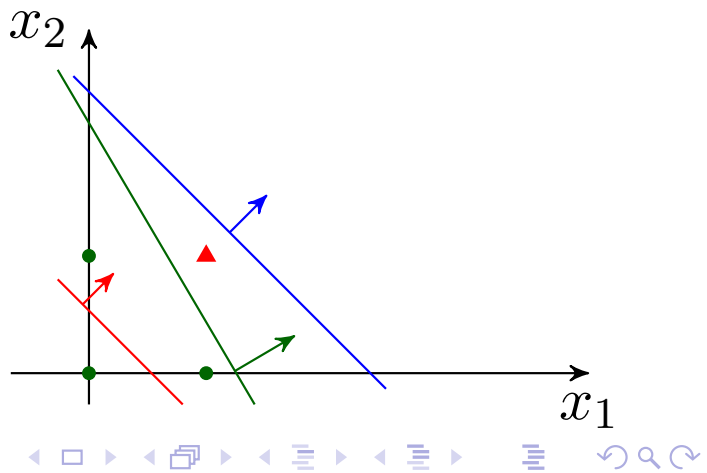
Perceptron Algorithm: *AND Operator*

1: Start with $\mathbf{w} = [1.1, 1.1]^\mathsf{T}$ and $b = -2.25$: `blue`
2: **while** $\hat{R}(\mathbf{w}, b) \neq 0$ **do**
3: **for** $i = 1 : I$ **do**
4: Determine $z_i = \mathbf{w}^\mathsf{T} \boldsymbol{x}_i + b$
5: **if** $s(z_i) \neq y_i$ **then**
6: $\mathbf{w} \leftarrow \mathbf{w} - \text{sign}(z_i)\, \boldsymbol{x}_i$ and $b \leftarrow b - \text{sign}(z_i)$
7: **end if**
8: **end for**
9: **end while**



Let's show data-points with $y_i = 1$ by ▲ and those with $y_i = 0$ by ●

- Updated by $\boldsymbol{x}_i = [1, 1]^\mathsf{T}$
  - ↳ $\mathbf{w} = [2.1, 2.1]^\mathsf{T}$ and $b = -1.25$: `red`
- Updated by $\boldsymbol{x}_i = [0, 1]^\mathsf{T}$
  - ↳ $\mathbf{w} = [2.1, 1.1]^\mathsf{T}$ and $b = -2.25$: `green`



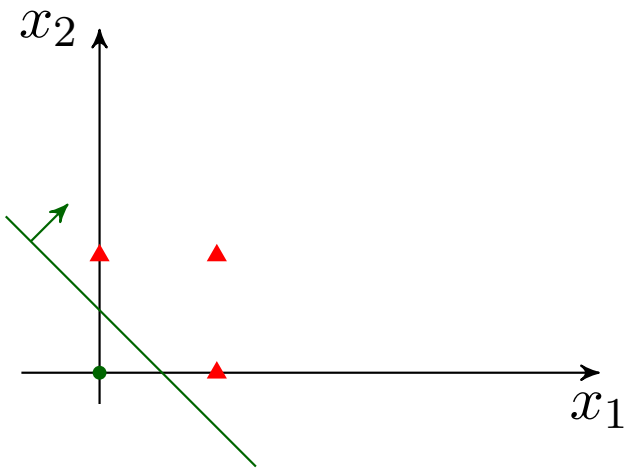
Perceptron Algorithm: *AND Operator*

Great! We trained the perceptron to behave as logical AND

- \+ *What about logical OR?*
- – Easy! Let's write the dataset

$$\mathbb{D} = \left\{ \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}, 1 \right) \right\}$$

Trying the perceptron algorithm, we end up with some linear classifier like

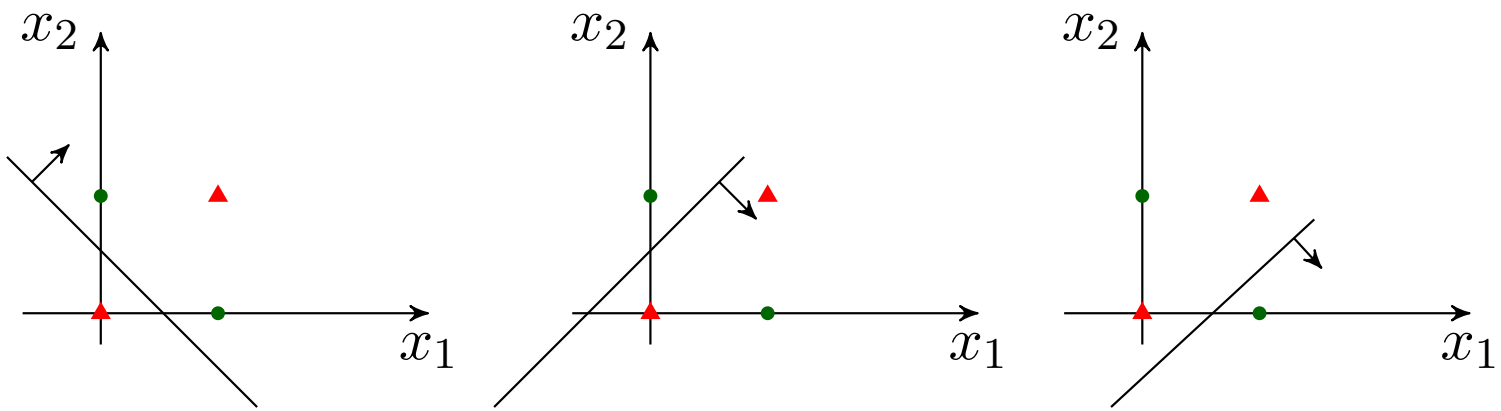


Perceptron Algorithm: *False Conclusion*

A false initial conclusion is that *perceptron can realize any binary function!*
But, it is easy to see that *this is not the case!* Let's consider logical XOR:

$$x_1 \oplus x_2 = \begin{cases} 1 & \text{if } x_1 \neq x_2 \\ 0 & \text{if } x_1 = x_2 \end{cases} \rightsquigarrow \mathbb{D} = \left\{ \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}, 0 \right) \right\}$$

It's not hard to see that perceptron cannot learn this function[3]



[3] If you don't see it clearly, no worries! You'll show it as an assignment

Perceptron Algorithm: *Linearly Separable Functions*

+ *What happens if we try the perceptron algorithm on this dataset?*
– It will iterate for ever!
+ *Why does this happen?*
– This is because XOR is not linearly separable

Perceptron can classify only linearly separable functions

At this point, it was concluded that perceptron should be replaced by some other model in order to learn nonlinear function

Finding models that learn nonlinear function led to the birth of

Representation Learning

But, we don't need to study it, since deep learning solved the problem!

Multi-Layering Perceptrons

Let's play a bit with logical XOR. Recall that

$$x_1 \bigoplus x_2 = \begin{cases} 1 & \text{if } x_1 \neq x_2 \\ 0 & \text{if } x_1 = x_2 \end{cases}$$

We can write this function as

$$x_1 \bigoplus x_2 = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$$

where we have used the following notation

$$\bar{x} \equiv \text{complement of } x = 1 - x \qquad \text{and} \qquad \vee \equiv \text{logical OR}$$

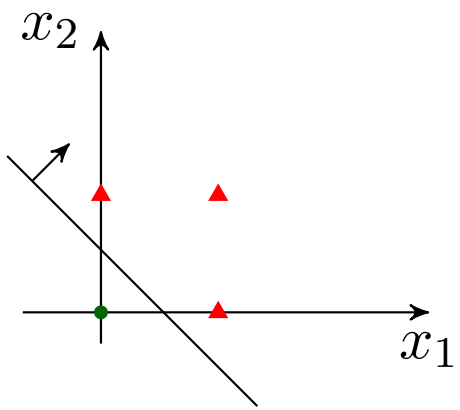
Multi-Layering Perceptrons

Logical XOR is expanded as

$$y = x_1 \oplus x_2 = \underbrace{(x_1 \vee x_2)}_{h_1} \wedge \underbrace{(\bar{x}_1 \vee \bar{x}_2)}_{h_2} = h_1 \wedge h_2$$

We can learn h_1 and h_2 by two different perceptrons

$$h_1 = x_1 \vee x_2$$



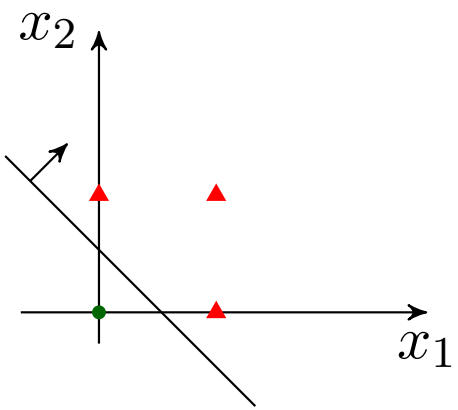
Multi-Layering Perceptrons

Logical XOR is expanded as

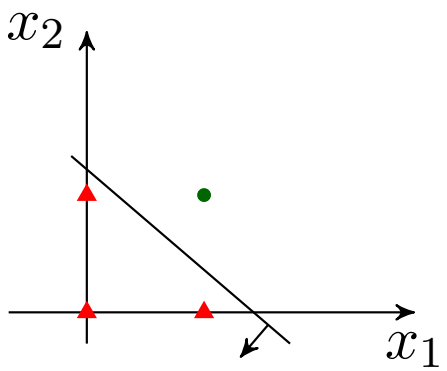
$$y = x_1 \oplus x_2 = \underbrace{(x_1 \vee x_2)}_{h_1} \wedge \underbrace{(\bar{x}_1 \vee \bar{x}_2)}_{h_2} = h_1 \wedge h_2$$

We can learn h_1 and h_2 by two different perceptrons

$$h_1 = x_1 \vee x_2$$

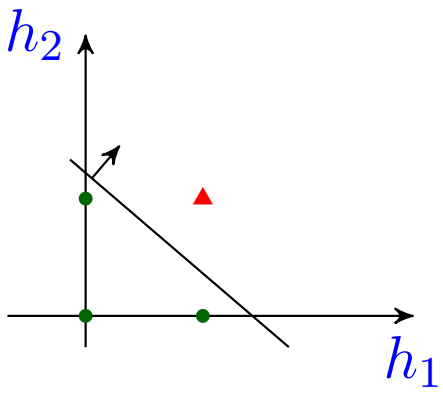


$$h_2 = \bar{x}_1 \vee \bar{x}_2 \equiv \overline{x_1 \wedge x_2}$$



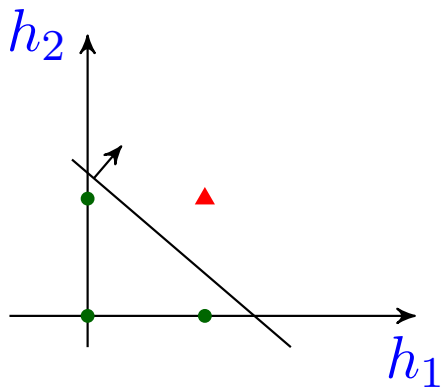
Multi-Layering Perceptrons

We can further learn $y = h_1 \wedge h_2$ *by another perceptron*

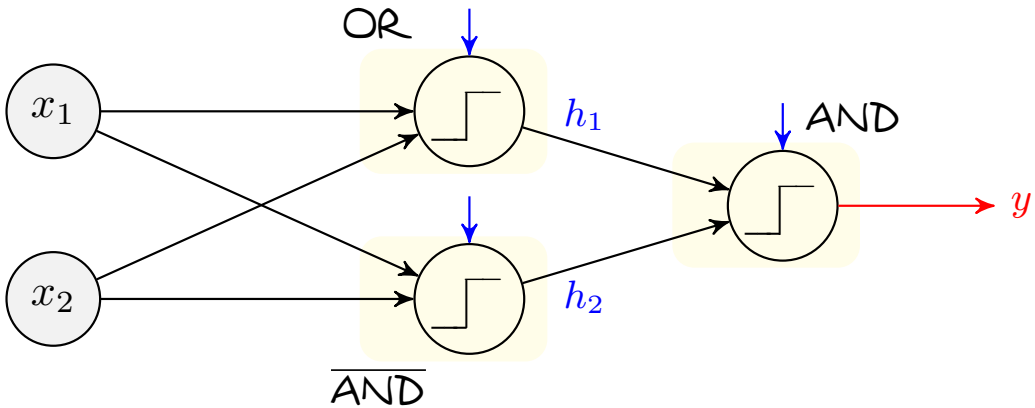


Multi-Layering Perceptrons

We can further learn $y = h_1 \wedge h_2$ by another perceptron

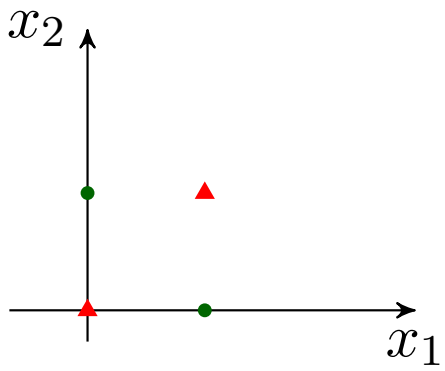


Well! It's true that we cannot learn XOR by a single perceptron, but we can learn it with a network of three perceptrons!



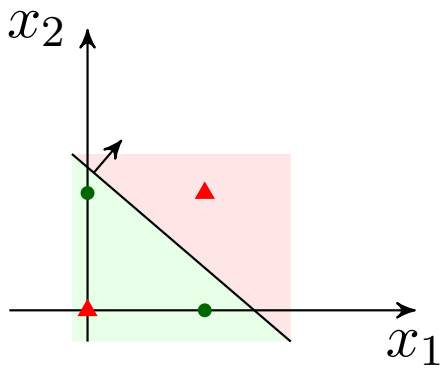
Multi-Layering Perceptrons: *Geometrical Interpretation*

Let's see geometrically what this network of perceptrons does



Multi-Layering Perceptrons: *Geometrical Interpretation*

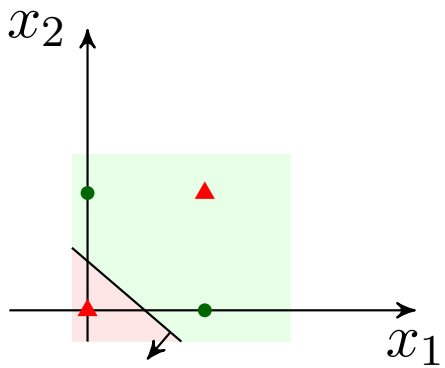
Let's see geometrically what this network of perceptrons does



- First perceptron classifies

Multi-Layering Perceptrons: *Geometrical Interpretation*

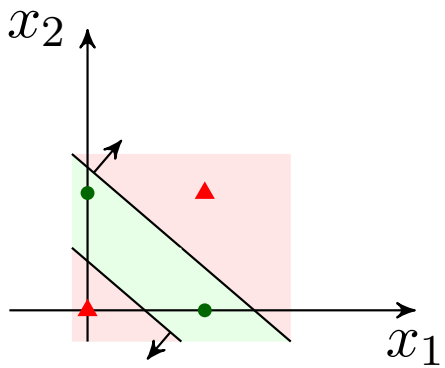
Let's see geometrically what this network of perceptrons does



- First perceptron classifies
- Second perceptron classifies

Multi-Layering Perceptrons: *Geometrical Interpretation*

Let's see geometrically what this network of perceptrons does



- First perceptron classifies
- Second perceptron classifies
- Third perceptron intersects the two regions

Multi-Layering Perceptrons: *Correct Conclusion*

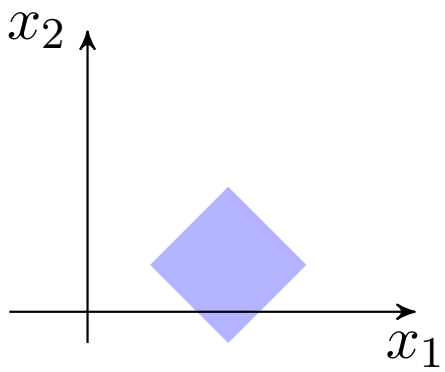
We can learn any binary function using a network of perceptrons

This has given birth to the idea of neural network: looking at perceptron as an artificial model of neuron, we can learn even complicated functions when we have a network of these neurons $\equiv$ a *Neural Network*

+ *But, why should we care only about binary functions? Don't we learn other type of functions as well?*
- Well! We can extend the idea to other problems as well!

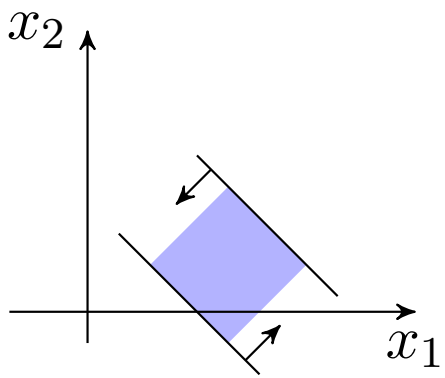
Multi-Layering Perceptrons: *More Complicated Functions*

Assume we have binary classification with real-valued inputs: we want to train a classifier that distinguishes between the two-dimensional points inside the blue area and outside of it



Multi-Layering Perceptrons: *More Complicated Functions*

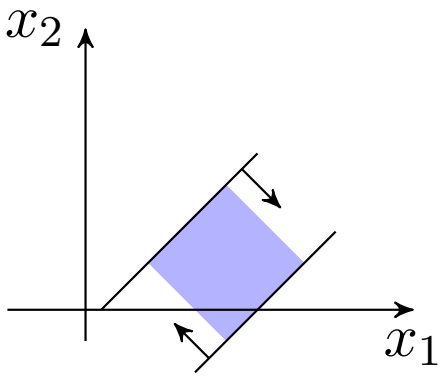
Assume we have binary classification with real-valued inputs: we want to train a classifier that distinguishes between the two-dimensional points inside the blue area and outside of it



- We classify *Region 1* with three perceptrons

Multi-Layering Perceptrons: *More Complicated Functions*

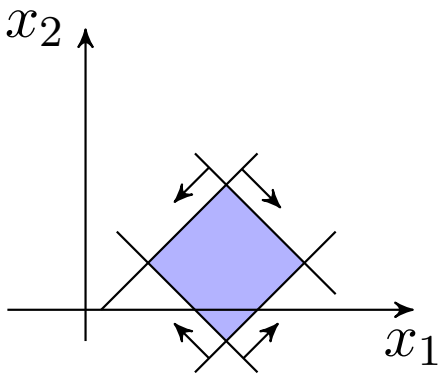
Assume we have binary classification with real-valued inputs: we want to train a classifier that distinguishes between the two-dimensional points inside the blue area and outside of it



- We classify *Region 1* with three perceptrons
- We classify *Region 2* with three perceptrons

Multi-Layering Perceptrons: *More Complicated Functions*

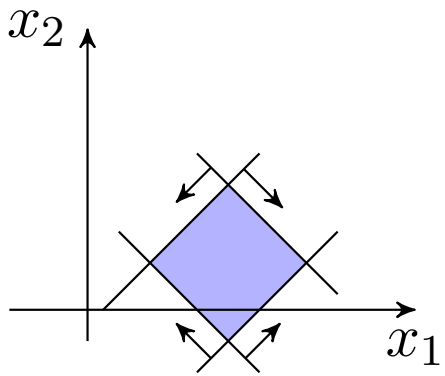
Assume we have binary classification with real-valued inputs: we want to train a classifier that distinguishes between the two-dimensional points inside the blue area and outside of it



- We classify *Region 1* with three perceptrons
- We classify *Region 2* with three perceptrons
- We intersect the two regions with a perceptron

Multi-Layering Perceptrons: *More Complicated Functions*

Assume we have binary classification with real-valued inputs: we want to train a classifier that distinguishes between the two-dimensional points inside the blue area and outside of it



- We classify *Region 1* with three perceptrons
- We classify *Region 2* with three perceptrons
- We intersect the two regions with a perceptron

A network of 7 perceptrons is more than enough!

Multi-Layering Perceptrons: *More Complicated Functions*

+ *What if the classification problem is not binary?*
– Well! We already have seen that we can reduce a multi-class classification to a series of binary classifications[4]

We want to classify an image as dog, cat or car

- *Binary Classification 1: Is it class 0: dog or class 1: {cat, car}?*
  - ↳ *If class 0 ⇝ classification ended*
  - ↳ *If class 1 ⇝ Binary Classification 2: Is it class 0: cat or class 1: car?*

[4] There are better multi-class classification techniques that we learn later

Multi-Layering Perceptrons: *More Complicated Functions*

+ *What if the classification problem is not binary?*
– Well! We already have seen that we can reduce a multi-class classification to a series of binary classifications[4]

We want to classify an image as dog, cat or car

- *Binary Classification 1: Is it class 0: dog or class 1: {cat, car}?*
  - ↳ *If class 0 ⇝ classification ended*
  - ↳ *If class 1 ⇝ Binary Classification 2: Is it class 0: cat or class 1: car?*

Moral of Story

We can learn any classifier with high accuracy using a network of perceptrons

[4] There are better multi-class classification techniques that we learn later

Multi-Layering Perceptrons: *More Complicated Functions*

+ *Fair enough! We are happy with classification! But, what about the case that we have real-valued labels? Can we do it by perceptrons?*
- This is the regression problem!
- And, Yes! We can do this as well using perceptrons

Regression is a supervised learning problem in which

labels belong to a continuous set, e.g., $y_i \in \mathbb{R}$

the label y_i in this case is the function sample at $\boldsymbol{x}_i$

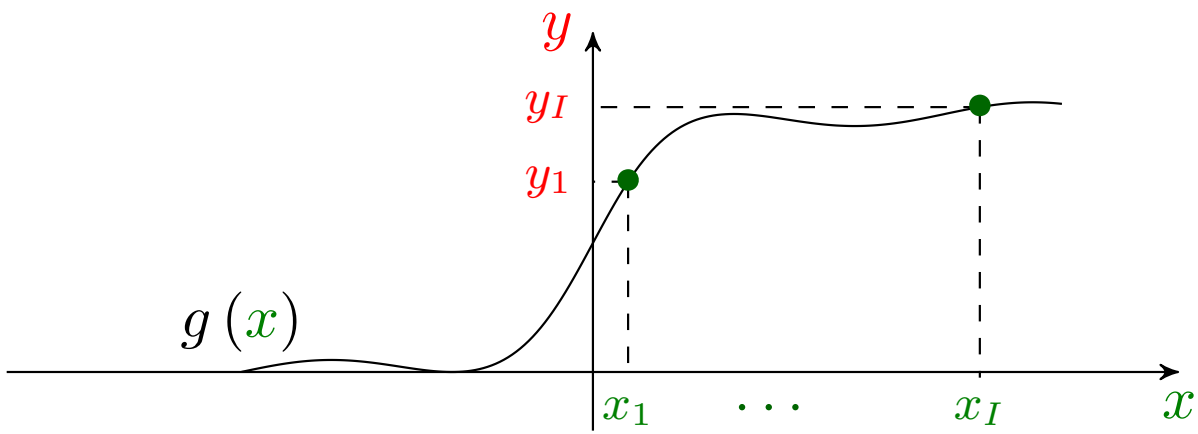
Multi-Layering Perceptrons: *Regression*

In regression, we learn a real-valued function

Let's look at a simple case with one-dimensional inputs, where we can visualize

$$\mathbb{D} = \{(x_i, y_i) : i = 1, \ldots, I\}$$

A visualization of this dataset is



and we want to learn $g(\cdot)$

Multi-Layering Perceptrons: *Regression*

The main question is

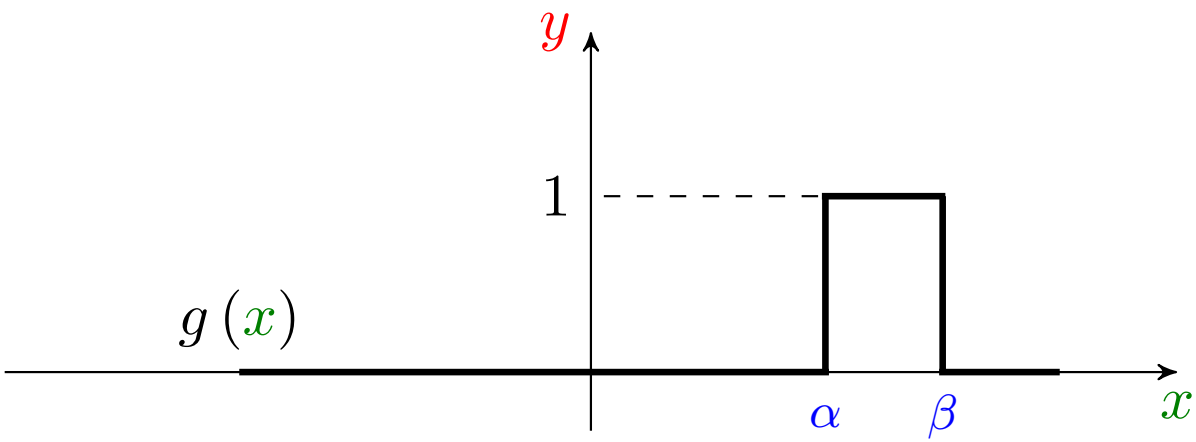
can we realize an arbitrary $g(\cdot)$ via a network of perceptrons?

Multi-Layering Perceptrons: *Regression*

The main question is

> *can we realize an arbitrary $g(\cdot)$ via a network of perceptrons?*

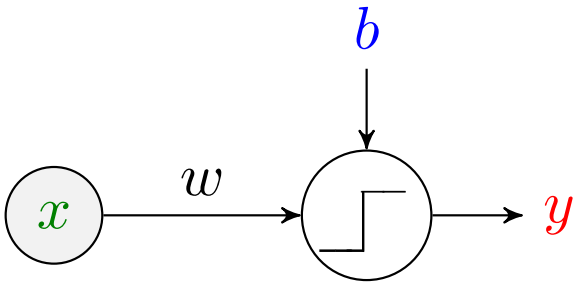
Let's start with a simple function: *the unit pulse*

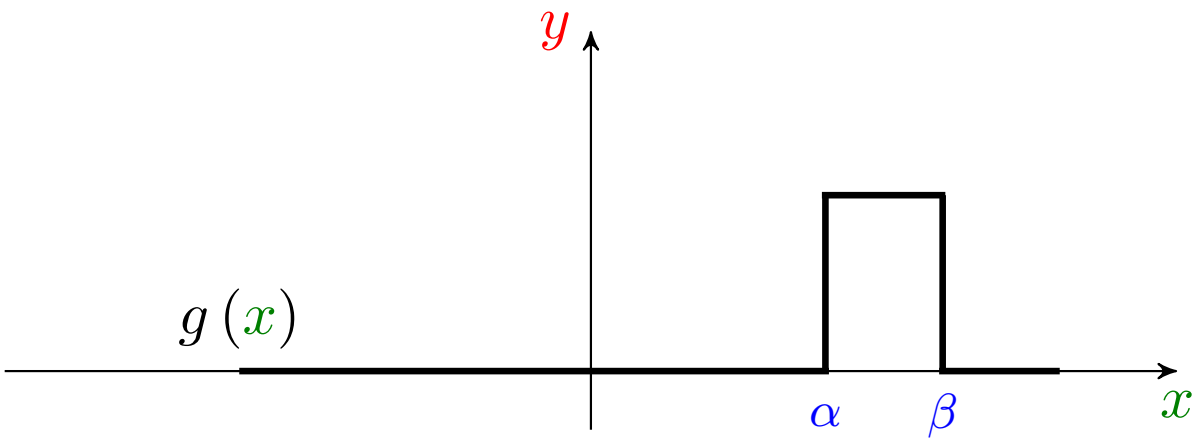


We can realize this function using three perceptrons

Multi-Layering Perceptrons: *Regression*

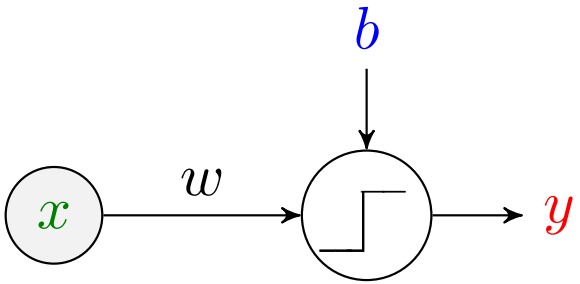
First, let's recall how perceptron looks with one-dimensional input



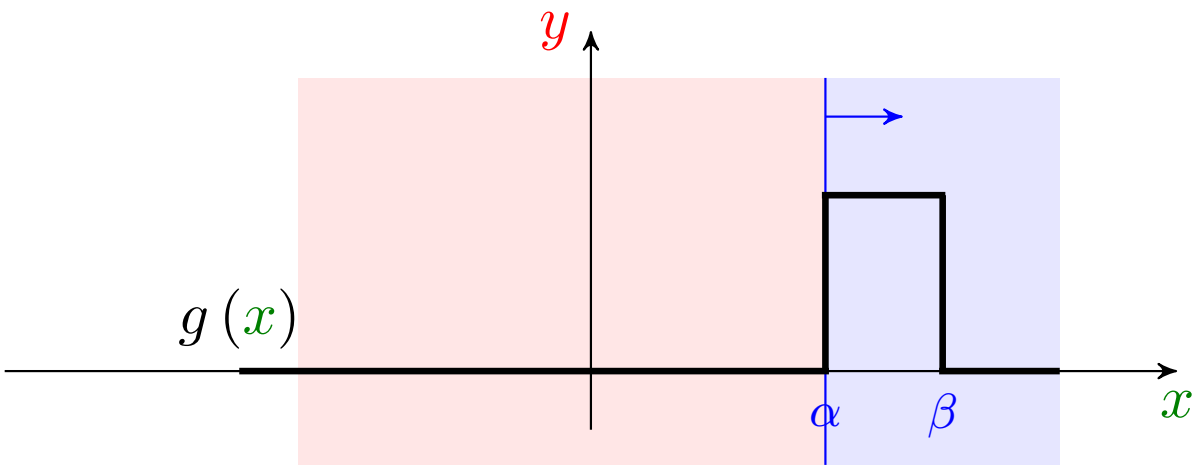


Multi-Layering Perceptrons: *Regression*

First, let's recall how perceptron looks with one-dimensional input

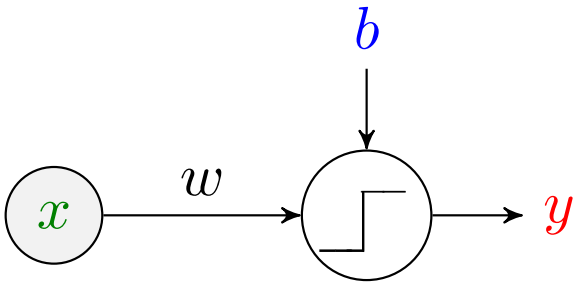


↳ with $w = 1$ and $b = -\alpha$: `blue`



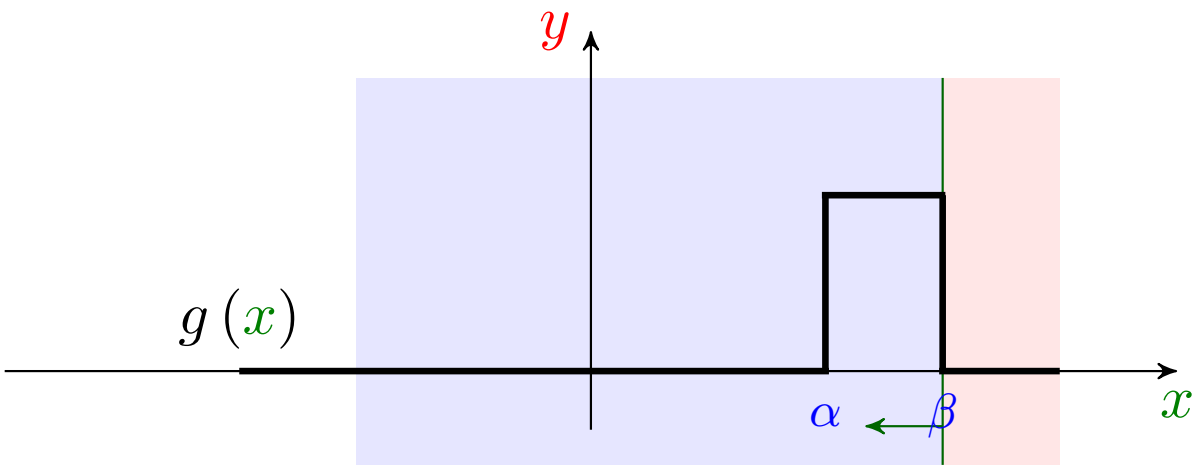
Multi-Layering Perceptrons: *Regression*

First, let's recall how perceptron looks with one-dimensional input



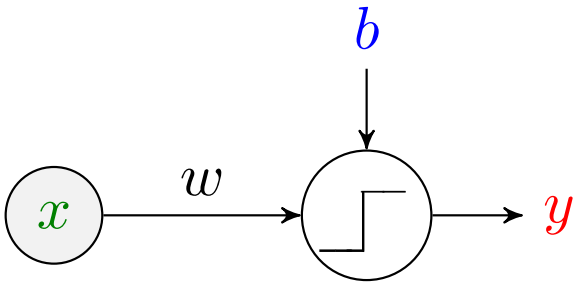
↳ with $w = 1$ and $b = -\alpha$: `blue`

↳ with $w = -1$ and $b = \beta$: `green`



Multi-Layering Perceptrons: *Regression*

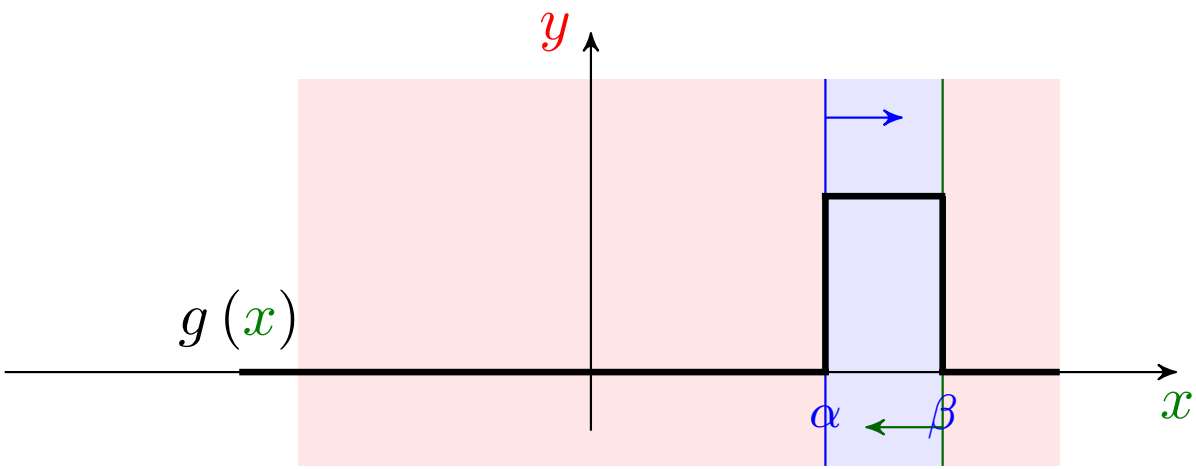
First, let's recall how perceptron looks with one-dimensional input



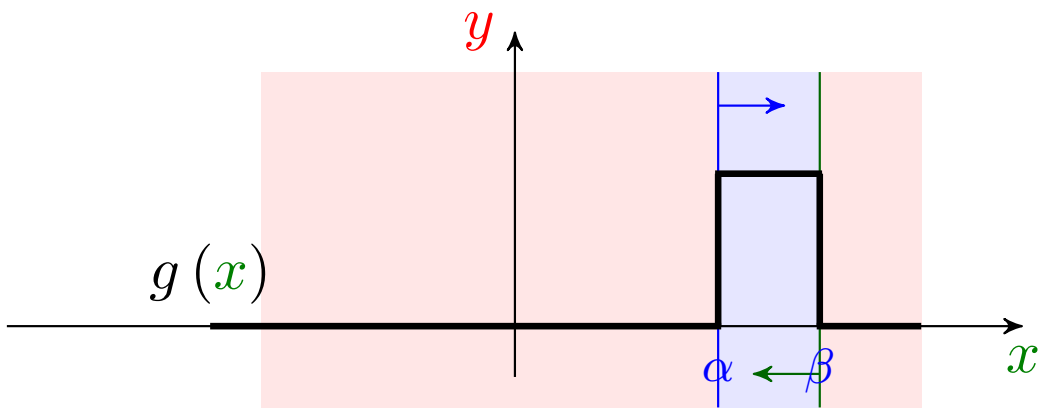
↳ with $w = 1$ and $b = -\alpha$: `blue`

↳ with $w = -1$ and $b = \beta$: `green`

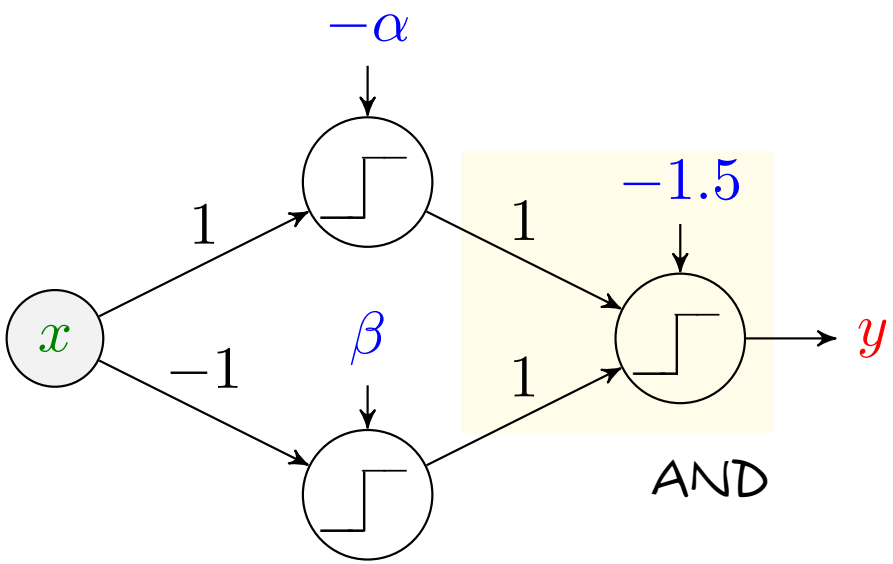
We realize the pulse by applying AND on the outputs of these perceptrons



Multi-Layering Perceptrons: *Regression*

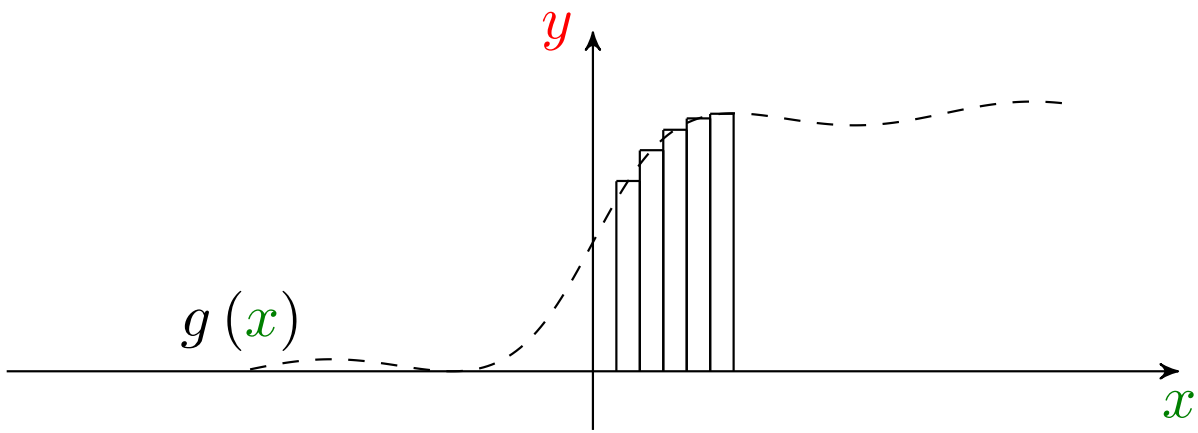


So, we can realize the unit pulse via the following network of perceptrons



Multi-Layering Perceptrons: *Regression*

We can approximate a general function via a weighted sum of unit pulses



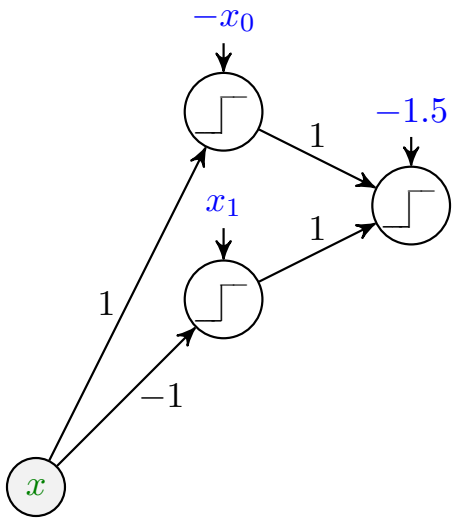
This means that we can approximate any function by

- realizing each pulse by a network of perceptrons
- applying linear transform on the outputs of these networks

Multi-Layering Perceptrons: *Regression*

Real-valued functions are *well-approximated* by

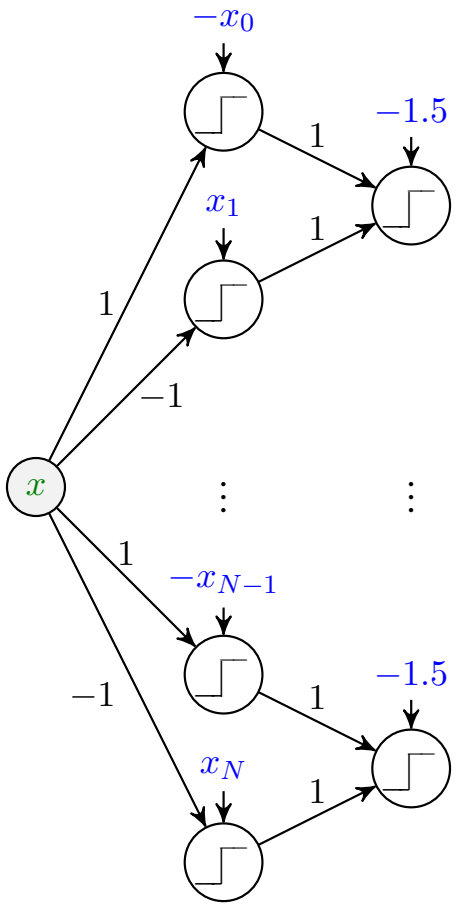
a network of *perceptrons* + a *linear transform*



Multi-Layering Perceptrons: *Regression*

Real-valued functions are *well-approximated* by

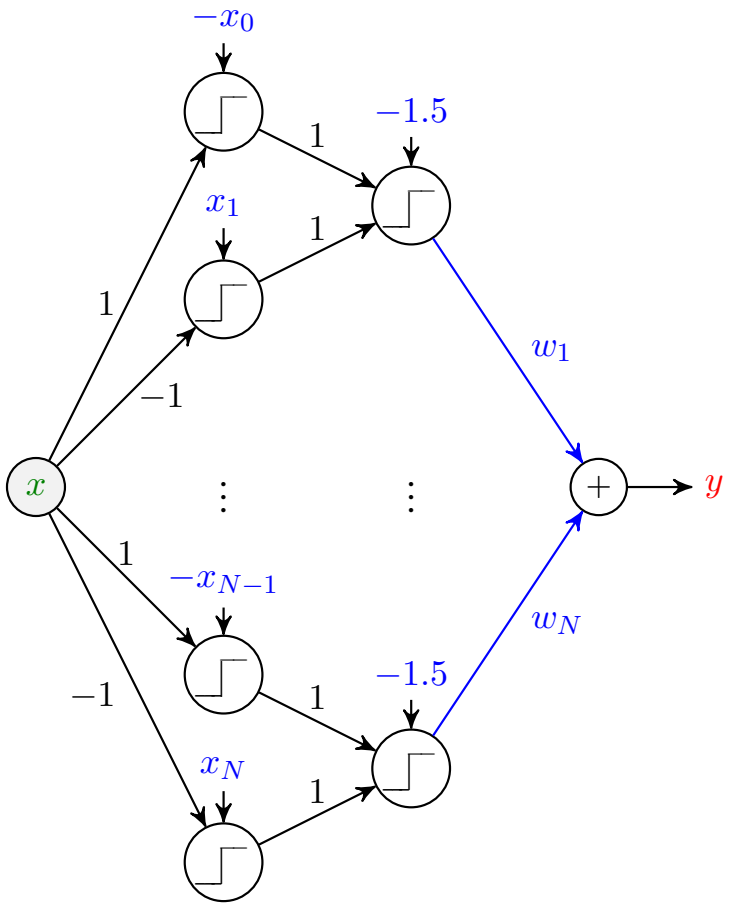
a network of *perceptrons* + a *linear transform*



Multi-Layering Perceptrons: *Regression*

Real-valued functions are *well-approximated* by

a network of *perceptrons* + a *linear transform*



Multi-Layering Perceptrons: *Summary*

A network of perceptrons seems to be a very sophisticated model

- It can learn any classifier
- It can approximate any function with arbitrary accuracy

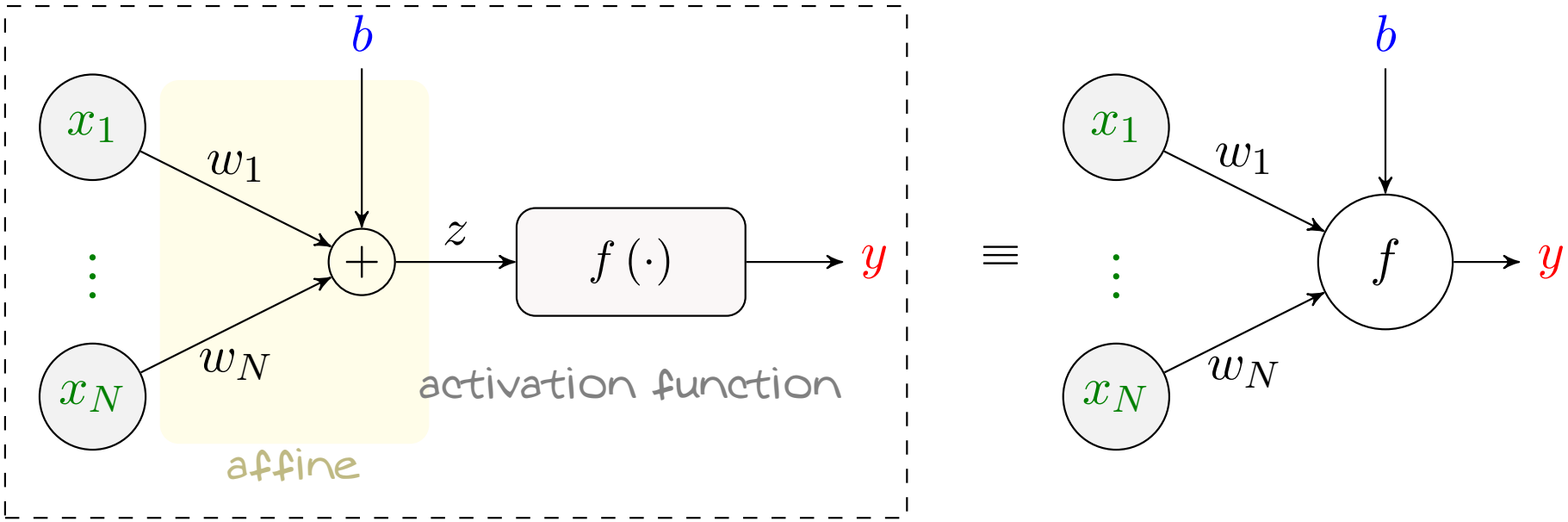
So, it seems natural to train them for our learning tasks

A network of perceptrons is a special artificial neural network

we now formally introduce artificial neural network

Artificial Neuron

Perceptron is a *special* artificial neuron. In general, an artificial neuron is



A neuron with an N-dimensional input has $N + 1$ learnable parameters

- N weights, i.e., $w_1, \ldots, w_N$
- a bias

From now on, when not needed, we drop them from diagram for compactness

Artificial Neuron

The output of neuron y is related to its inputs as $\boldsymbol{x}$

$$y = f\left(\mathbf{w}^{\mathsf{T}}\boldsymbol{x} + b\right)$$

where we define

- $\mathbf{w}^{\mathsf{T}} = [w_1, \ldots, w_N]$ to be the vector of weights
- b to be the bias
- $f(\cdot) : \mathbb{R} \mapsto \mathbb{R}$ to be the activation function

In perceptron, the activation function was *step function*.[5]

A neuron can in general have any activation

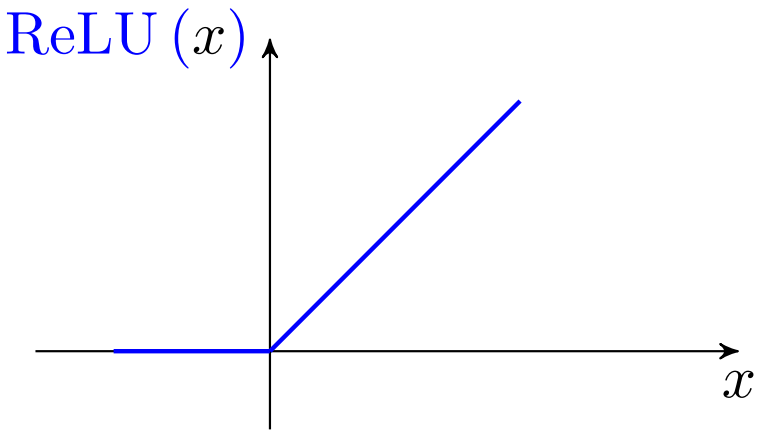
Another special case is the linear activation $f(z) = z$ by which the neuron reduces to an affine function

[5] It turns out soon that this was in fact a bad choice of activation!

Artificial Neuron: *Activation*

If we intend to learn nonlinear functions; then, we need nonlinear activation

Some sample activation functions: *rectified linear unit (ReLU) function*

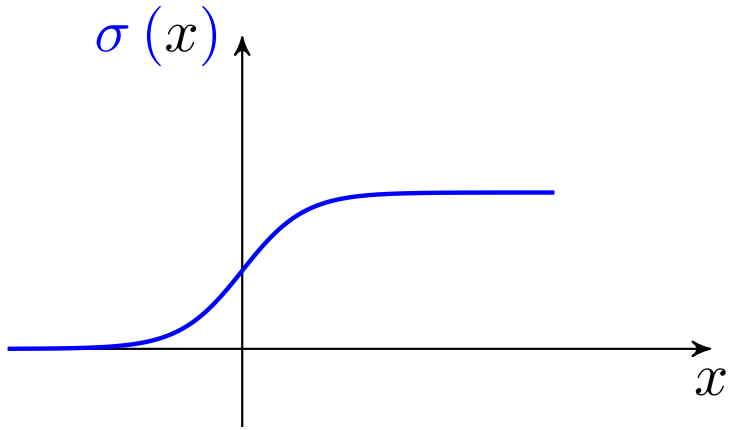


$$\text{ReLU}\,(x) = \max\{x, 0\}$$

Artificial Neuron: *Activation*

If we intend to learn nonlinear functions; then, we need nonlinear activation

Some sample activation functions: *sigmoid function*

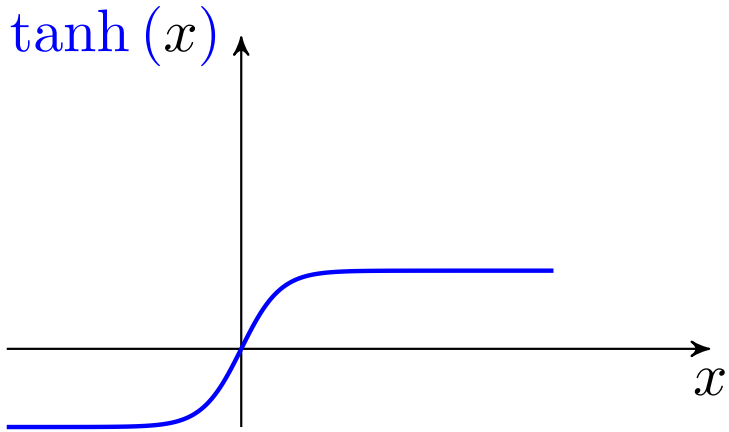


$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Artificial Neuron: *Activation*

If we intend to learn nonlinear functions; then, we need nonlinear activation

Some sample activation functions: *hyperbolic tangent function*

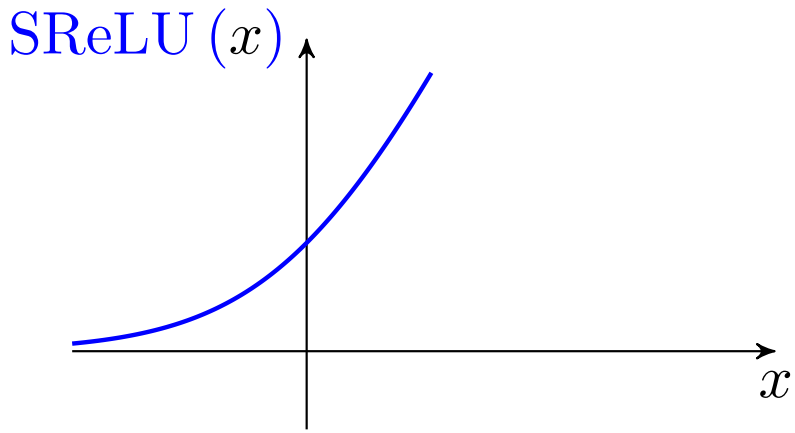


$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Artificial Neuron: *Activation*

If we intend to learn nonlinear functions; then, we need nonlinear activation

Some sample activation functions: *soft ReLU function*

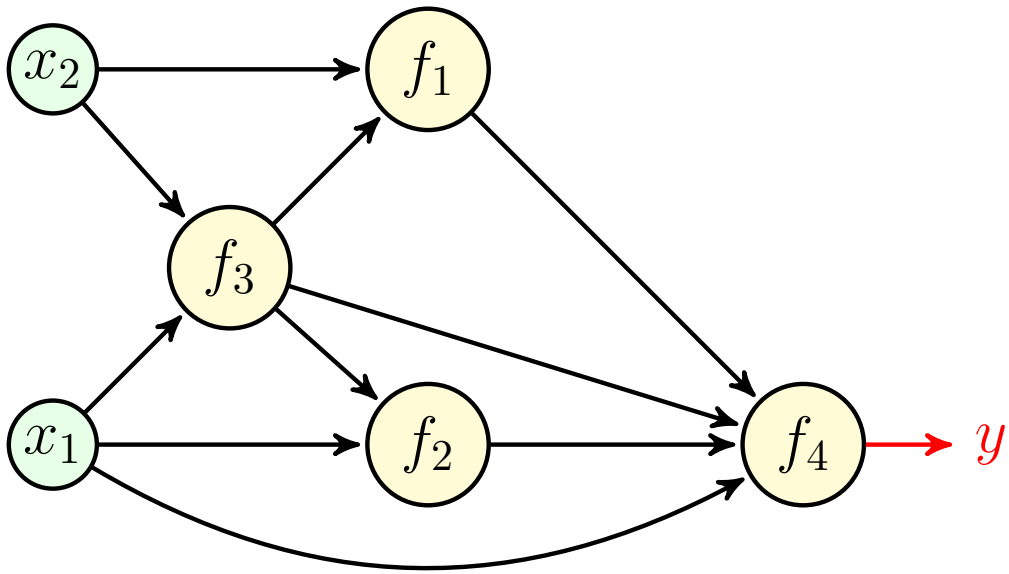


$$\mathrm{SReLU}\,(x) = \log\left(1 + e^{x}\right)$$

Neural Network

Artificial Neural Network

Artificial neural network is a *directed graph* that connects a set of inputs to a set of outputs. The nodes of this graph are neurons whose activation functions are known and whose *weights and biases are learnable*.

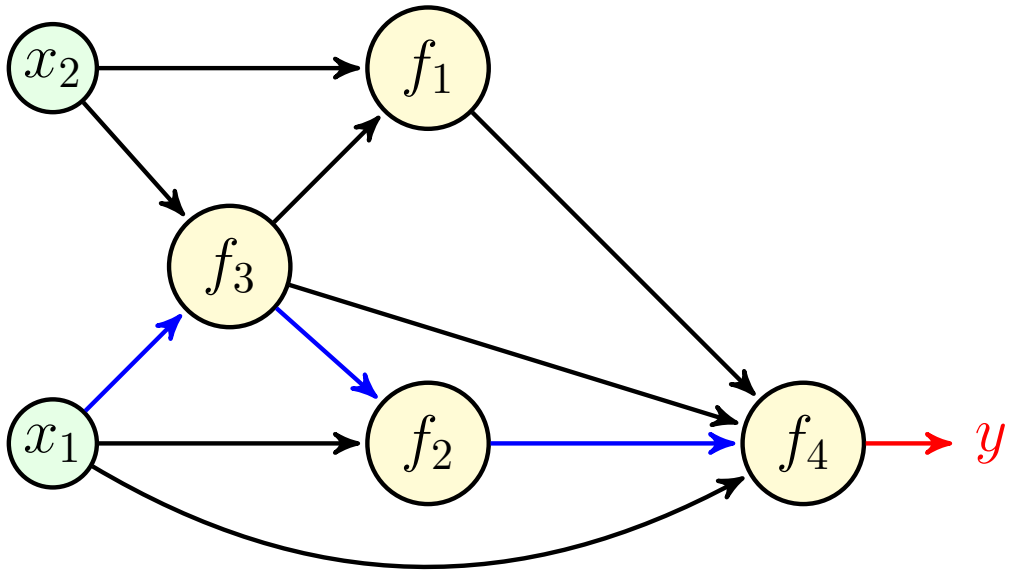


Unless we are talking to biologists, we can safely drop the term artificial ☺

Neural Network: *Depth*

Depth of a Neural Network

The longest path between an input and the output

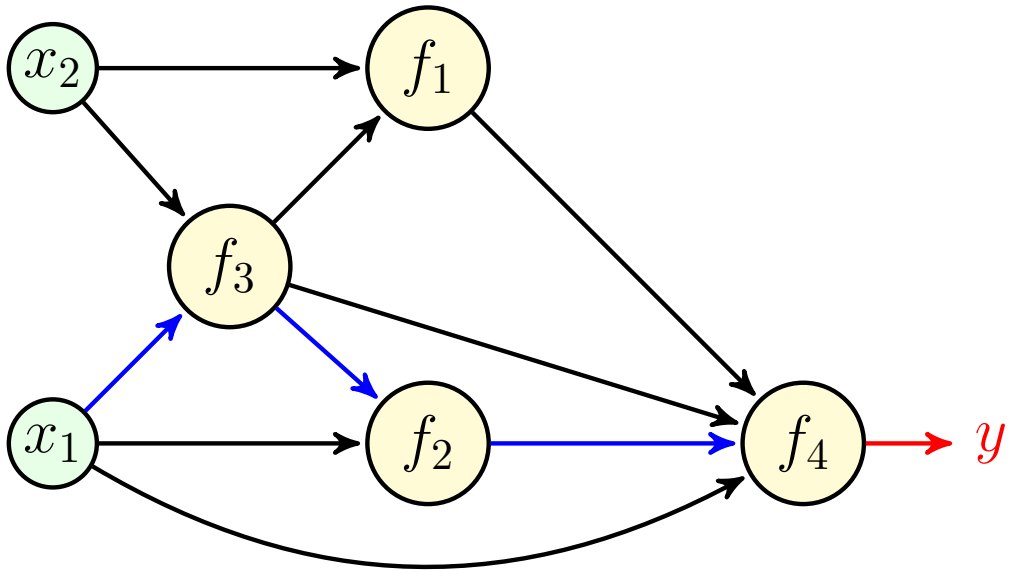


Here, the depth is 3

Deep Neural Network

Deep Neural Network

A neural network whose depth is larger than 2



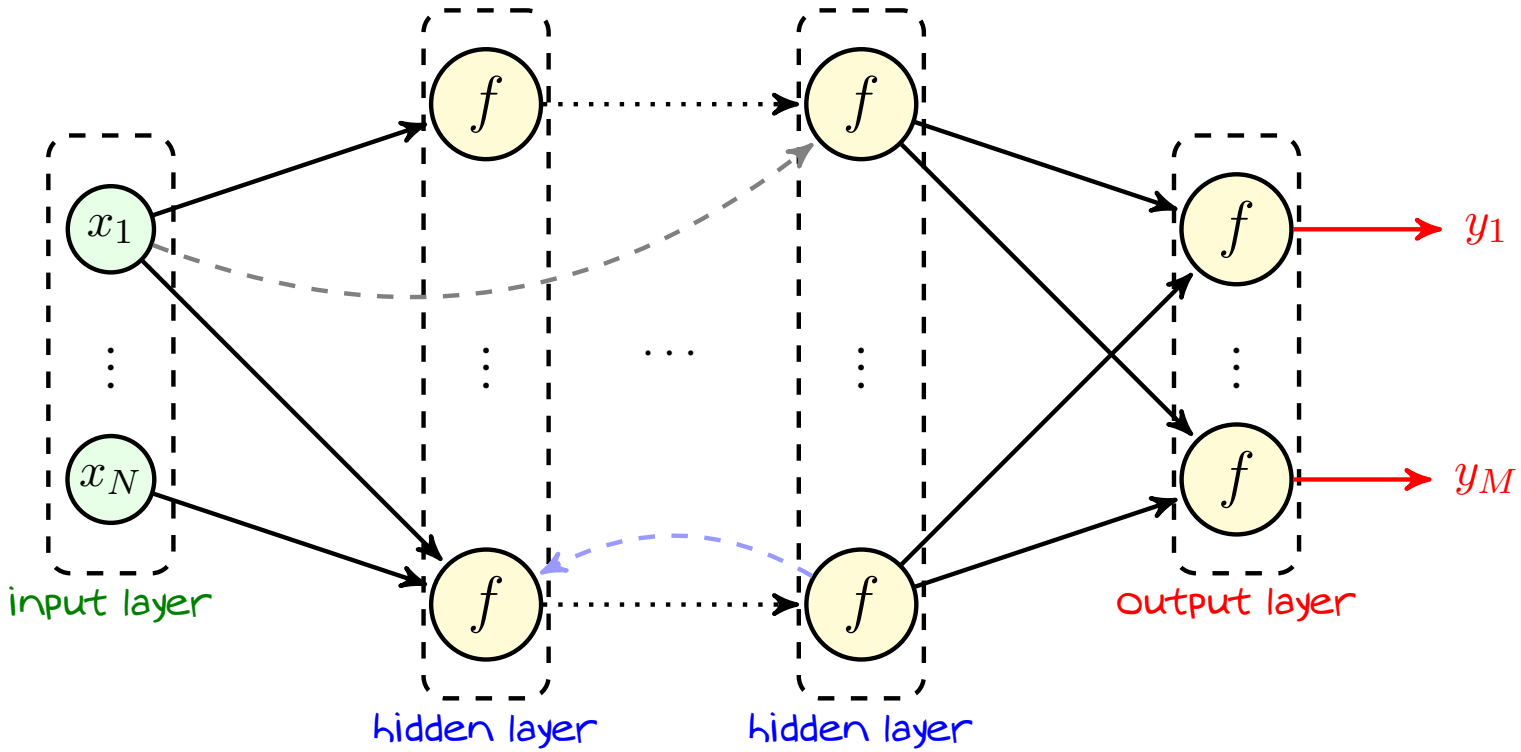
Here, the depth is 3 > 2 $\rightsquigarrow$ it's a deep neural network

Neural Network: *Layered Architecture*

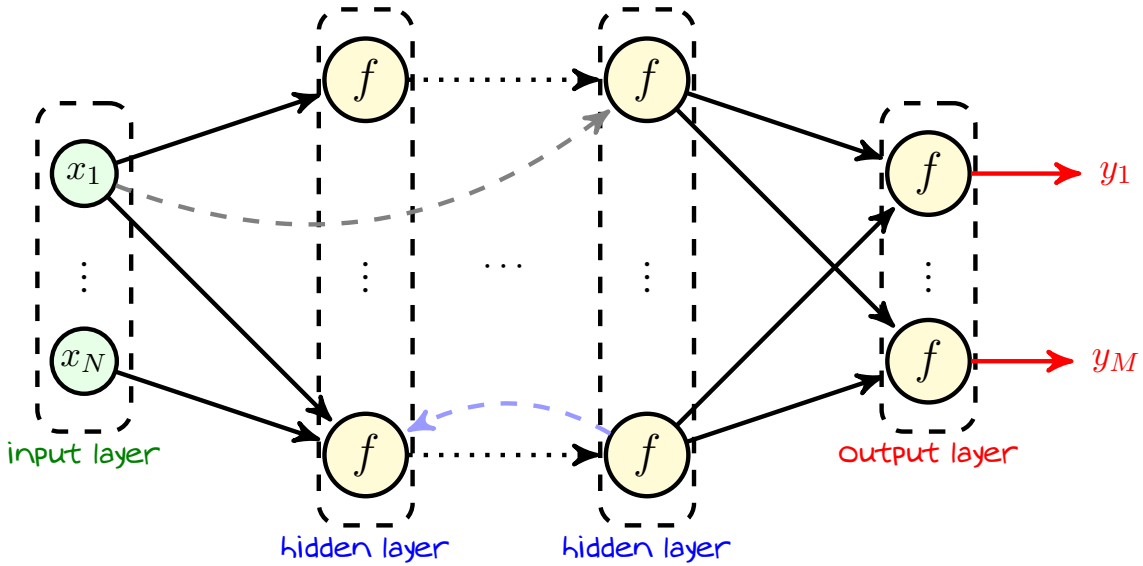
In practice, we use neural networks with layered architectures

Layer

A subset of neurons that are in the same distance from inputs



Neural Network: *Layered Architecture*

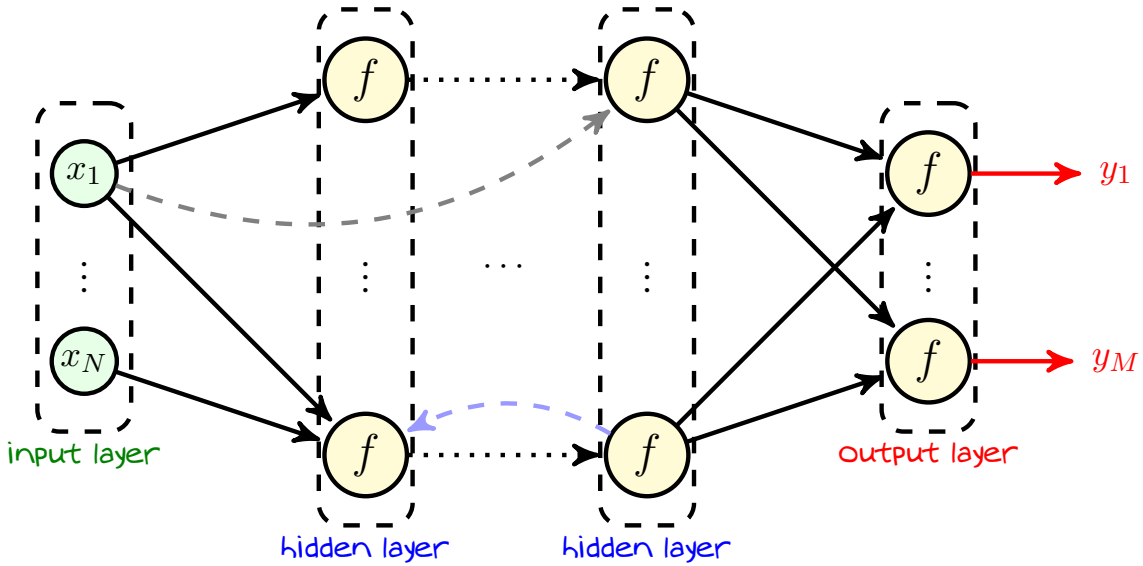


There are three types of layers

- Input layer that only contains inputs (no neuron)
- Output layer that contains the neurons computing network outputs
- hidden layers that contain neurons and are in between

It's readily seen from the definition that *depth = # hidden layers + 1*

Neural Network: *Layered Architecture*



It's readily seen from the definition that with layered architecture

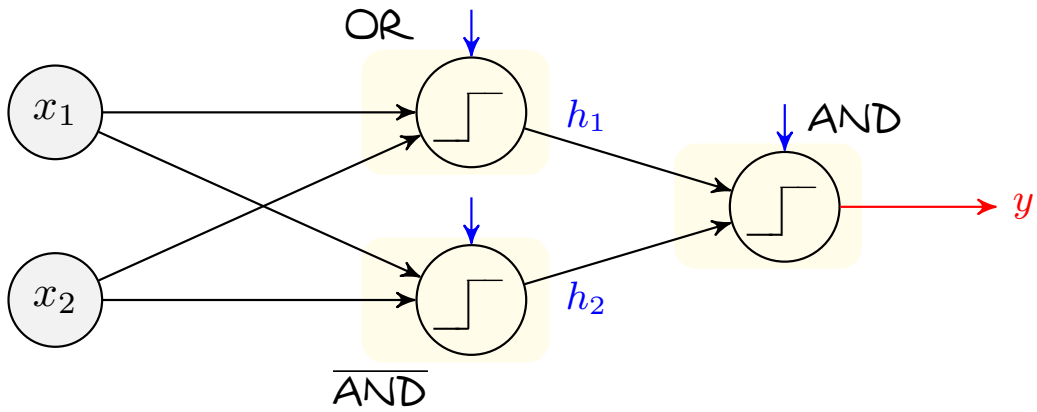
$$\boxed{\textit{depth} = \textit{\# hidden layers + 1}}$$

Hence, we could equivalently say that

a deep neural network has more than one hidden layer

Neural Network: *Layered Architecture*

Let's try our knowledge on the XOR neural network

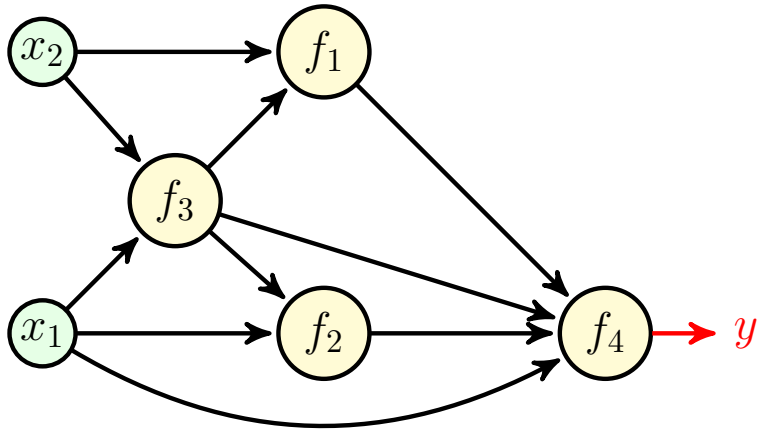


- Input layer has two inputs
- Output layer has a single neuron (perceptron)
- One hidden layer that has two neurons
  - ↳ It's not a deep neural network, since it has only one hidden layer

Neural Networks are *Models*

+ *If we talk in terms of ML components, what is a deep neural network?*
– It's a parameterized function from inputs to outputs; so, *it's a model*
+ *Well! What are then the hyperparameters and learnable parameters?*
– Anything that describes the architecture is hyperparameter
  ↳ *Number of neurons, how they are connected, depth, activations, . . .*
– The weights and biases are the learnable parameters
+ *So, it means if the architecture of the neural network is known, we explicitly know the learnable parameters?*
– Exactly! Let's look at the dummy diagram we had in the previous slides!

Neural Networks are *Models*



Here, we've chosen to have 4 neurons with activations $f_1(\cdot), \ldots, f_4(\cdot)$ arranged in the above form: *these are hyperparameters*

Now that the architecture is fixed, we could say

- *Neurons 1, 2, 3 have two inputs: each of them has two weights and a bias*
- *Neurons 4 has four inputs: it has four weights and a bias*

So in total we have $3 \times (2+1) + (4+1) = 14$ learnable parameters

Deep Learning

+ *Are we finally ready to define Deep Learning?*
– Yes! There we go

Deep Learning (DL)

When we use a deep neural network to address a learning task,

we are doing deep learning

Now, let's get things a bit clear

- In ML, we talk about any model, any loss, any dataset
- In *Representation Learning*, we roughly talk about models that can describe nonlinear functions: *this includes deep neural networks*
- In DL, we have a deep neural network as the model

So, we can say DL ⊂ Representation Learning ⊂ ML

Starting with Deep Learning

At this point, we know

- For a given learning task, we specify the dataset, a model and a loss
- In DL, we use deep neural networks as models
- To accomplish the learning task, we need to train the model
  - ↳ *find weights and biases that achieve minimal empirical risk*

But, we yet don't know?

- How to minimize the empirical loss?
  - ↳ *what algorithm should we use?*
- What kind of hyperparameters, i.e., architecture, should we use?
  - ↳ *how many neurons do we need?*
  - ↳ *how should we connect the neurons?*

This is what we learn from now on! The only last piece of preliminary knowledge that we need is the method of gradient decent which we study next.

Function Optimization

The model training always reduces to an optimization problem

$$\mathbf{w}^{\star} = \underset{\mathbf{w}}{\operatorname{argmin}}\, \hat{R}\left(\mathbf{w}\right) = \underset{\mathbf{w}}{\operatorname{argmin}}\, \frac{1}{I} \sum_{i=1}^{I} \mathcal{L}\left(f_{\mathbf{h}}\left(\boldsymbol{x}_i | \mathbf{w}\right), y_i\right) \qquad \text{(Training)}$$

Let's recall each component of this optimization problem

- $f_{\mathbf{h}}\left(\cdot | \mathbf{w}\right)$ model with hyperparameters $\mathbf{h}$ and learnable parameters $\mathbf{w}$
  - ↳ *in DL, it is the input-output relation of a neural network whose architecture is specified by $\mathbf{h}$ and whose weights and biases are collected in $\mathbf{w}$*
- $\boldsymbol{x}_i$ is a data-point with label y_i, and I is size of dataset
- $\mathcal{L}$ is the loss function

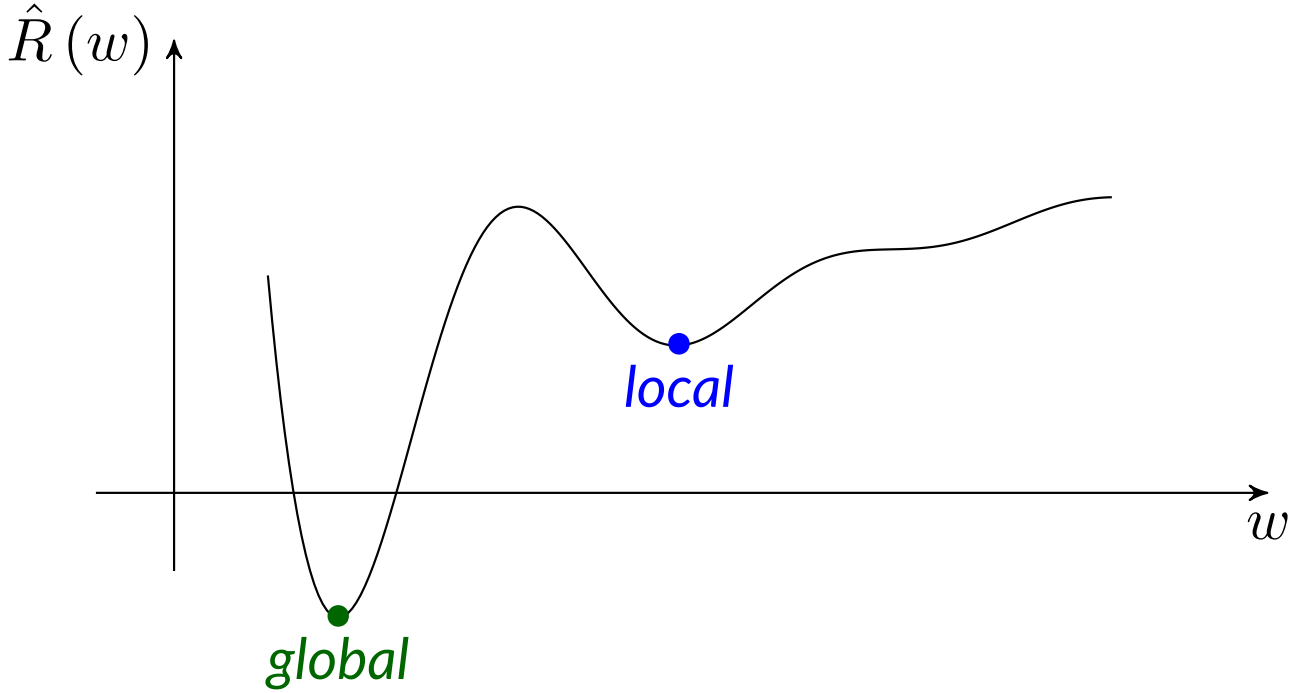
No matter what we choose, at the end of the day we need to solve

$$\min_{\mathbf{w}} \hat{R}\left(\mathbf{w}\right)$$

Function Optimization

In general, the empirical risk $\hat{R}(\mathbf{w})$ can have local and global minima

Let's take a look at a simple visual case with only one parameter, i.e., $w \in \mathbb{R}$



We are happy if we get the global; but, many times getting to a local is enough!

Function Optimization

+ *Why is it a big problem? We cold grid w and search for the grid with smallest empirical risk. We then find it with a good accuracy!*
- For only one parameter yes! But, we have seen deep neural networks. They have too many neurons, and hence too many parameters!

Say for an accurate approximation with only one parameter, we need G grids.

If we have D parameters, i.e., $\mathbf{w} \in \mathbb{R}^D$, we need

$$G^D \text{ grids}$$

to get an approximation with the same accuracy!

For practical neural networks with $D = 10^5$, this is impossible!

we need to have an optimization algorithm with feasible complexity

Optimization Algorithms

We look for an optimization algorithm, *or as ML people call it "an optimizer"*, that starts from an initial point and moves us towards the point where the empirical risk is minimized *(at least a local one)* in a feasible number of steps

---

Let's clear things up: we are looking for an iterative approach as below

1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
2: Choose some small ϵ, and set $t = 1$
3: **while** $\Delta > \epsilon$ **do**
4: Determine a vector $\boldsymbol{\mu}^{(t)} \in \mathbb{R}^D$ based on $\hat{R}(\mathbf{w})$ ← we need to figure out
5: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} + \boldsymbol{\mu}^{(t)}$
6: Update the deviation $\Delta = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{(t-1)})|$
7: **end while**

where we would like to have following properties

- ↳ most of the time the empirical risk reduces in each iteration
- ↳ the algorithm stops after a feasible number of iterations

Optimization Algorithms

1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
2: Choose some small ϵ, and set $t = 1$
3: **while** $\Delta > \epsilon$ **do**
4: Determine a vector $\boldsymbol{\mu}^{(t)} \in \mathbb{R}^D$ based on $\hat{R}(\mathbf{w})$ ← we need to figure out
5: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} + \boldsymbol{\mu}^{(t)}$
6: Update the deviation $\Delta = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{(t-1)})|$
7: **end while**

We are going to get what we want, if we set

$$\boldsymbol{\mu}^{(t)} \textit{ to be proportional to the negative of gradient at } \mathbf{w}^{(t-1)}$$

This is what we call the *gradient descent algorithm*. *But, before we start with this algorithm, let's recap some basic notions of calculus!*

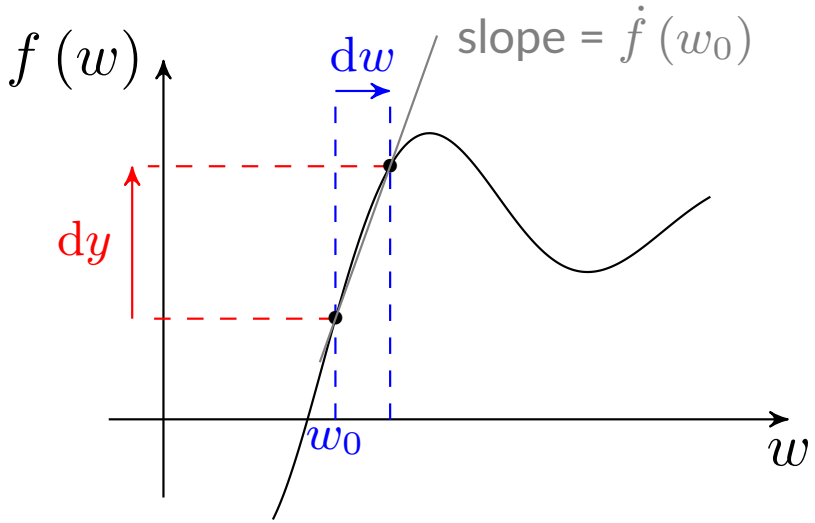
Review: *Derivative of a Function*

Derivative of one-dimensional function $f(w)$ at point $w = w_0$ is defined as

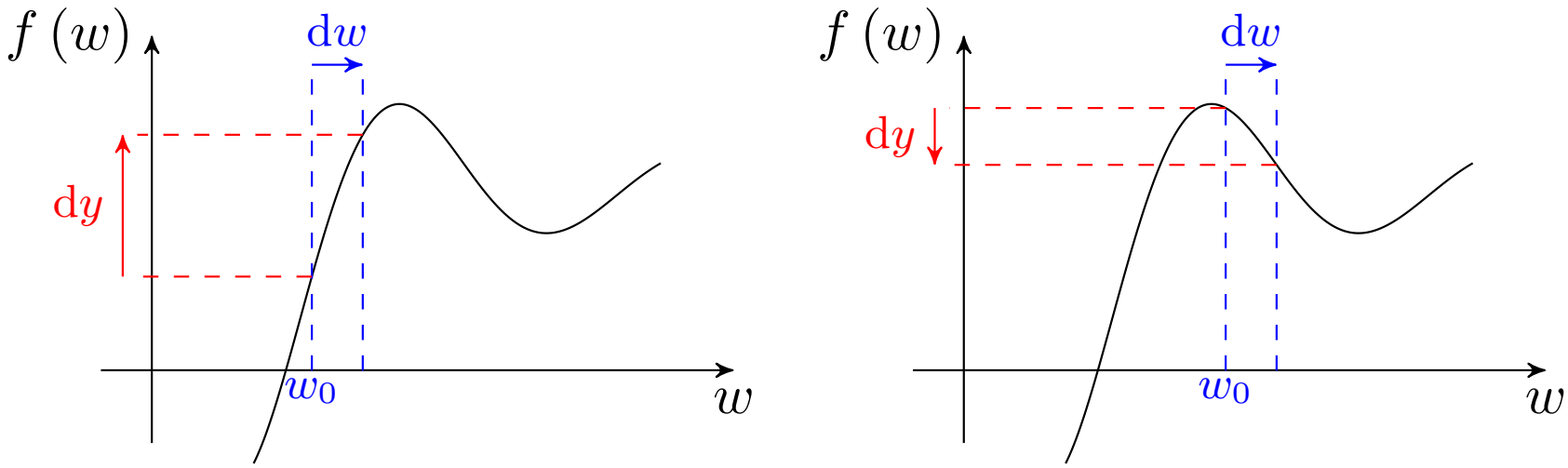
$$\dot{f}(w_0) = \frac{\mathrm{d}}{\mathrm{d}w} f(w_0) = f'(w_0) = \lim_{\delta \to 0} \frac{f(w_0 + \delta) - f(w_0)}{\delta}$$

This definition is intuitively interpreted as follows:

> Let $y = f(w)$. If we vary w around w_0 with a tiny step $\mathrm{d}w$; then,
>
> Variation of $y = \mathrm{d}y = \dot{f}(w_0)\,\mathrm{d}w$



Review: *Derivative of a Function*



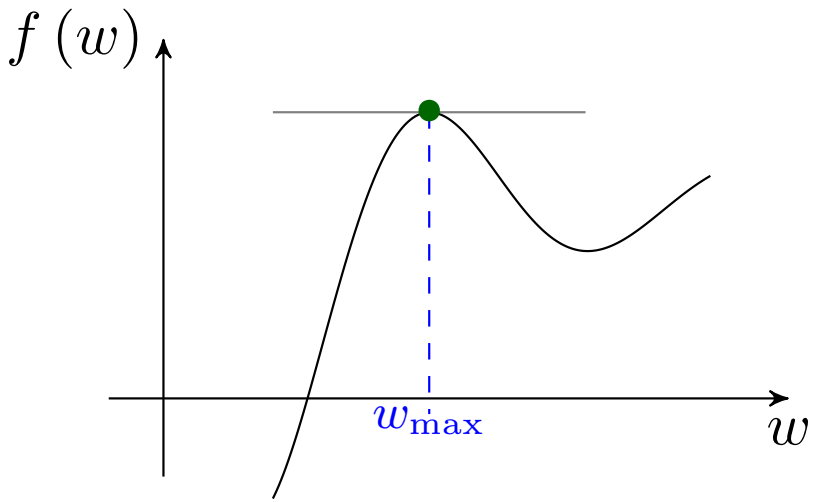
The derivative represents the slope of function

- $\dot{f}(w_0) > 0$ means *increasing* w *will increase* $y = f(w)$
- $\dot{f}(w_0) < 0$ means *increasing* w *will decrease* $y = f(w)$

So, we could also say: *the derivative shows the moving direction on* w*-axis towards which the function increases*; or alternatively, *its negative is the direction that function decreases*

Review: *Derivative of a Function*

When do we have the derivative equal to zero? Either we are at a *maximum*



Starting before the maximum,

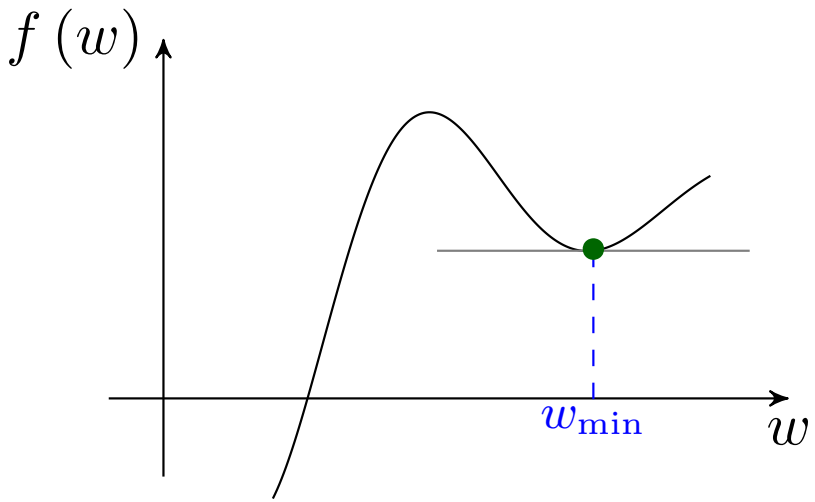
- The derivative is *first positive* and gradually *reduces to zero*
- As we pass the maximum the derivative gets *more and more negative*

So around the maximum as we increase w, the derivative reduces

$$\ddot{f}(w_{\max}) = \frac{\mathrm{d}^2}{\mathrm{d}w^2} f(w_{\max}) = f''(w_{\max}) < 0$$

Review: *Derivative of a Function*

When do we have the derivative equal to zero? Either we are at a *minimum*



Starting before the minimum,

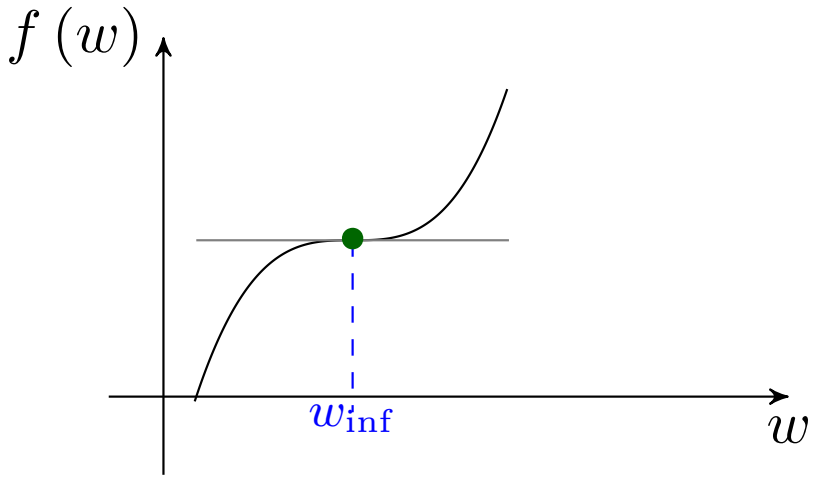
- The derivative is *first negative* and gradually *increases to zero*
- As we pass the minimum the derivative gets *more and more positive*

So around the maximum as we increase w, the derivative reduces

$$\ddot{f}(w_{\min}) = \frac{\mathrm{d}^2}{\mathrm{d}w^2} f(w_{\min}) = f''(w_{\min}) > 0$$

Review: *Derivative of a Function*

When do we have the derivative equal to zero? Either we are at an *inflection*



Starting before the inflection point,

- The derivative is *first positive* and gradually *decreases to zero*
- As we pass the inflection point the derivative gets *again positive*

So around the inflection point, the second derivative changes sign

$$\ddot{f}(w_{\text{inf}}) = \frac{\mathrm{d}^2}{\mathrm{d}w^2} f(w_{\text{inf}}) = f''(w_{\text{inf}}) = 0$$

Review: *Gradient of a Function*

+ *What about multi-variable functions, e.g., $f(\mathbf{w})$ for $\mathbf{w} = [w_1, \ldots, w_N]$?*
- We can take derivative with respect to each variable, i.e.,

$$\dot{f}_n(\mathbf{w}_0) = \frac{\partial}{\partial w_n} f(\mathbf{w}_0)$$

This is what we call *partial derivative*

Partial derivative n represents the same thing: *slope in direction of w_n*

Let $y = f(\mathbf{w})$. If we vary $\mathbf{w}$ around $\mathbf{w}_0$ in N-dimensional space with

$$\mathrm{d}\mathbf{w} = [\mathrm{d}w_1, \ldots, \mathrm{d}w_N]$$

whose entries are very tiny; then, the variation of y is

$$\mathrm{d}y = \dot{f}_1(\mathbf{w}_0)\,\mathrm{d}w_1 + \ldots + \dot{f}_N(\mathbf{w}_0)\,\mathrm{d}w_N = \sum_{n=1}^{N} \dot{f}_n(\mathbf{w}_0)\,\mathrm{d}w_n$$

Review: *Gradient of a Function*

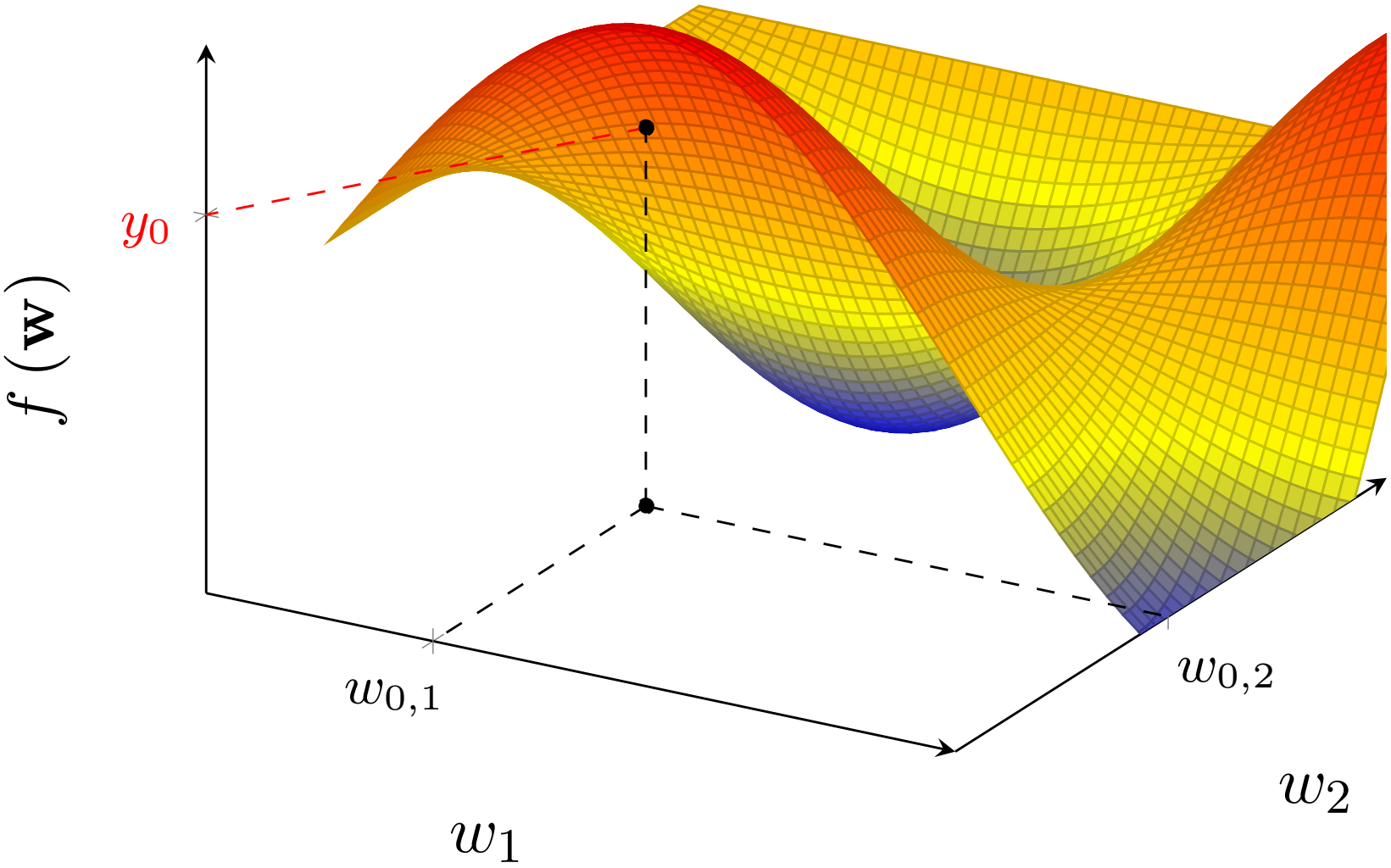
We can use inner-product to represent $\mathrm{d}y$ *compactly*

$$\mathrm{d}y = \sum_{n=1}^{N} \dot{f}_n(\mathbf{w}_0)\,\mathrm{d}w_n = \underbrace{\begin{bmatrix}\dot{f}_1(\mathbf{w}_0) & \dots & \dot{f}_N(\mathbf{w}_0)\end{bmatrix}}_{\nabla f(\mathbf{w}_0)^{\mathsf{T}}} \begin{bmatrix}\mathrm{d}w_1 \\ \vdots \\ \mathrm{d}w_N\end{bmatrix}$$

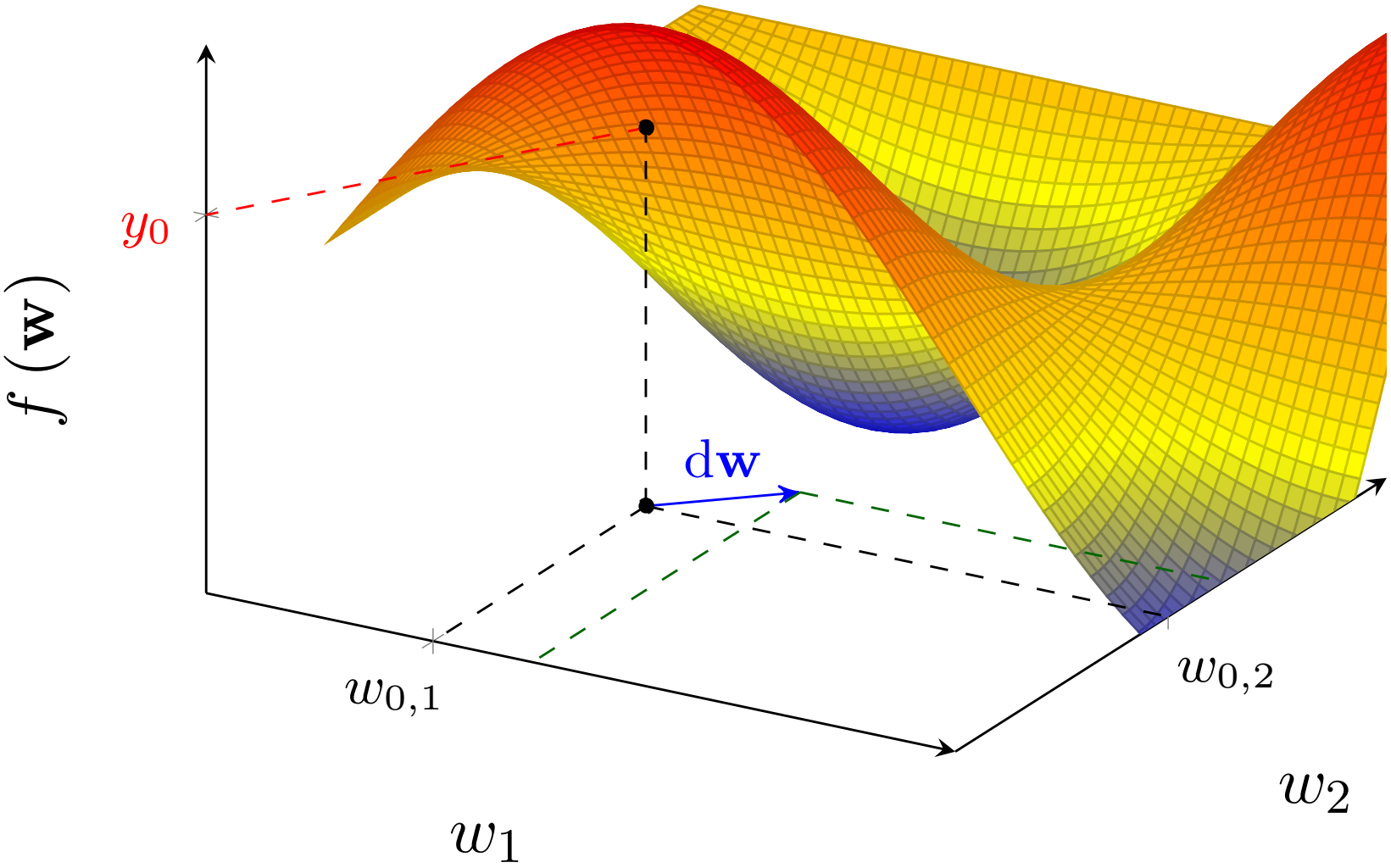
$$= \nabla f(\mathbf{w}_0)^{\mathsf{T}} \mathrm{d}\mathbf{w}$$

We call $\nabla f(\mathbf{w}_0)$ *the gradient of* $f(\cdot)$ *at* $\mathbf{w} = \mathbf{w}_0$

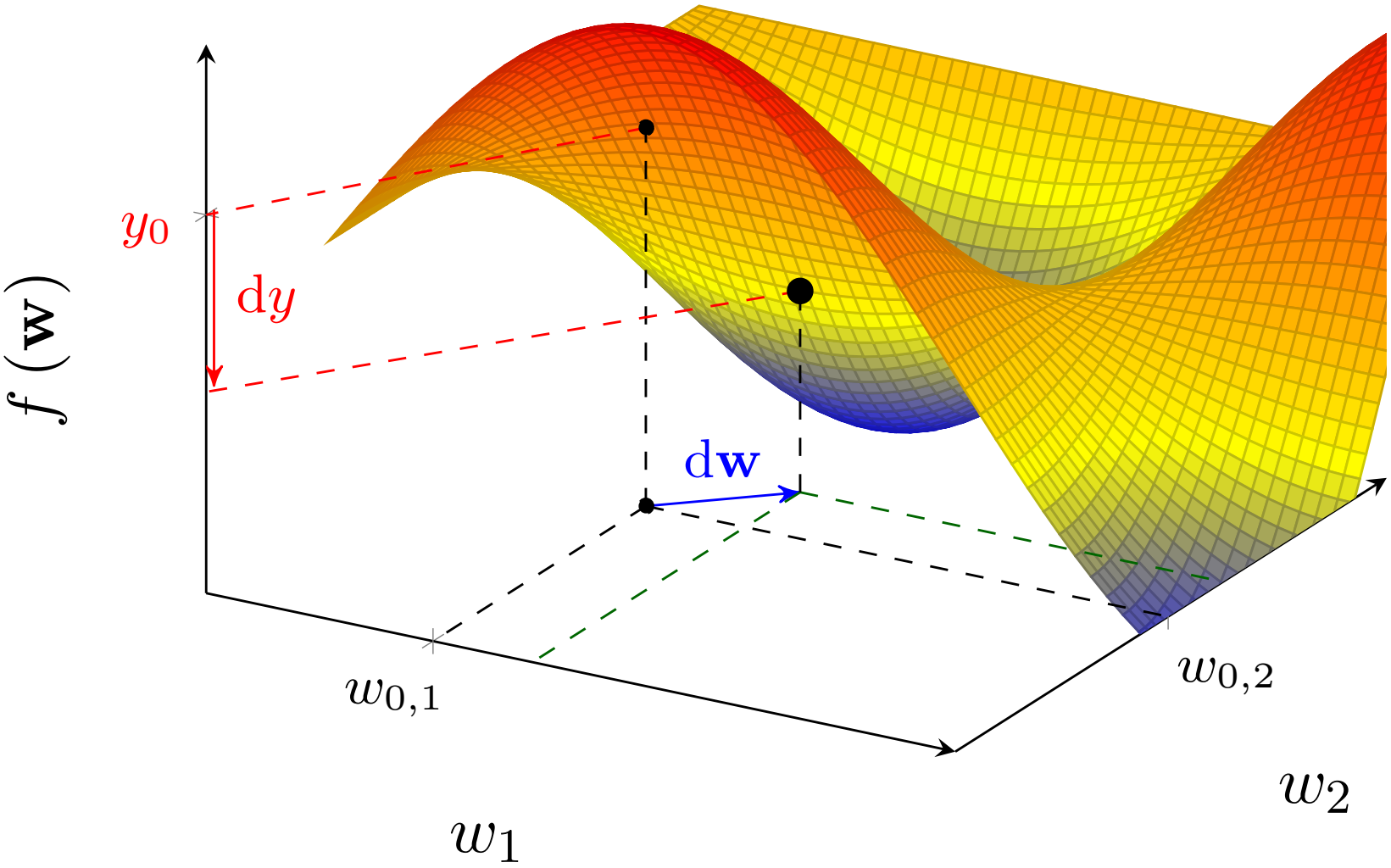
Review: *Gradient of a Function*



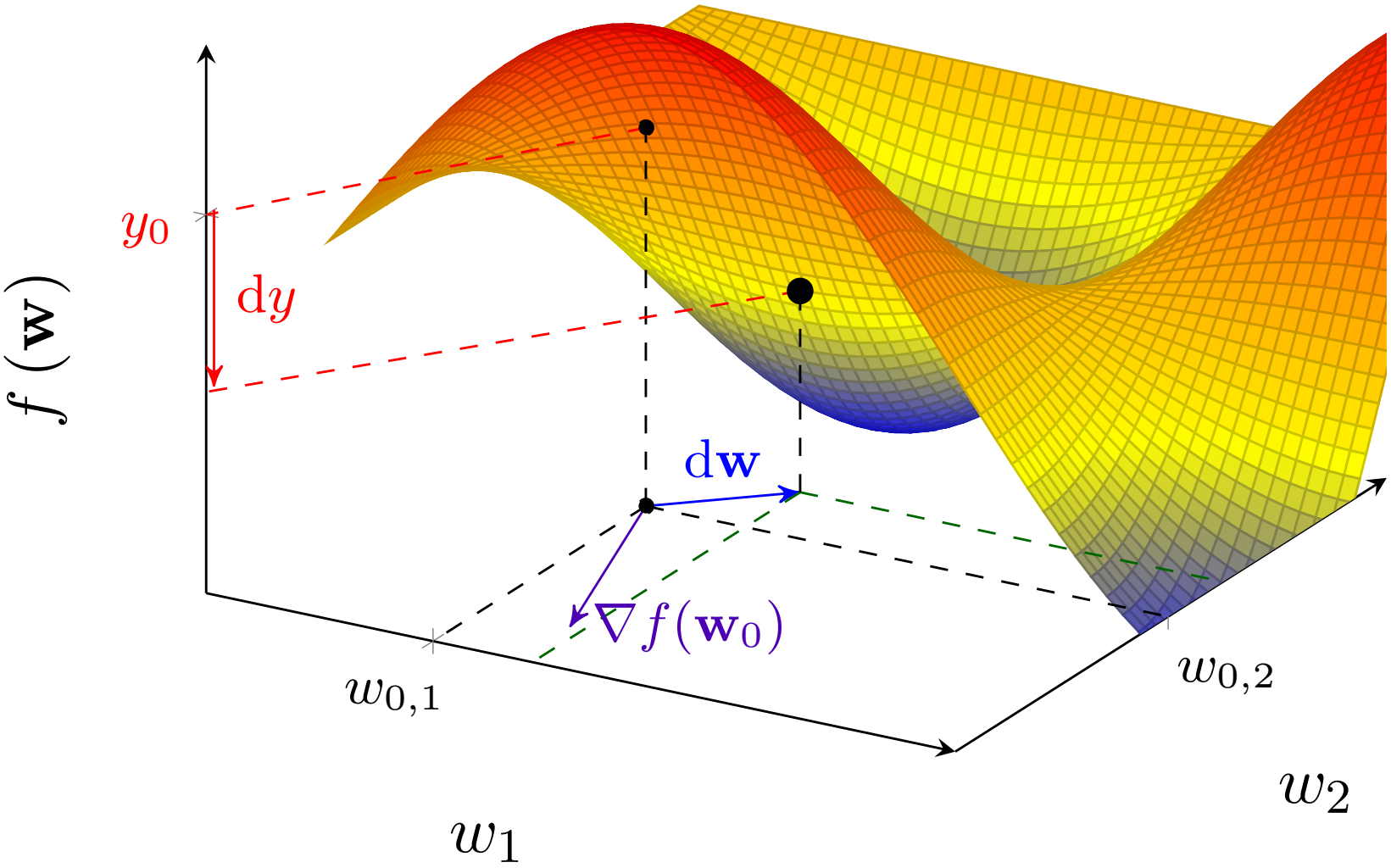
Review: *Gradient of a Function*



Review: *Gradient of a Function*

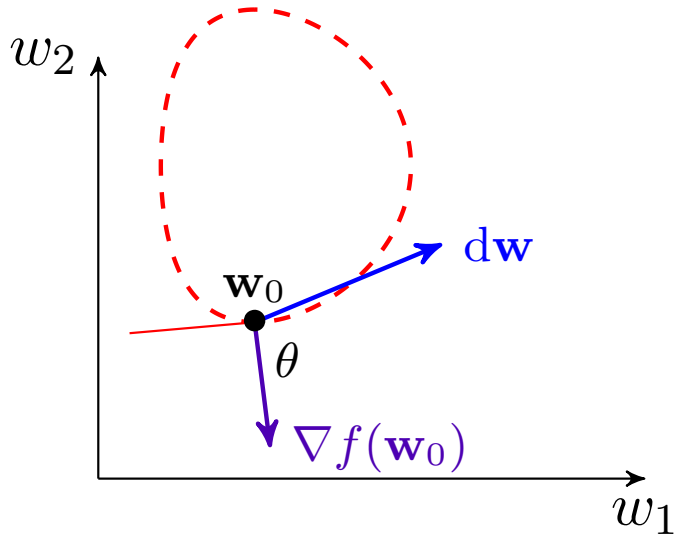


Review: *Gradient of a Function*



Review: *Gradient of a Function*

Let's get to the $\mathbf{w}$*-plane:* the gradient is perpendicular to the contour level



The variation of y is the inner product of these two vectors

$$\mathrm{d}y = \nabla f\left(\mathbf{w}_0\right)^{\mathsf{T}} \mathrm{d}\mathbf{w} = \|\nabla f\left(\mathbf{w}_0\right)\| \|\mathrm{d}\mathbf{w}\| \cos\left(\theta\right)$$

where $\|\cdot\|$ is the Euclidean norm, i.e., $\|\mathbf{w}\| = \sqrt{w_1^2 + w_2^2}$

Review: *Gradient of a Function*

We move with a tiny step of fixed size: so we have

$$\|\mathrm{d}\mathbf{w}\| = \epsilon$$

for some small ϵ

How can we move, such that y *maximally increases?* Well, we need $\theta = 0$ meaning that

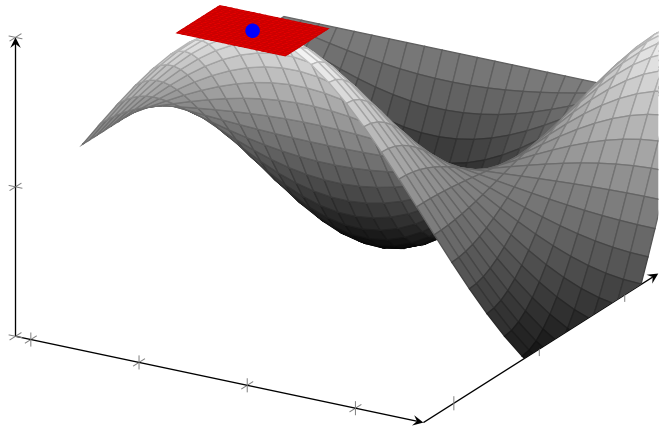
we should move in the direction of gradient

Alternatively, the function decreases maximally if $\theta = \pi$ or

we move in the direction of negative gradient

Review: *Gradient of a Function*

When do we have zero gradient? Either when we are at a maximum

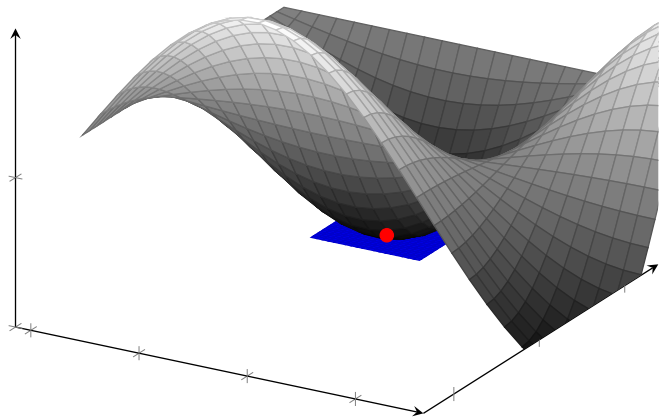


We can again relate it to the *second order derivatives* of the function

at maximum Hessian matrix is negative definite

Review: *Gradient of a Function*

When do we have zero gradient? or when we are at a minimum

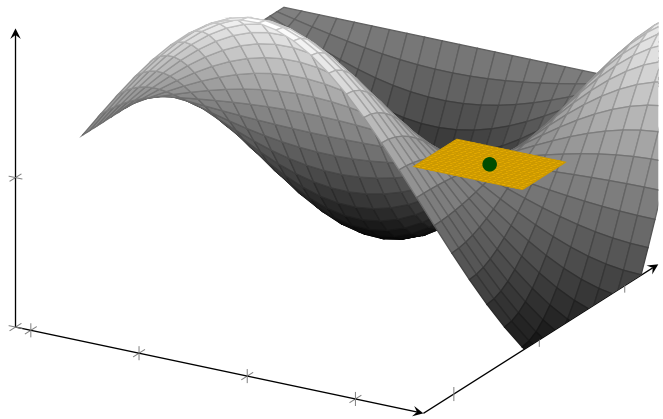


We can again relate it to the *second order derivatives* of the function

at minimum Hessian matrix is positive definite

Review: *Gradient of a Function*

When do we have zero gradient? or when we are at a *saddle point*



We can again relate it to the *second order derivatives* of the function

at saddle point Hessian matrix is neither negative nor positive definite

Review: *Gradient of a Function*

Just as a reminder: Hessian is the matrix of all *second order derivatives*

$$\nabla^2 f(\mathbf{w}_0) = \begin{bmatrix} \frac{\partial^2}{\partial w_1^2} f(\mathbf{w}_0) & \frac{\partial^2}{\partial w_1 \partial w_2} f(\mathbf{w}_0) & \cdots & \frac{\partial^2}{\partial w_1 \partial w_N} f(\mathbf{w}_0) \\ \vdots & & & \vdots \\ \frac{\partial^2}{\partial w_N \partial w_1} f(\mathbf{w}_0) & \frac{\partial^2}{\partial w_N \partial w_2} f(\mathbf{w}_0) & \cdots & \frac{\partial^2}{\partial w_N^2} f(\mathbf{w}_0) \end{bmatrix}$$

We never use the Hessian matrix in this course

Moral of Story: *Gradient Decent*

+ *What is the whole motive of this discussions?*
– Simple: *at any point* $\mathbf{w}_0$, *if we want to move in a direction that the function reduces, the best direction is negative of gradient at* $\mathbf{w}_0$

So, we can complete our optimization algorithm as follows:

1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
2: Choose some small ϵ and η, and set $t = 1$
3: **while** $\Delta > \epsilon$ **do**
4: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta \nabla \hat{R}(\mathbf{w}^{(t-1)})$
5: Update the deviation $\Delta = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{(t-1)})|$
6: **end while**

The scalar η is the step-size we take in each iteration:

we usually call it learning rate

Behavior of Gradient Decent

+ *Can we always use gradient descent?*
– Pretty much Yes! The problem starts only when *the empirical loss is not differentiable*

How to handle this problem?

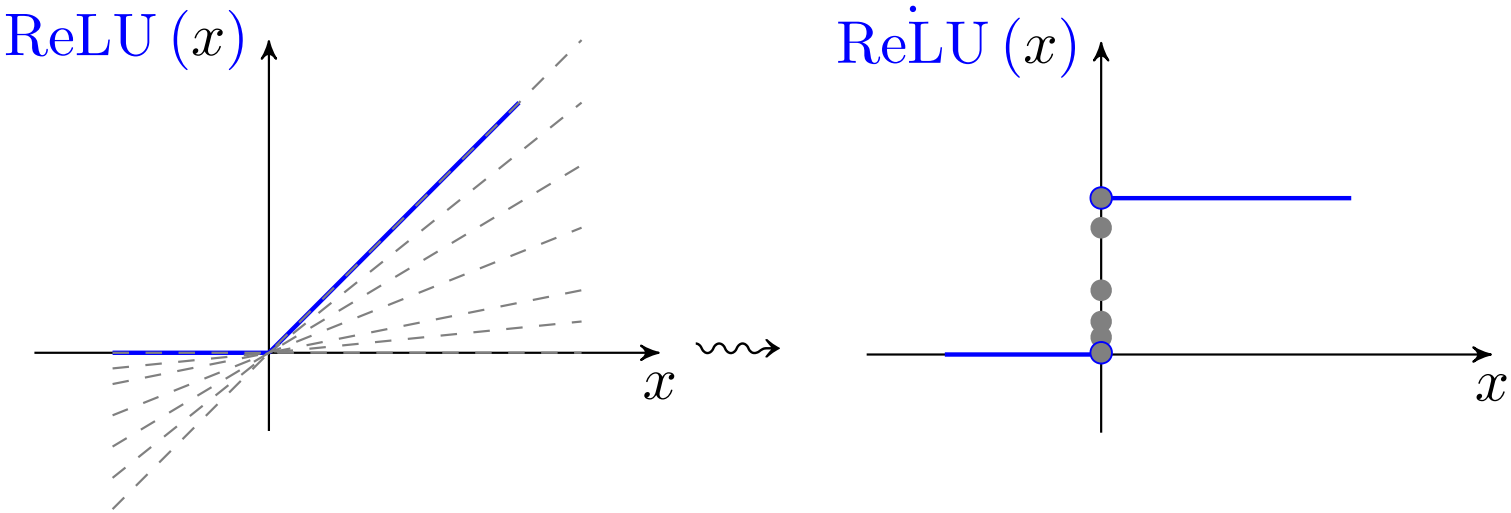
There are two sources for *being non-differentiable*

1. a function that is continuous but not differentiable
2. a discontinuous function

Let's look at each case separately

Behavior of Gradient Decent: *Non-differentiable Elements*

An example of a non-differentiable continuous function is ReLU

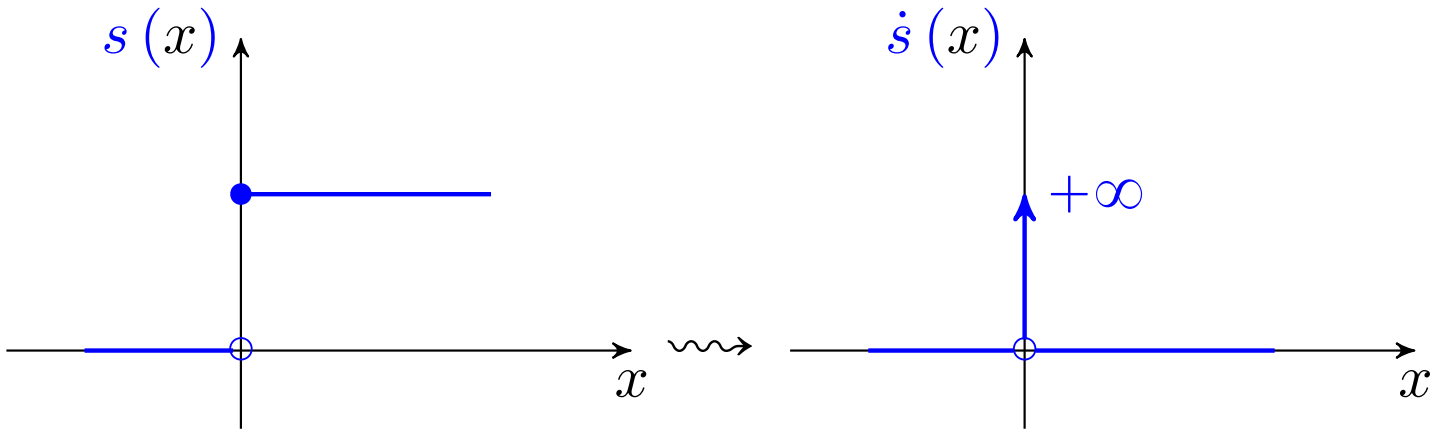


In this case, we define a *sub-gradient* and use it instead of gradient

take the slope of a line that lies below the curve at the point

Behavior of Gradient Decent: *Discontinuous Elements*

An example of a discontinuous function is the step function



The gradient is somehow infinite! *We can only rely on the sign of variation*

we always avoid such elements in our model and loss

Bingo! You may recall that *we discouraged the choice of activation and loss function in the perceptron machine*

Behavior of Gradient Decent: *At Stationary Points*

+ *Now, let's assume that we've handled differentiability. Does gradient decent always end up at the minimum point?*
- This brings up the concept of *convergence*

Let's look at the algorithm again

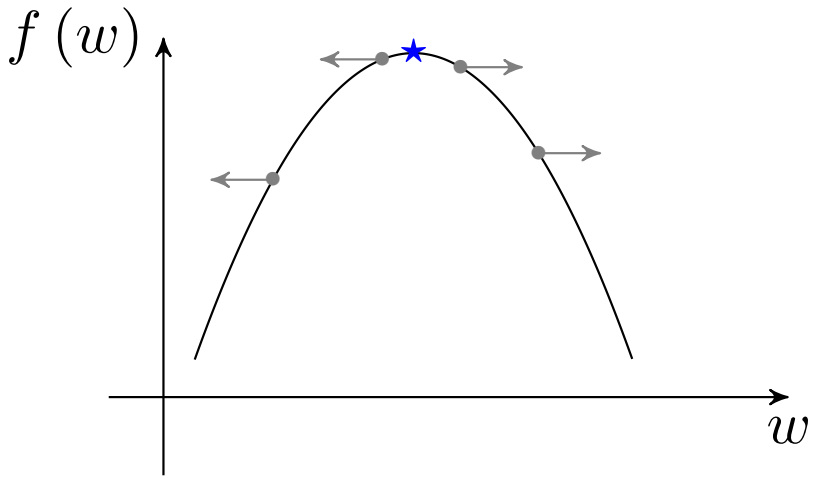
1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
2: Choose some small ϵ and η, and set $t = 1$
3: **while** $\Delta > \epsilon$ **do**
4: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta \nabla \hat{R}(\mathbf{w}^{(t-1)})$
5: Update the deviation $\Delta = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{(t-1)})|$
6: **end while**

Intuitively, if we set ϵ *very small*: *the algorithm stops when the gradient is close to zero, i.e., when we are at a maximum, minimum or an inflection/saddle-point*

Let's see how the algorithm behaves when we get close to such point

Behavior of Gradient Decent: *At Stationary Points*

When we are around a maximum



If we are exactly at a maximum; then, the algorithm stops. But, in reality

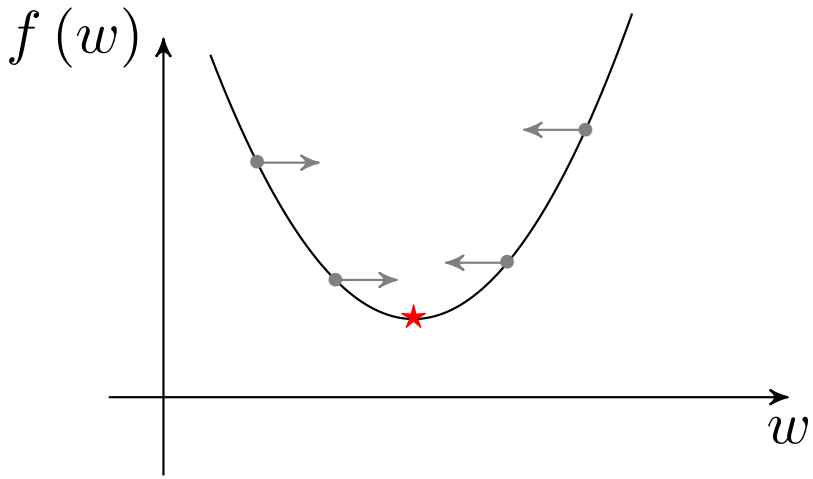
we land somewhere around it

at such points, *the algorithm always pushes us outwards*

Gradient descent practically does not get into a maximum

Behavior of Gradient Decent: *At Stationary Points*

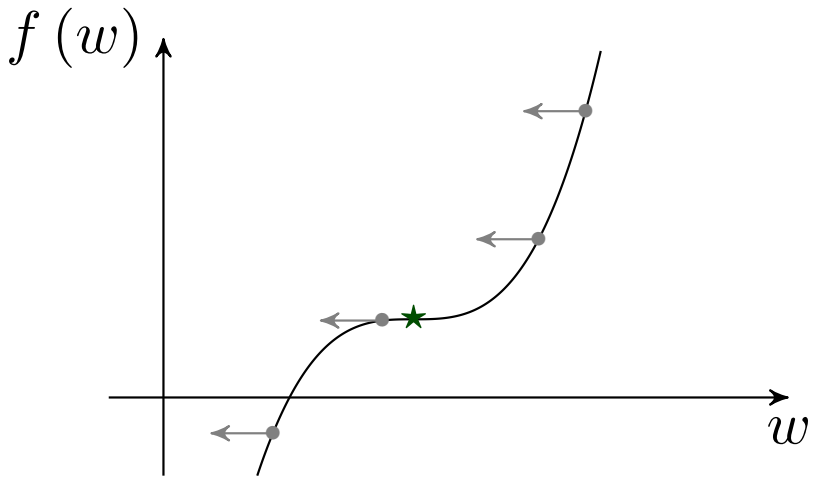
When we are around a minimum



Around minima, the algorithm always pushes us towards the minimum

Behavior of Gradient Decent: *At Stationary Points*

When we are around an inflection point



If we are exactly at an inflection; then, the algorithm stops. But, in reality

we land somewhere around it

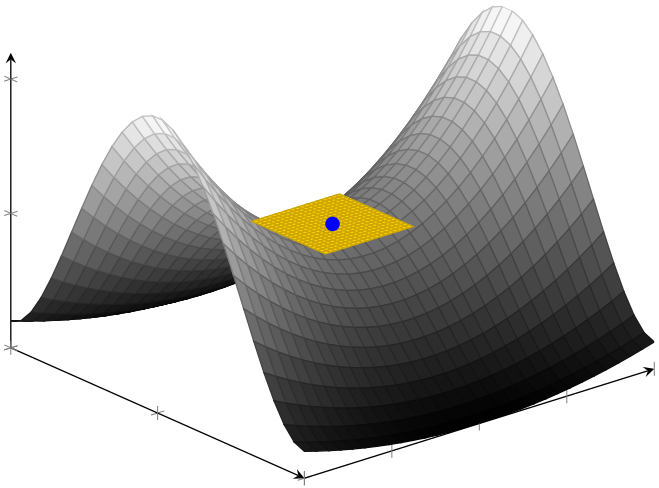
at such points, *the algorithm always pushes us somewhere else*

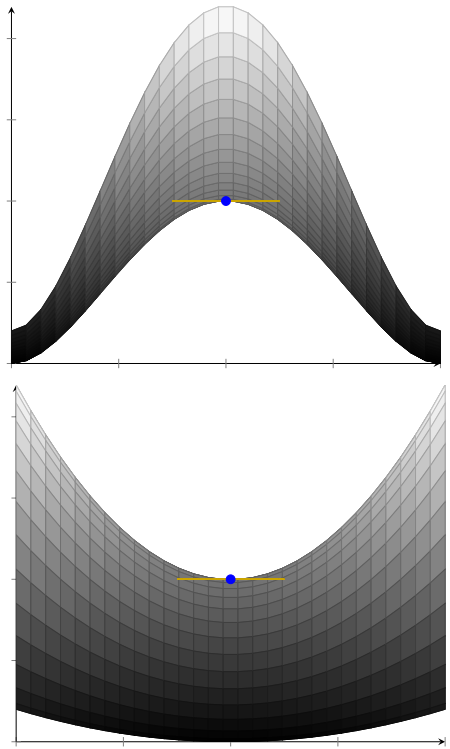
Gradient descent practically does not get into an inflection

Behavior of Gradient Decent: *At Stationary Points*

+ *Can we extend this conclusion to saddle-points?*
– Yes, but with a bit of caution!

At saddle points, function is maximized in a direction and minimized in another





Behavior of Gradient Decent: *At Stationary Points*

So, for a saddle-point we can conclude: if we are exactly at a saddle-point; then, the algorithm stops. But, in reality

we land somewhere around it

If at that point, the gradient has a component in the direction that the function is maximized; then, *the algorithm pushes us outwards*.

- \+ *Can it happen that we do not land at such point?*
- – Thinking with an engineer's mind: *Not really!*

So we could say

> *Gradient descent almost never gets trapped at a saddle-point*

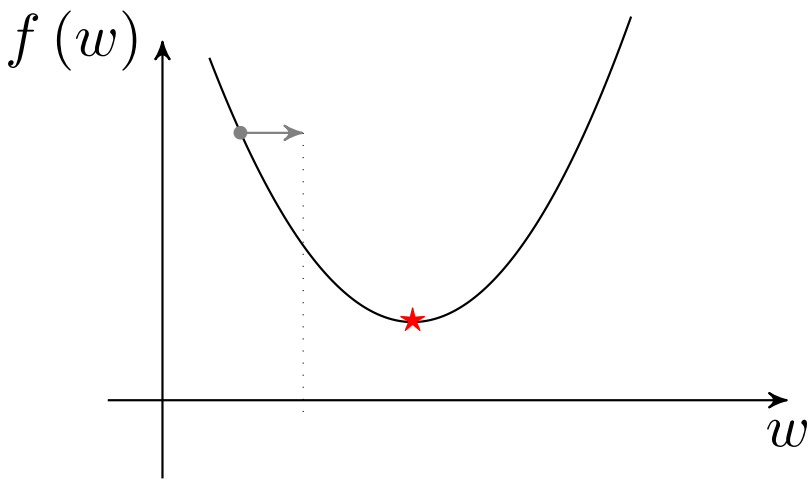
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With small *learning rates*, the algorithm *converges*;



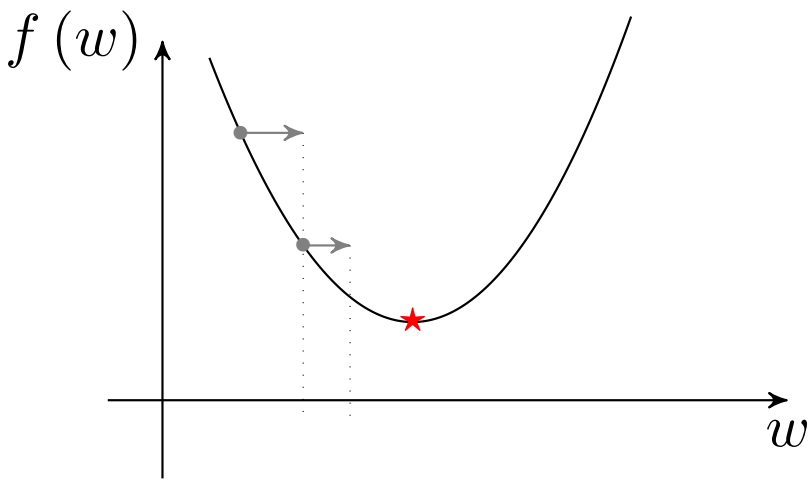
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With small *learning rates*, the algorithm *converges*;



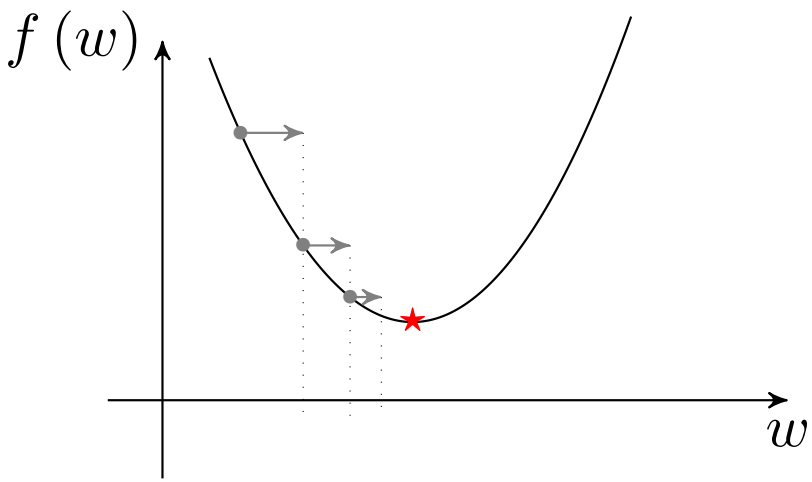
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With small *learning rates*, the algorithm *converges*;



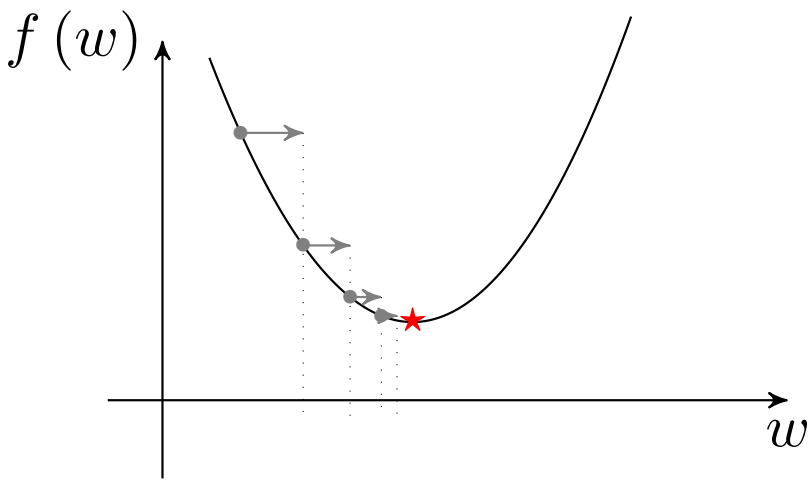
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With small *learning rates*, the algorithm *converges*;



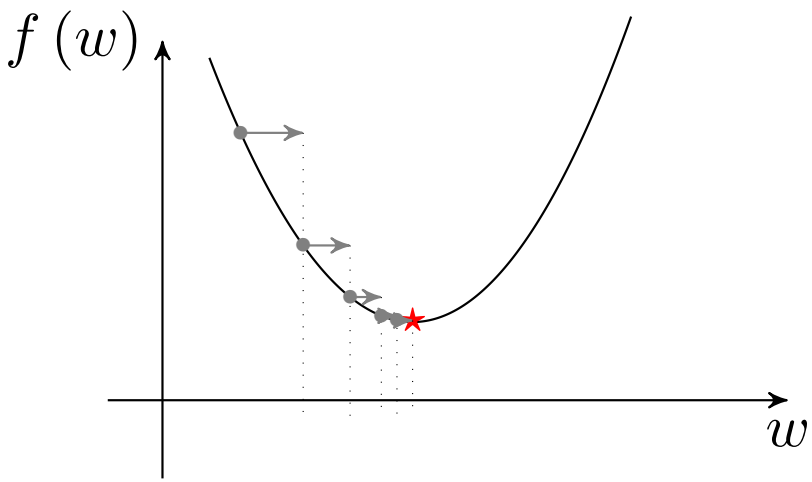
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With small *learning rates*, the algorithm *converges*;



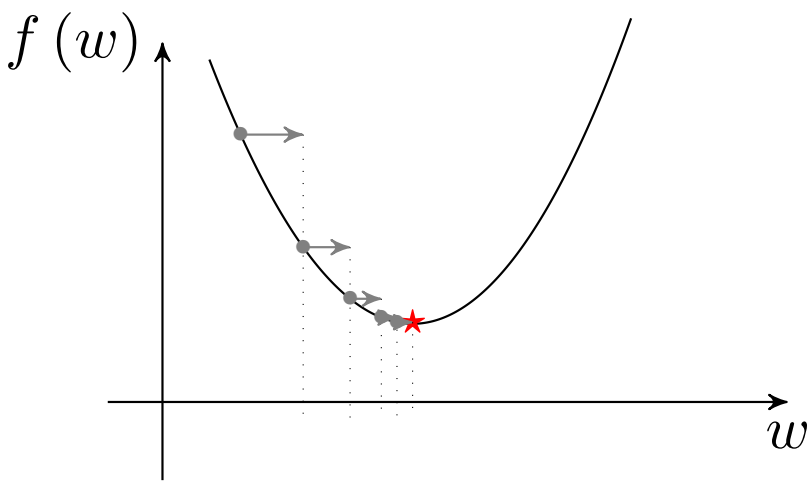
Behavior of Gradient Decent: *Convergence*

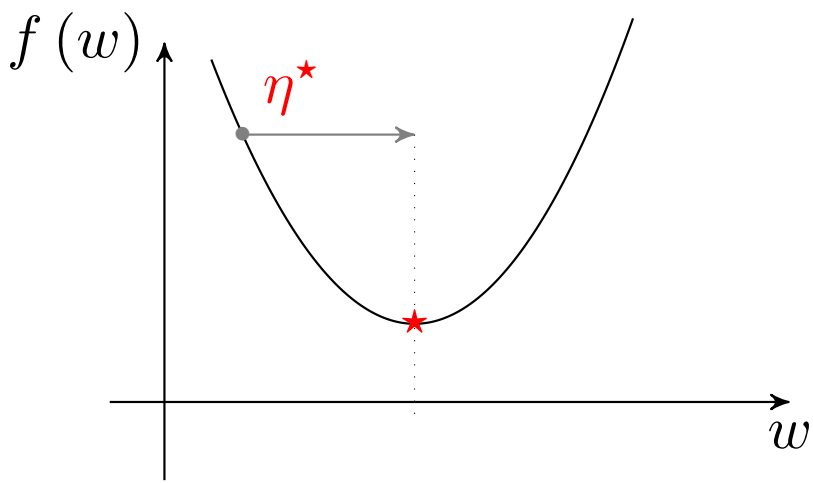
Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With small *learning rates*, the algorithm *converges*; *how small?* $\eta < \eta^\star$





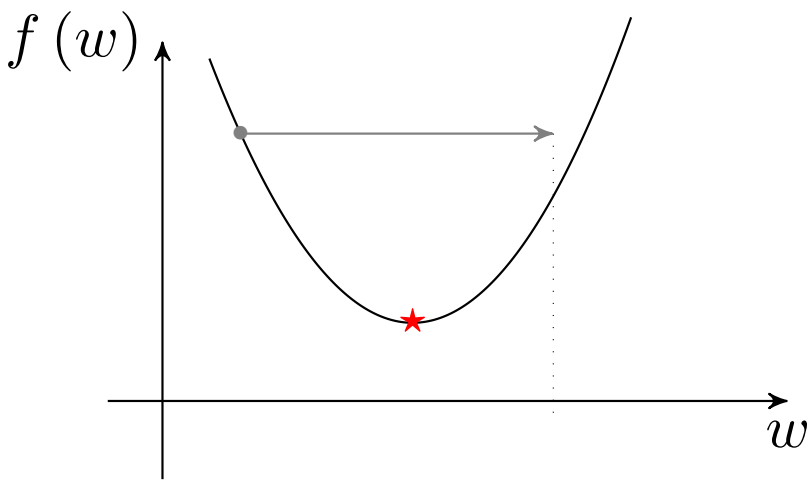
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With larger *learning rates*, the algorithm starts *oscillating*:



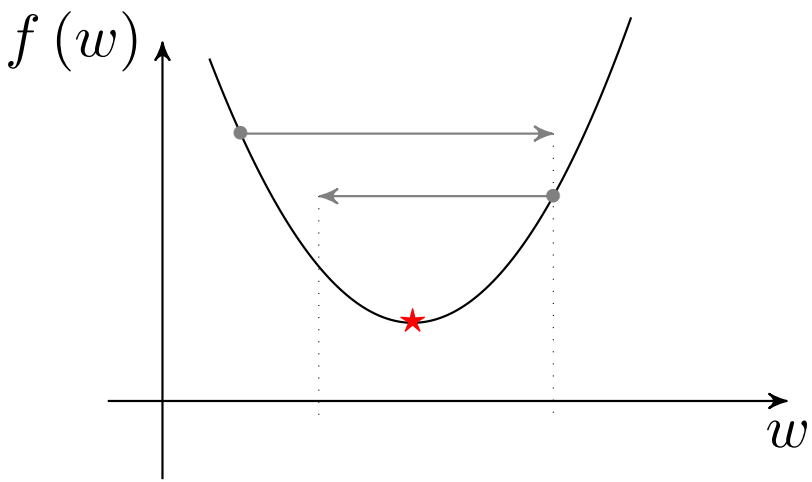
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With larger *learning rates*, the algorithm starts *oscillating*:



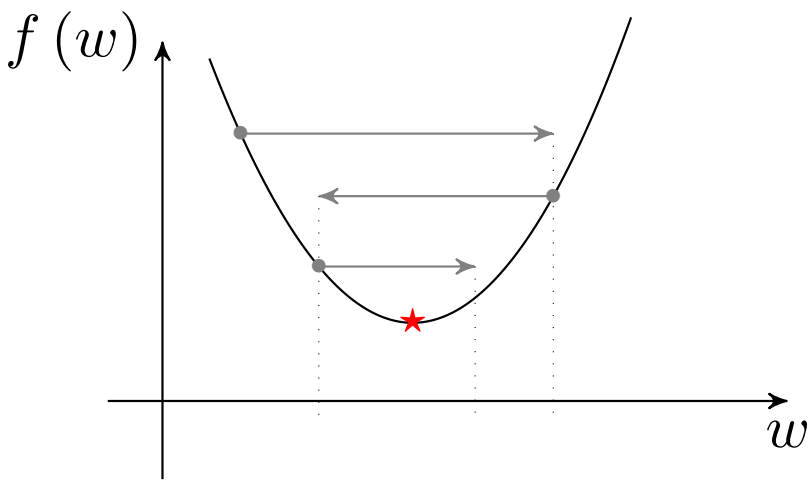
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With larger *learning rates*, the algorithm starts *oscillating*:



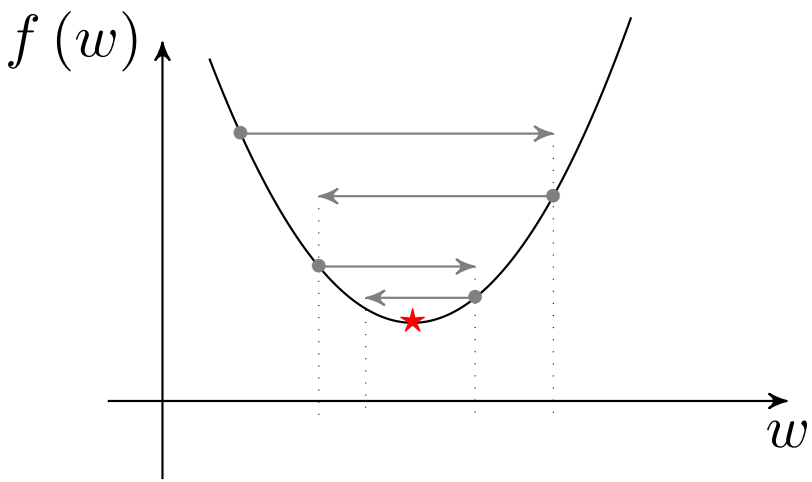
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With larger *learning rates*, the algorithm starts *oscillating*:



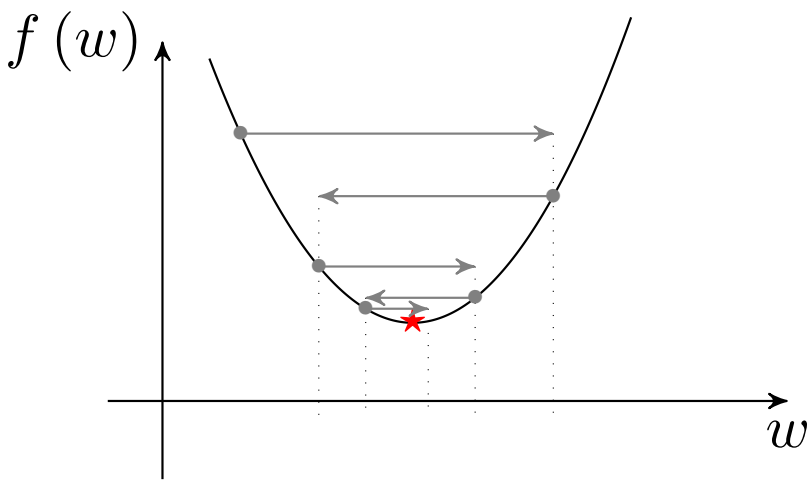
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With larger *learning rates*, the algorithm starts *oscillating*:



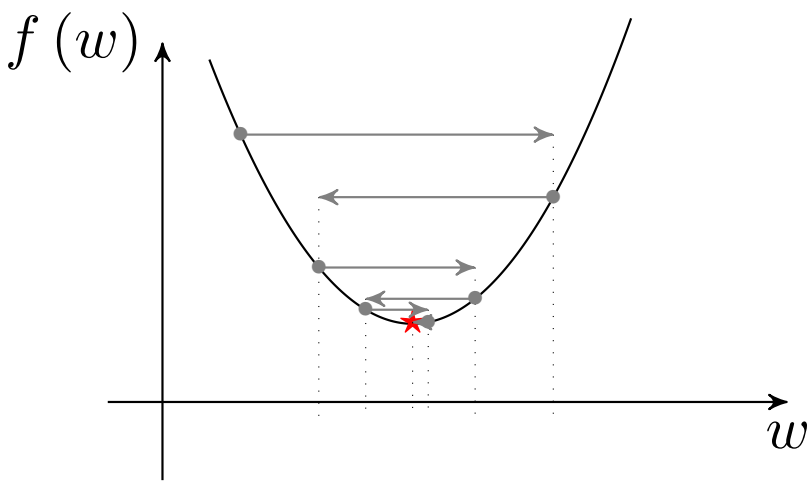
Behavior of Gradient Decent: *Convergence*

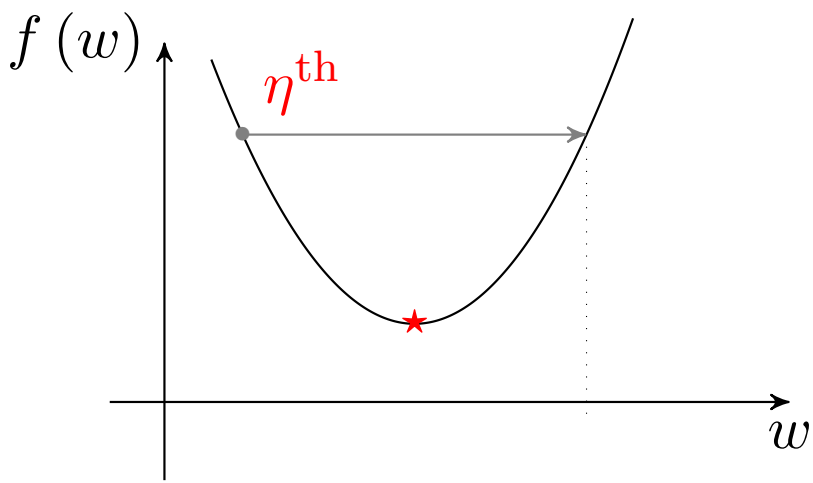
Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With larger *learning rates*, the algorithm starts *oscillating*: $\eta^\star < \eta < \eta^{\text{th}}$





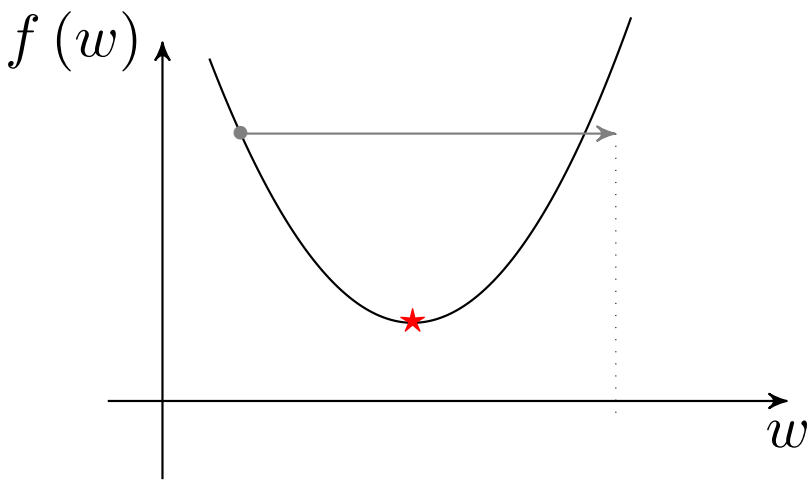
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With extremely large *learning rates*, the algorithm *diverges*:



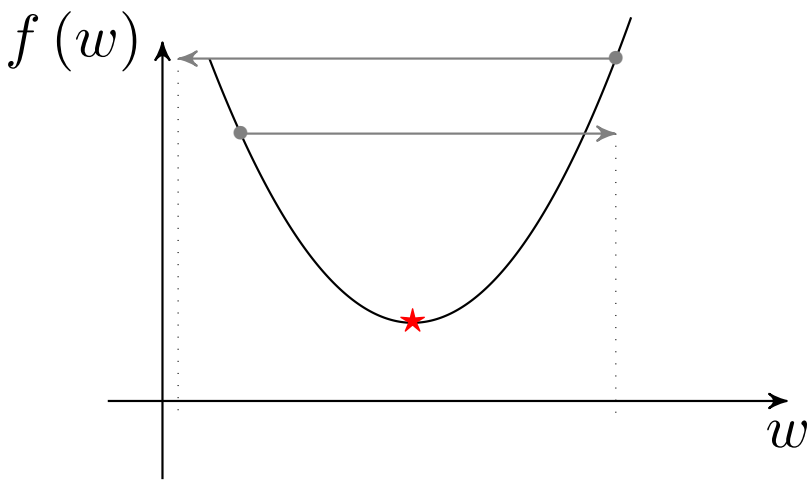
Behavior of Gradient Decent: *Convergence*

Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With extremely large *learning rates*, the algorithm *diverges*:



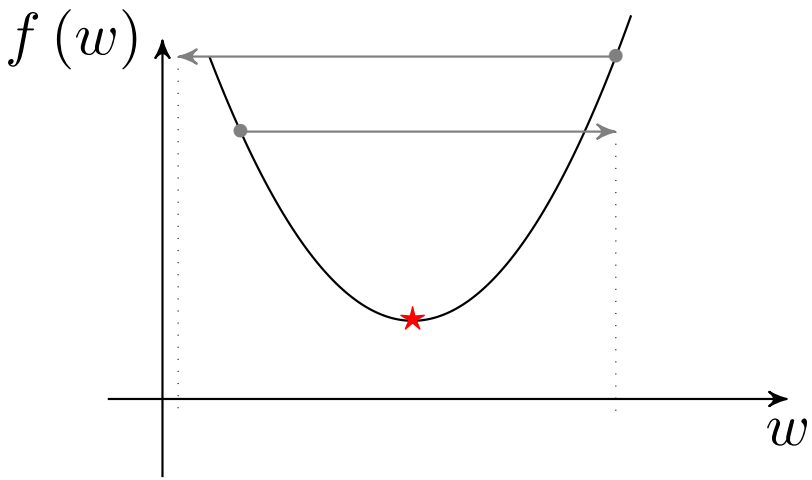
Behavior of Gradient Decent: *Convergence*

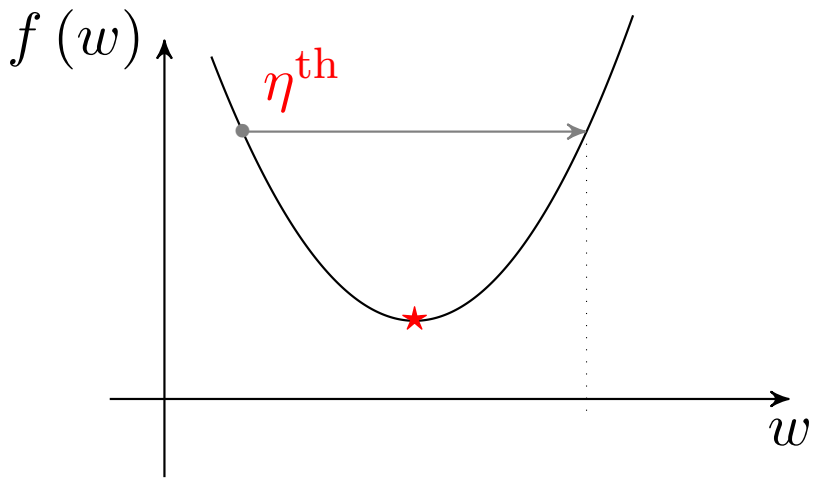
Moral of Story

Gradient descent almost never gets trapped at a point that is not minimum

+ *Nice! But, does it always converge?*
– Well! If we choose the *learning rate* properly; then, Yes!

With extremely large *learning rates*, the algorithm *diverges*: $\eta > \eta^{\text{th}}$

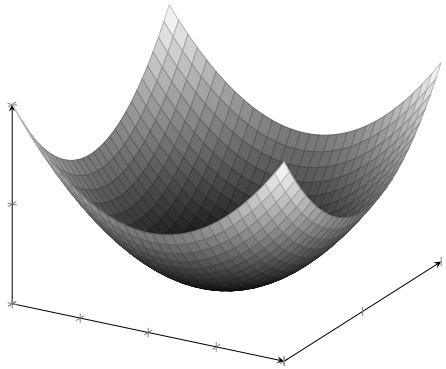


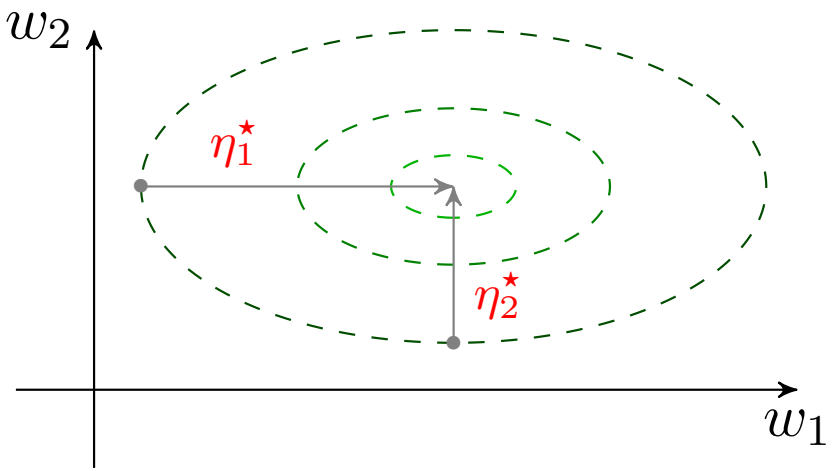


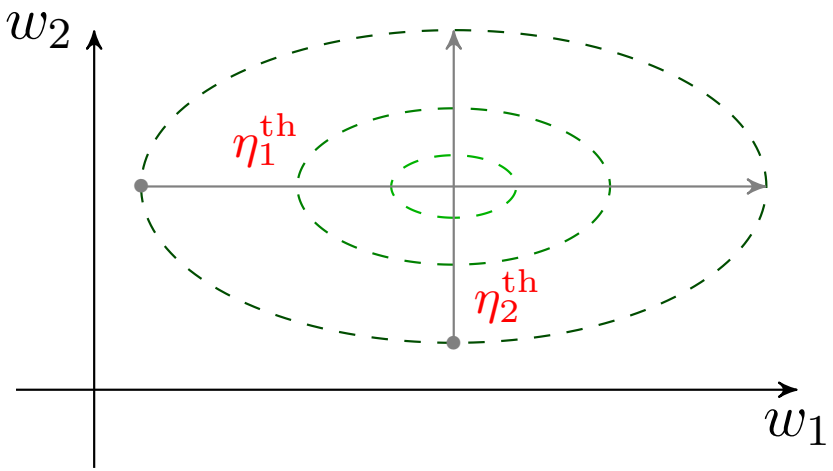
Behavior of Gradient Decent: *Convergence*

We can extend this idea to the multi-dimensional functions; however,

$\eta^\star$ and η^{th} are different on each axis







Behavior of Gradient Decent: *Convergence*

One may suggest that we use a *vector* of *learning rates*, i.e.,

$$w_n^{(t)} \leftarrow w_n^{(t-1)} - \eta_n \frac{\partial}{\partial w_n} \hat{R}(\mathbf{w}^{(t-1)})$$

for each $n = 1, \ldots, N$. *This is however not easy; the easier way is to focus on*

$$\eta^\star = \min_n \eta_n^\star \qquad \text{and} \qquad \eta^{\text{th}} = \min_n \eta_n^{\text{th}}$$

Behavior of Gradient Decent: *Convergence*

Clearly, there is always a trade-off

- We can choose a large learning rate
  - \+ Gradient descent converges faster: *high convergence speed*
  - – The chance of divergence however increases: *high divergence rate*
- We can choose a small learning rate
  - – Gradient descent converges slowly: *low convergence speed*
  - \+ The chance of divergence is now very low: *low divergence rate*

\+ *Well, say we are patient! Then, choosing a small learning rate is safe! Right?!*

– Well! If the empirical risk is convex; then, Yes! But, with non-convex risks *not always!*

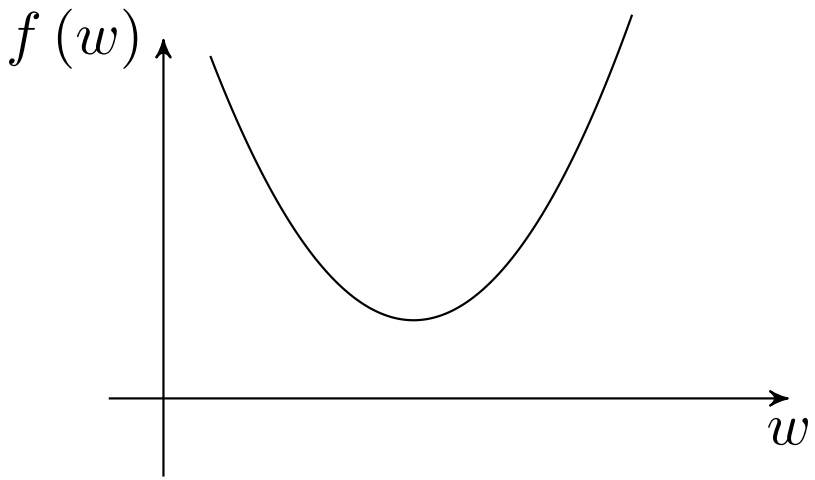
Recap: Convex Function

You really don't need to know the definition of a convex function in this course; however, just in case you're interested, here it goes:

$f(\cdot) : \mathbb{R}^N \mapsto \mathbb{R}$ *is convex, if for any two points* $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^N$, *we have*

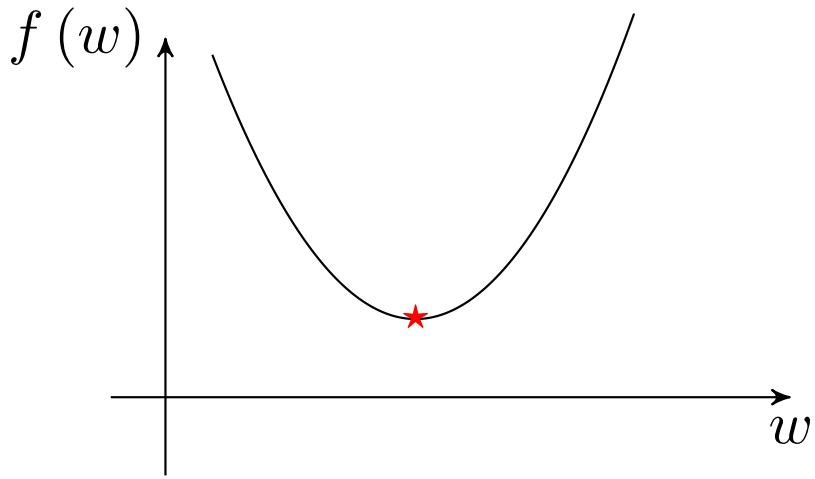
$$f(\lambda \boldsymbol{x}_1 + (1-\lambda)\,\boldsymbol{x}_2) \leqslant \lambda f(\boldsymbol{x}_1) + (1-\lambda)\, f(\boldsymbol{x}_2)$$

for all $0 \leqslant \lambda \leqslant 1$



Behavior of Gradient Decent: *Convergence*

In convex functions, we don't have disjoint local minima



So, if we choose a small learning rate

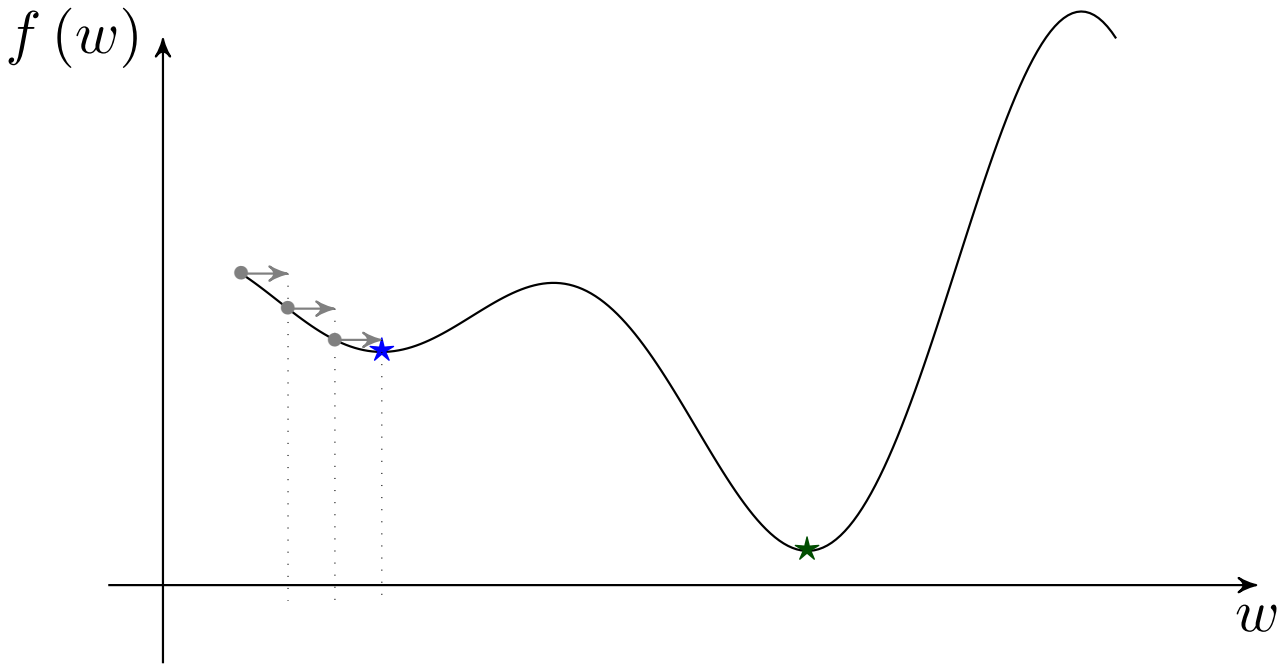
we surely converge to the global minimum

Behavior of Gradient Decent: *Convergence*

But, most empirical losses in deep learning are non-convex:

we have multiple disjoint local minima

Gradient descent converges to one of them, but not necessarily the global one



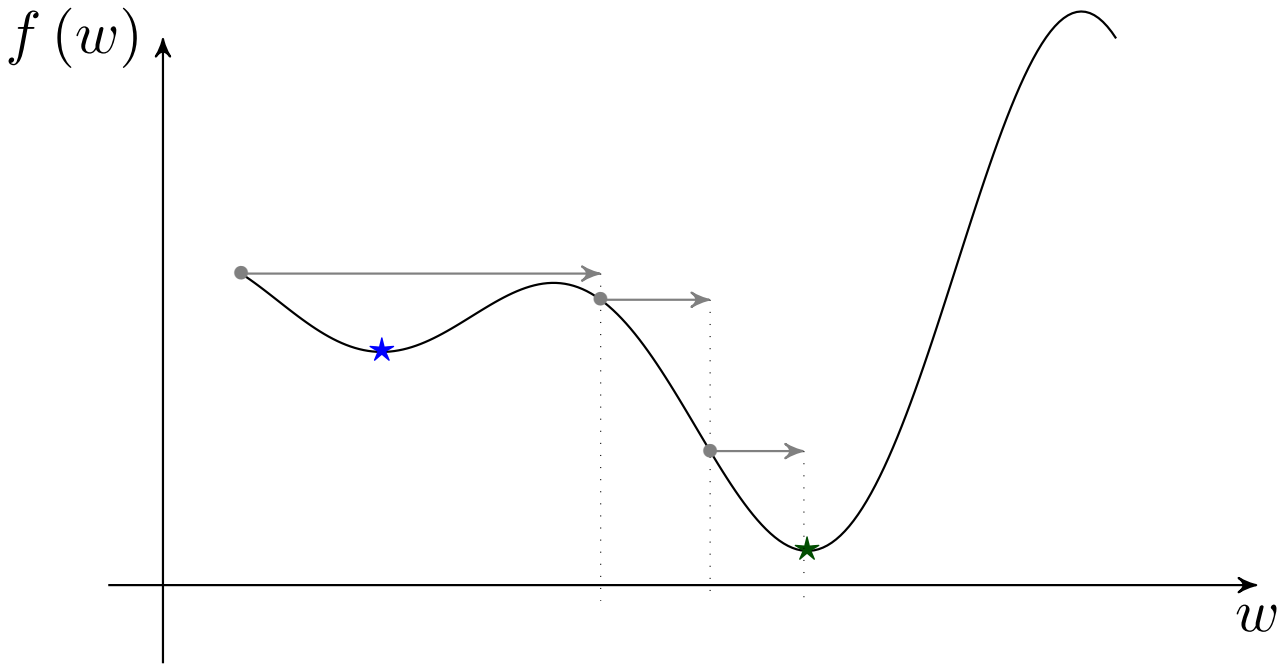
Too small learning rate can leave us in a bad local minimum!

Behavior of Gradient Decent: *Convergence*

But, most empirical losses in deep learning are non-convex:

we have multiple disjoint local minima

Gradient descent converges to one of them, but not necessarily the global one



Initial larger learning rates can take us out of bad local minima!

Behavior of Gradient Decent: *Convergence*

We can hence conclude a more general trade-off

- We can choose a large learning rate
  - \+ Gradient descent converges faster: *high convergence speed*
  - \+ Gradient descent may fall out of a local minimum: *lower risk*
  - – The chance of divergence however increases: *high divergence rate*
- We can choose a small learning rate
  - – Gradient descent converges slowly: *low convergence speed*
  - – Gradient descent traps in a local minimum: *high risk*
  - \+ The chance of divergence is now very low: *low divergence rate*

+ *How do we do it in practice?*

– In practice, we start with large learning rates and reduce it gradually as we get close to the minimum

We will talk about this more once we start training practical neural networks!

Gradient Decent: *Summary*

1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
2: Choose some small ϵ and η, and set $t = 1$
3: **while** $\Delta > \epsilon$ **do**
4: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta \nabla \hat{R}(\mathbf{w}^{(t-1)})$
5: Update the deviation $\Delta = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{(t-1)})|$
6: **end while**

Gradient descent converges almost always to a local minimum

- With convex empirical risks, this is global minimum
- With non-convex empirical risks, this is not necessarily global minimum
- Learning rate is a crucial parameter that tunes the convergence

There are other optimization algorithms that work based on gradient: we will talk about them later!