



Alice's Adventures in a **differentiable** wonderland

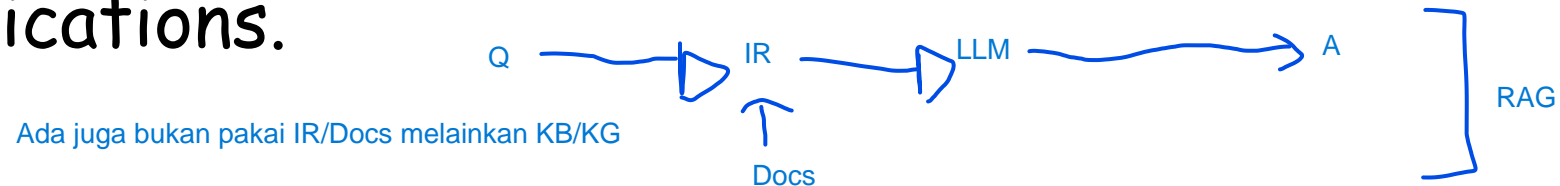
Simone Scardapane (<https://www.sscardapane.it/>)

This set of slides is prepared by **Alfan F. Wicaksono**
Information Retrieval, CS UI

* Some slides were originally made by Alfan

Mengapa Anda perlu mempelajari ini?

- **Neural Information Retrieval** is evolving quickly.
- This is an introduction to the topic of (deep) neural networks (NNs), the core technique at the hearth of large language models, generative artificial intelligence - and many other applications.



- Because the field is evolving quickly, you must strike a **good balance between theory and code**, historical considerations and recent trends.

Jadi sebenarnya context yang diberikan IR (dari docs-docs yang ada) diharapkan memberikan context lagi ke LLM agar tidak halusinasi

Ada RAG (Retrieval Augmented Generation)

Dimana bisa terjadi seperti "halusinasi" kalau pakai LLM untuk menjawab question

Supervised Learning

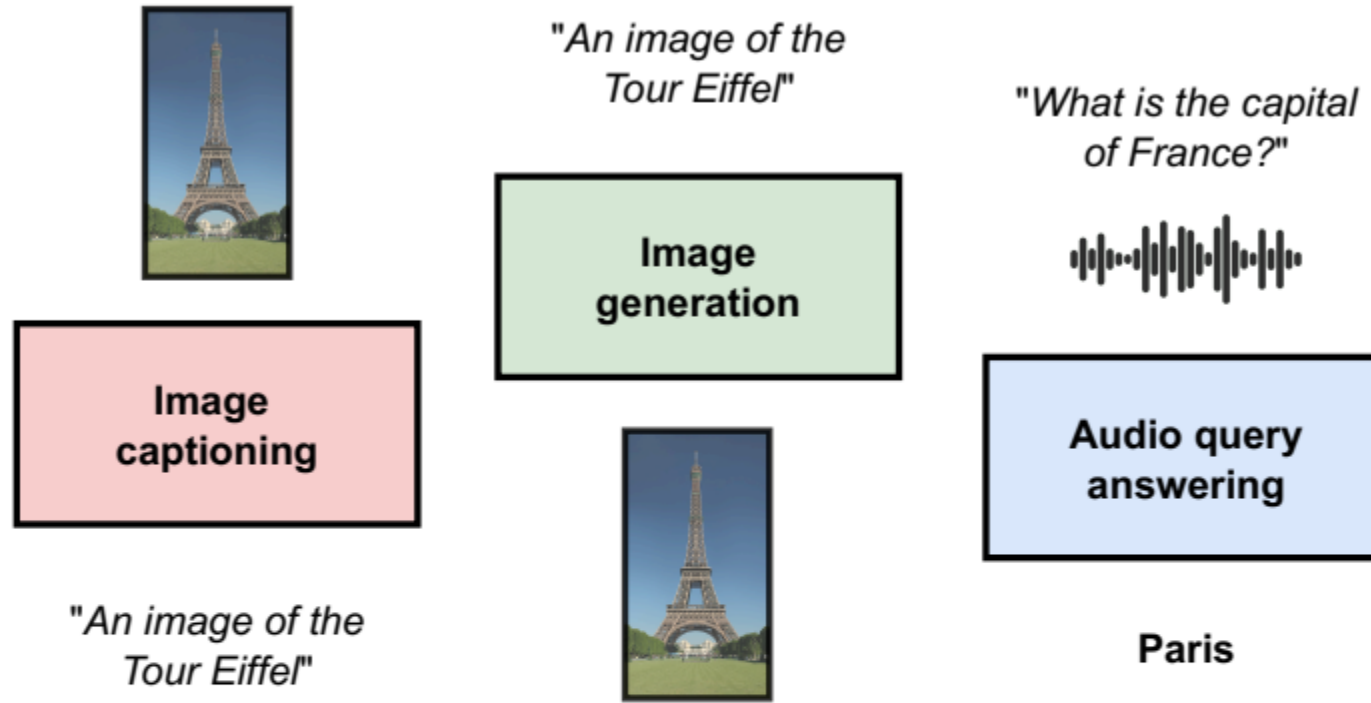


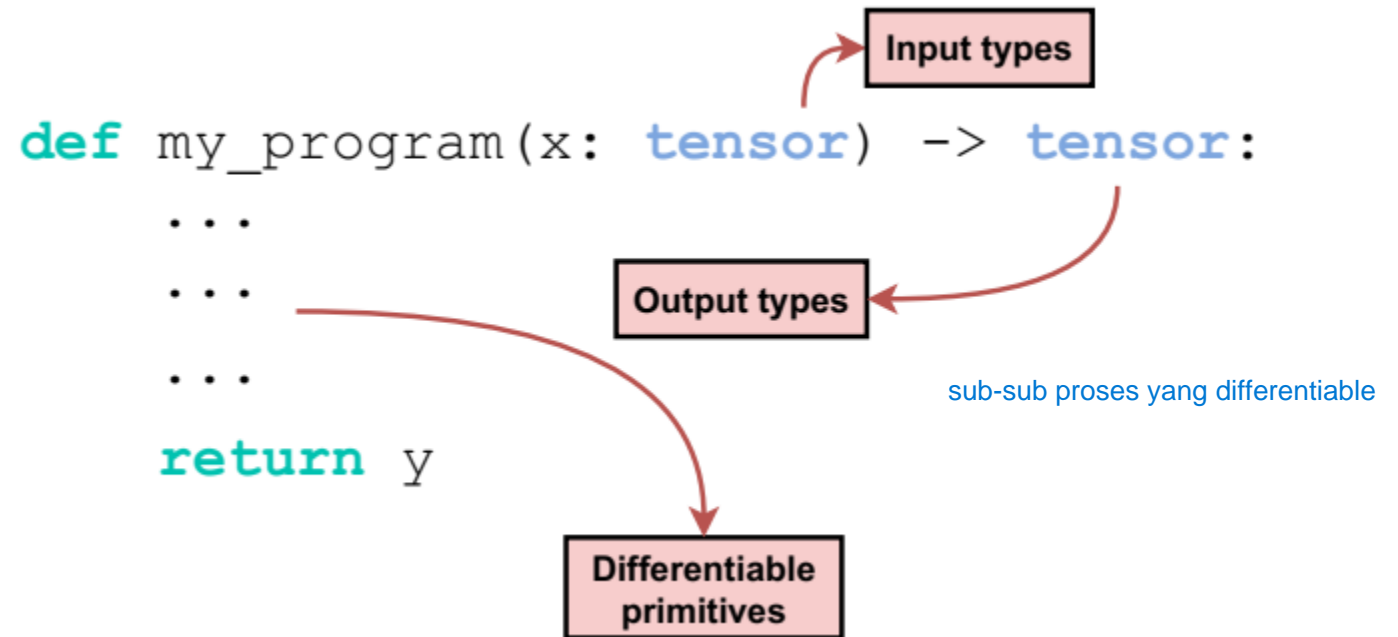
Figure F.1.2: Most tasks can be categorized based on the desired input - output we need: *image generation* wants an image (an ordered grid of pixels) from a text (a sequence of characters), while the inverse (*image captioning*) is the problem of generating a caption from an image. As another example, *audio query answering* requires a text from an audio (another ordered sequence, this time numerical). Fascinatingly, the design of the models follow similar specifications in all cases.

Learning is a **search** problem

- We start by defining a program with a large number of degree-of-freedom (that we call **parameters**), and we manipulate the parameters until the model performance is satisfying.
- As the name implies, **differentiable models** do this by restricting the selection of the model to differentiable components, i.e., **mathematical functions that we can differentiate**.

Neural Networks as Differentiable Models

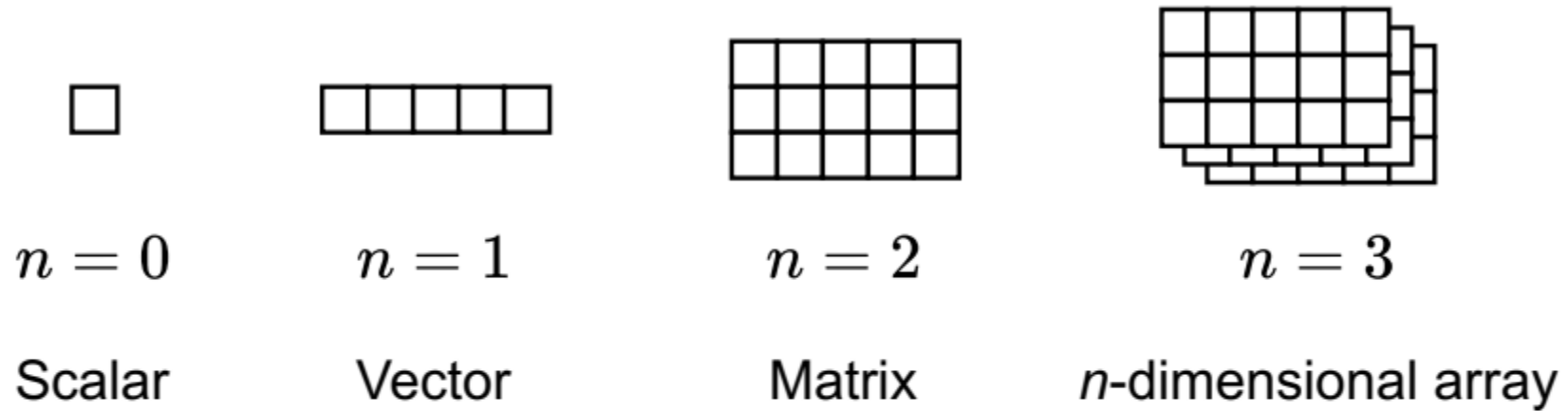
Figure E.1.3: *Neural networks are sequences of differentiable **primitives** which operate on structured arrays (**tensors**): each primitive can be categorized based on its input/output signature, which in turn defines the rules for composing them.*



Being able to compute a derivative of a high-dimensional function (a **gradient**) means **knowing what happens if we slightly perturb their parameters**, which in turn leads to **automatic routines for their optimization**.

Tensors

Tensors: Our fundamental datatypes when dealing with differentiable models.



*Fundamental data types: scalars, vectors, matrices, and generic n -dimensional arrays. We use the name **tensors** to refer to them. n is called the **rank** of the tensor. We show the vector as a row for readability, but in the text we assume all vectors are column vectors.*

Tensors and Their Operators

tipe tensor harus sama semua (float harus float, integer harus integer, dll)

Definition D.2.1 (Tensors) *A tensor X is an n -dimensional array of elements of the same type. We use $X \sim (s_1, s_2, \dots, s_n)$ to quickly denote the **shape** of the tensor.*

$X \sim S(\dots)$ -> shape/dimensi dari tensor

contoh $X = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}$

maka itu $X \sim (3, 2)$ ada 3 baris 2 kolom

Kalo X_0 itu, bakal refer ke baris (ada berapa fitur di baris tsb, contoh $X_0_matrix = [2 \ 3]$ atau $X_0_tensor \sim (2)$)

- Tensors can be indexed to get slices (subsets) of their values
- For example, for a 3-dimensional tensor $X \sim (a, b, c)$, we can write X_i to denote a slice of size (b, c) ; and X_{ijk} for a scalar.
- We use commas for more complex expressions. For example, $X_{i,:,j:k}$ to denote a slice of size $(b, k - j)$.

Tensors and Their Operators

Definition D.2.1 (Tensors) *A **tensor** X is an n -dimensional array of elements of the same type. We use $X \sim (s_1, s_2, \dots, s_n)$ to quickly denote the **shape** of the tensor.*

- Vectors $x \sim (d)$ are examples of 1-dimensional tensors.
- **In math**, we distinguish between column vectors x and row vectors x^T .
- **In code**, row and column vectors correspond to **2-dimensional** tensors of shape $(1, d)$ or $(d, 1)$, which are different from 1-dimensional tensors of shape (d) .
 - This is because most frameworks implement **broadcasting rules**

Tensors and Their Operators

Beware of **broadcasting**, resulting in a matrix output from an elementwise operation on two vectors due to their shapes.

```
import torch
```

```
x = torch.randn((4, 1)) # "Column" vector
```

```
y = torch.randn((4,)) # 1-dimensional tensor
```

```
print((x + y).shape)
```

```
# [Out]: (4,4) (because of broadcasting!)
```

```
tensor([[ -1.6426],  
        [ -1.1397],  
        [ -0.4903],  
        [ -1.3791]])
```

```
tensor([ 0.1747,  0.6879,  0.9005, -0.0122])
```

```
tensor([[ -1.4678, -0.9547, -0.7420, -1.6548],  
        [ -0.9650, -0.4518, -0.2392, -1.1519],  
        [ -0.3155,  0.1976,  0.4103, -0.5025],  
        [ -1.2043, -0.6912, -0.4785, -1.3912]])
```

broadcasting != perkalian

Tensors and Their Operators

If we understand a vector as a point in **d-dimensional Euclidean space**, the **distance of a vector from the origin** is given by the Euclidean (**ℓ_2**) norm:

$$\|\mathbf{x}\| = \sqrt{\sum_i x_i^2}$$

Tensors and Their Operators

(Inner Product)

The inner product between two vectors $\mathbf{x}, \mathbf{y} \sim (d)$ is given by:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\top \mathbf{y} = \sum_i x_i y_i$$

$$\mathbf{x} = [0.1, 0, -0.3] \quad \mathbf{y} = [-4.0, 0.05, 0.1]$$

$$\langle \mathbf{x}, \mathbf{y} \rangle = -0.4 + 0 - 0.03 = -0.43$$

Tensors and Their Operators

In the 2-D case we have matrices:

$$\mathbf{X} = \begin{bmatrix} X_{11} & \cdots & X_{1d} \\ \vdots & \ddots & \vdots \\ X_{n1} & \cdots & X_{nd} \end{bmatrix} \sim (n, d)$$

A matrix can be understood as a **stack of n vectors**

$(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$, where the **stack** is organized in a row-wise fashion:

ini kayak di compressed gitu $X_{11} \dots X_{1d}$

menjadi \mathbf{X}^T

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

We say that \mathbf{X} represents a **batch of data vectors**.

Tensors and Their Operators

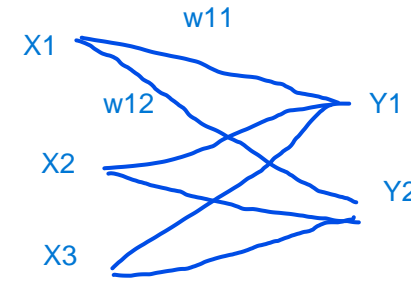
Definition D.2.3 (Matrix multiplication) Given two matrices $\mathbf{X} \sim (a,b)$ and $\mathbf{Y} \sim (b,c)$, matrix multiplication $\mathbf{Z} = \mathbf{XY}$, with $\mathbf{Z} \sim (a,c)$ is defined element-wise as:

$$Z_{ij} = \langle \mathbf{X}_i, \mathbf{Y}_j^\top \rangle \quad (\text{E.2.4})$$

i.e., the element (i, j) of the product is the dot product between the i -th row of \mathbf{X} and the j -th column of \mathbf{Y} .

[kayak matrix multiplication biasa](#)

Tensors and Their Operators



intinya

$Y_1 \rightarrow W_{11} * x_1 + W_{12} x_2 + W_{13} * x_3$

nah ini bisa dipisah jadi matrix * vector kayak

(Matrix-Vector Product)

As a special case, if the second term is **a vector x** we have a matrix-vector product:

$$z = Wx$$

If we have a **batch of vectors X** (big X , not small x), then the following matrix multiplication:

$$XW^T$$

is a simple vectorized way of computing **n dot products**, as in **$z = Wx$** .

Tensors and Their Operators

Reduction operations (sum, mean, ...) across axes.

```
>>> import torch
>>> a = torch.randn((2,3))
>>> a
tensor([[ -1.0082,  0.1171,  0.3537],
        [ 1.0082,  0.9193, -0.1909]])

>>> torch.sum(a, dim=0) dimensi 0 itu dari atas kebawah
tensor([-2.1932, -2.3708, -2.3922])

>>> torch.sum(a, dim=1) dimensi 1 itu dari kiri ke kanan.
tensor([-2.9999, -3.9565])

>>> torch.sum(a)
tensor(-6.9563)
```


Tensors and Their Operators

Batched Matrix Multiplication (BMM)

Consider two tensors $X \sim (n, a, b)$ and $Y \sim (n, b, c)$.
Batched matrix multiplication (BMM) is defined as:

maksudnya X ada n buah matriks yang punya dimensi axb

$$[\text{BMM}(X, Y)]_i = X_i Y_i \sim (n, a, c)$$

```
X = torch.randn((4, 5, 2))
Y = torch.randn((4, 2, 3))
(torch.matmul(X, Y)).shape # Or X @ Y
# [Out]: (4, 5, 3)
```

Box C.2.3: *BMM in PyTorch is equivalent to standard matrix multiplication. Practically every operation is implemented to run on generically batched inputs.*

Gradients & Jacobians

As the name differentiable implies, gradients play a pivotal role, by allowing us to optimize our models through semi-automatic mechanisms deriving from gradient descent.

Derivatives of scalar functions

Consider a simple function $y = f(x)$ with a **scalar input** and a **scalar output**.

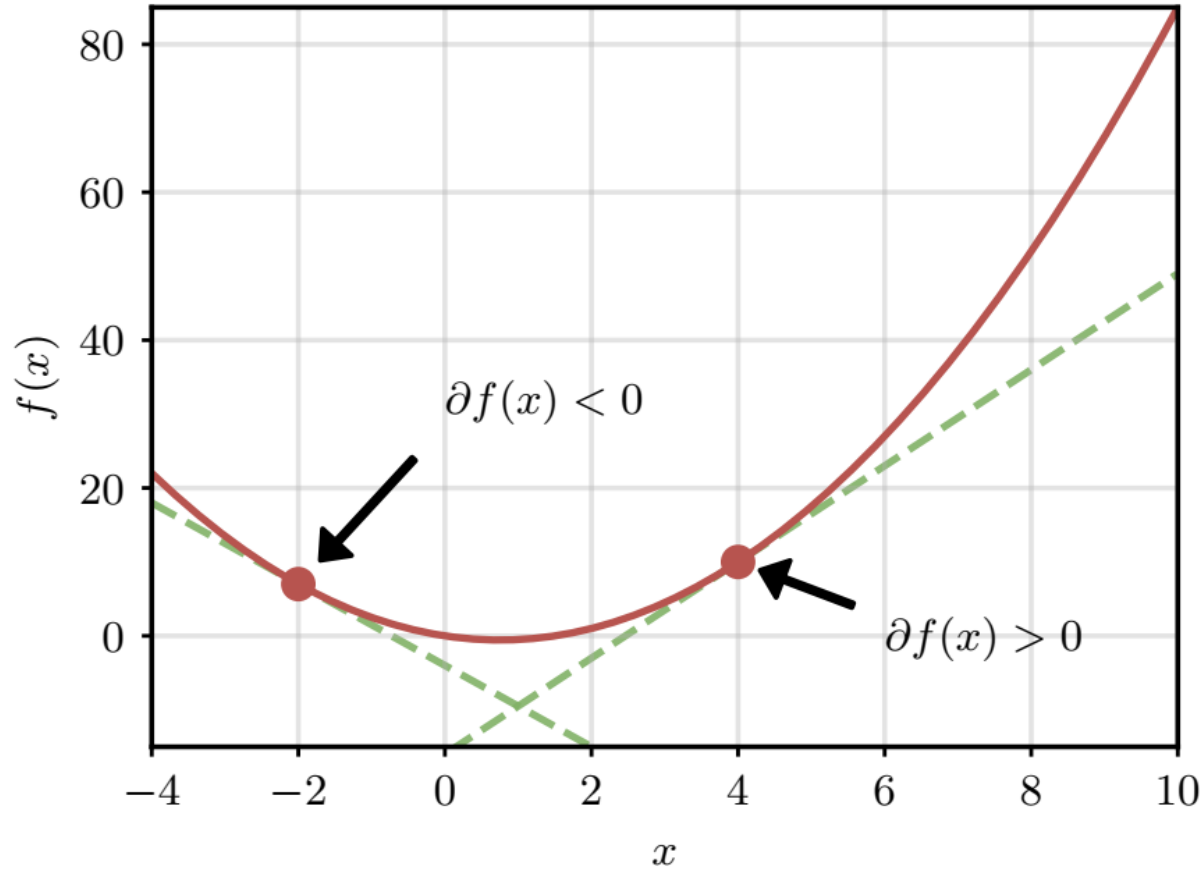
Definition D.2.5 (Derivative) *The derivative of $f(x)$ is defined as:*

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (\text{E.2.14})$$

We use a variety of notation to denote derivatives: ∂ will denote generically derivatives and gradients of any dimension (vectors, matrices); ∂_x or $\frac{\partial}{\partial x}$ to highlight the input argument we are differentiating with respect to (when needed); while $f'(x)$ is specific to scalar functions and it is sometimes called *Lagrange's notation*.

Derivatives of scalar functions

Plot of the function $f(x) = x^2 - 1.5x$



Geometrically, the derivative can be understood as **the best first-order approximation** of the function itself in that point.

The **slope of the line** tells us how the function is evolving in a close neighborhood: for **a positive slope**, the function is increasing on the right and decreasing on the left (again, for **a sufficiently small interval**), while for **a negative slope** the opposite is true.

Derivatives of scalar functions

Some properties ...

Linearity $\partial[f(x) + g(x)] = f'(x) + g'(x).$

Product Rule $\partial[f(x)g(x)] = f'(x)g(x) + f(x)g'(x),$

Chain Rule $\partial[f(g(x))] = f'(g(x))g'(x)$

Directional derivatives

Consider a function $y = f(\mathbf{x})$ taking a vector $\mathbf{x} \sim (d)$ as input and a **scalar output**.

In the scalar case we only had “left” and “right” infinitesimal perturbations. In this case, we have **infinite possible directions in the Euclidean space**.

$$\partial_{x_i} f(\mathbf{x}) = \frac{\partial y}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h},$$

This is called **partial derivatives**.

where $\mathbf{e}_i \sim (d)$ is the i -th basis vector: $[\mathbf{e}_i]_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$

Directional derivatives

Stacking all **partial derivatives** together gives us a d-dimensional vector called the **gradient** of the function.

Definition D.2.6 (Gradient) *The gradient of a function $y = f(\mathbf{x})$ is given by:*

$$\nabla f(\mathbf{x}) = \partial f(\mathbf{x}) = \begin{bmatrix} \partial_{x_1} f(\mathbf{x}) \\ \vdots \\ \partial_{x_d} f(\mathbf{x}) \end{bmatrix}$$

Directional derivatives

What about displacements in a general direction \mathbf{v} ? In this case we obtain the **directional derivative**:

$$D_{\mathbf{v}}f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h},$$

Movement in space can be decomposed by considering individual displacements along each axis:

$$D_{\mathbf{v}}f(\mathbf{x}) = \langle \nabla f(\mathbf{x}), \mathbf{v} \rangle = \sum_i \partial_{x_i} f(\mathbf{x}) v_i$$

dot product
gradien itu terharap basis

kalau terhadap umum (any direction \mathbf{v}).

Kita hanya perlu dot product gradiennya

knowing how to compute the gradient of a function is enough to compute all possible directional derivatives

Displacement on the i -th axis

Jacobians

Let us now consider the generic case of a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ with a **vector input** $\mathbf{x} \sim (d)$ as before, and this time a **vector output** $\mathbf{y} \sim (o)$.

We compute a gradient for each output, and **their stack provides an (o, d) matrix** we call the Jacobian of \mathbf{f} .

Definition D.2.7 (Jacobian) *The **Jacobian** matrix of a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$, $\mathbf{x} \sim (d)$, $\mathbf{y} \sim (o)$ is given by:*

$$\partial \mathbf{f}(\mathbf{x}) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_o}{\partial x_1} & \cdots & \frac{\partial y_o}{\partial x_d} \end{pmatrix} \sim (o, d) \quad (\text{E.2.21})$$

Jacobians

The Jacobian of a composition of functions is now a matrix multiplication of the corresponding individual Jacobians:

karena parameter x itu vector yang diterima g , terus g juga nerima vector. Artinya kalau $g(x)$ yang merupakan vektor masuk sebagai parameter f , maka $f(x)$ juga vektor.

$$\partial [f(g(\mathbf{x}))] = [\partial f(\bullet)] \partial g(\mathbf{x})$$

cara menghitung gradien untuk setiap matrix jacobian adalah matriks yang berantai (matrix multiplication)

where the first derivative is evaluated in $g(\mathbf{x}) \sim (\mathbf{h})$.

jadi ini itu kek backward multiplication dari matriks individual jacobianya. Nah intinya nanti semua perkalian jacobian kana menghasilkan nilai seluruh komposisinya.

Perkaliannya itu dari belakang sampai depan.

Jacobians

Taylor's Theorem: The gradient is the best “first-order approximation”.

For a point \mathbf{x}_0 , the best linear approximation in an infinitesimal neighborhood of $f(\mathbf{x}_0)$ is given by:

$$\tilde{f}(\mathbf{x}) = f(\mathbf{x}_0) + \langle \partial f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle$$

Slope of the line

Displacement from \mathbf{x}_0

Jacobians

Taylor's Theorem: The gradient is the best “first-order approximation”.

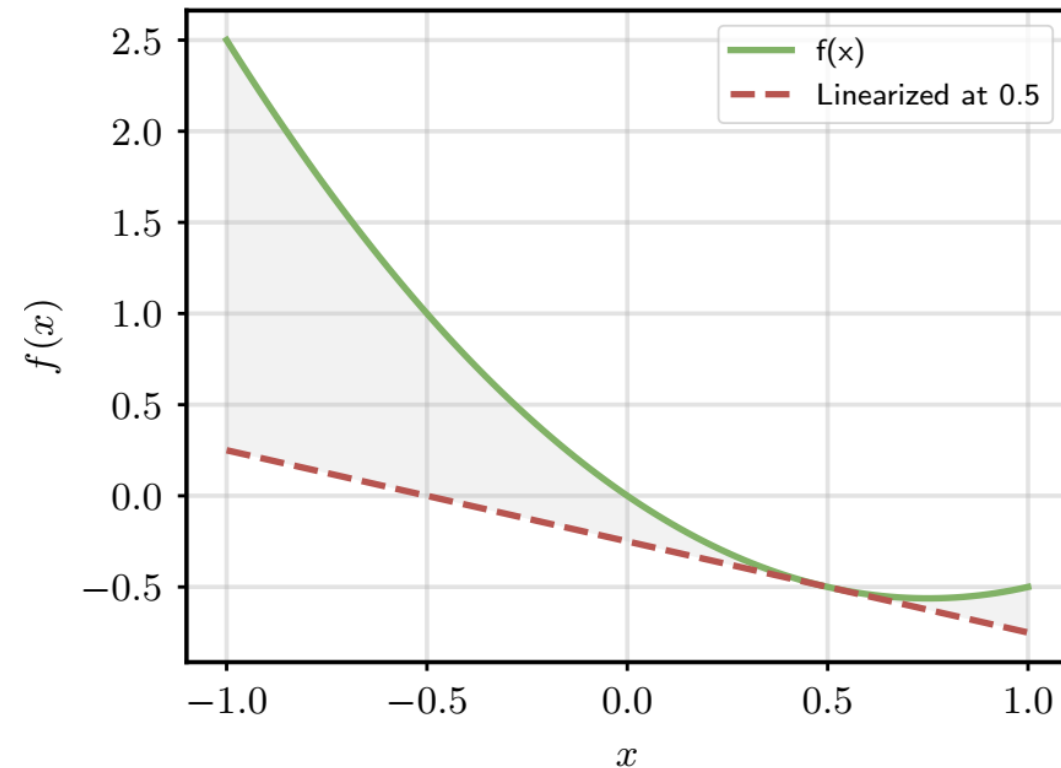
Example of computing a first-order approximation (scalar case) at $x = 0.5$

```
# Generic function
f = lambda x: x**2-1.5*x

# Derivative (computed manually for now)
df = lambda x: 2*x-1.5

# Linearization at 0.5
x=0.5
f_linearized = lambda h: f(x) + df(x)*(h-x)

# Comparing the approximation to the real derivative
print(f(x + 0.01))           # [Out]: -0.5049
print(f_linearized(x + 0.01)) # [Out]: -0.5050
```



Jacobians

Practice

Consider the following function:

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

Where $\mathbf{x} \sim (d)$, $\mathbf{y} \sim (o)$, $\mathbf{W} \sim (o, d)$.

Show that:

$$\partial_{\mathbf{x}} [\mathbf{W}\mathbf{x}] = \mathbf{W}$$

Numerical Optimization & Gradient Descent

Optimization Problems

To understand the usefulness of having access to gradients, consider the problem of **minimizing** a generic function **$f(x)$** , with **$x \sim (d)$** :

$$x^* = \arg \min_x f(x)$$

mau coba di nol kan

where, similarly to **argmax**, **argmin $f(x)$** denotes the operation of finding the value of **x** corresponding to the **lowest possible** value of **$f(x)$** .

We assume: harus ada

- the function has **a single output** (single-objective optimization), and that
- the domain over which we are optimizing **x** is **unconstrained**.

x gak boleh kayak di constraint (kayak x harus diantara 0 atau 1)

Optimization Problems

How? Closed-form? → **very rare!** jarang banget ada formula deskriptif. (Basically susah cari formula yg mendapatkan X terkecil)

In general, however, we are forced to resort to iterative procedures, starting from random guess \mathbf{x}_0 . oleh karena itu ada gradient descent untuk mendapatkan

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \eta_t \mathbf{p}_t$$

Guess at iteration t

A direction.

We expect a “**descent direction**”: a direction for which there exists an η_t such that $f(\mathbf{x}_t) \leq f(\mathbf{x}_{t-1})$.

pt terbaik adalah negative gradient

Displacement at iteration t

p itu adalah sebuah arah

Step size or learning rate

Optimization Problems

For **differentiable functions**, we can precisely quantify all **descent directions** by using the **directional derivative**.

D = directional derivative

$$\mathbf{p}_t \text{ is a descent direction} \Rightarrow D_{\mathbf{p}_t} f(\mathbf{x}_{t-1}) \leq 0$$

We have learned that:

\mathbf{p}_t dianggap turun (atau negative) apabila kalau $D_{\mathbf{p}_t}$ nya dibawah 0

Jadi kita cari \mathbf{p}_t (arah yang terbaik) untuk meminimalkan fungsi yang ada.

$$D_{\mathbf{p}_t} f(\mathbf{x}_{t-1}) = \langle \nabla f(\mathbf{x}_{t-1}), \mathbf{p}_t \rangle = \|\nabla f(\mathbf{x}_{t-1})\| \|\mathbf{p}_t\| \cos(\alpha)$$

The lowest possible value is when: $\mathbf{p}_t = -\nabla f(\mathbf{x}_{t-1})$

This is called **steepest descent direction (negative gradient)**

Gradient Descent

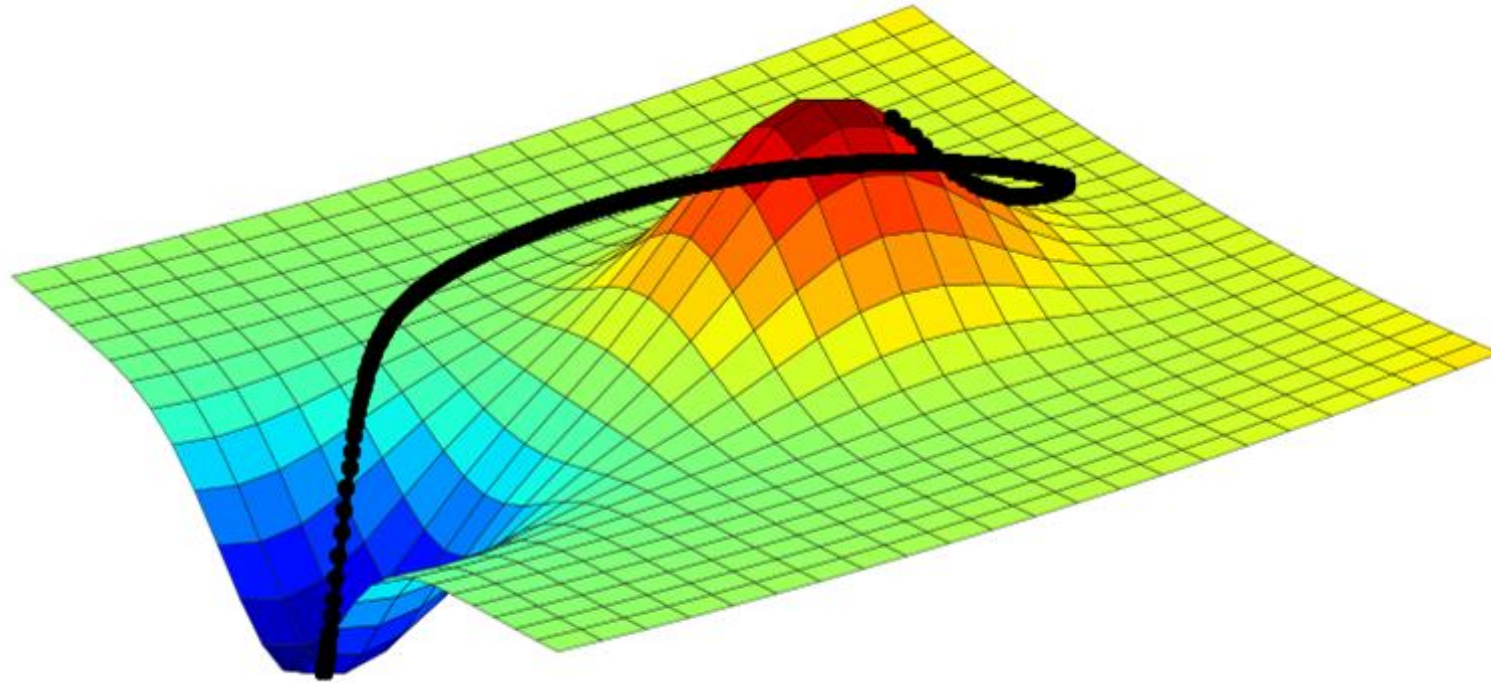
Combining them together! And we got an algorithm to minimize any differentiable function using **(steepest) gradient descent**.

karena kita memilih jalur paling baik untuk mencari yang terbaik

Definition D.2.8 ((Steepest) Gradient descent) *Given a differentiable function $f(\mathbf{x})$, a starting point \mathbf{x}_0 , and a step size sequence η_t , **gradient descent** proceeds as:*

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla f(\mathbf{x}_{t-1})$$

Gradient Descent: salah satu teknik untuk mencari parameter yang meminimalkan sebuah fungsi (tidak dijamin global minima).



Gradient Descent

Contoh: carilah parameter x sehingga fungsi $f(x) = 2x^4 + x^3 - 3x^2$ memberikan hasil minimal (local minima)!

epoch: max iteration

for $t = \langle 1, 2, 3, \dots, N_{\max} \rangle :$

$$f'(x) = 8x^3 + 3x^2 - 6x$$

$$x_{t+1} := x_t - \alpha_t f'(x_t)$$

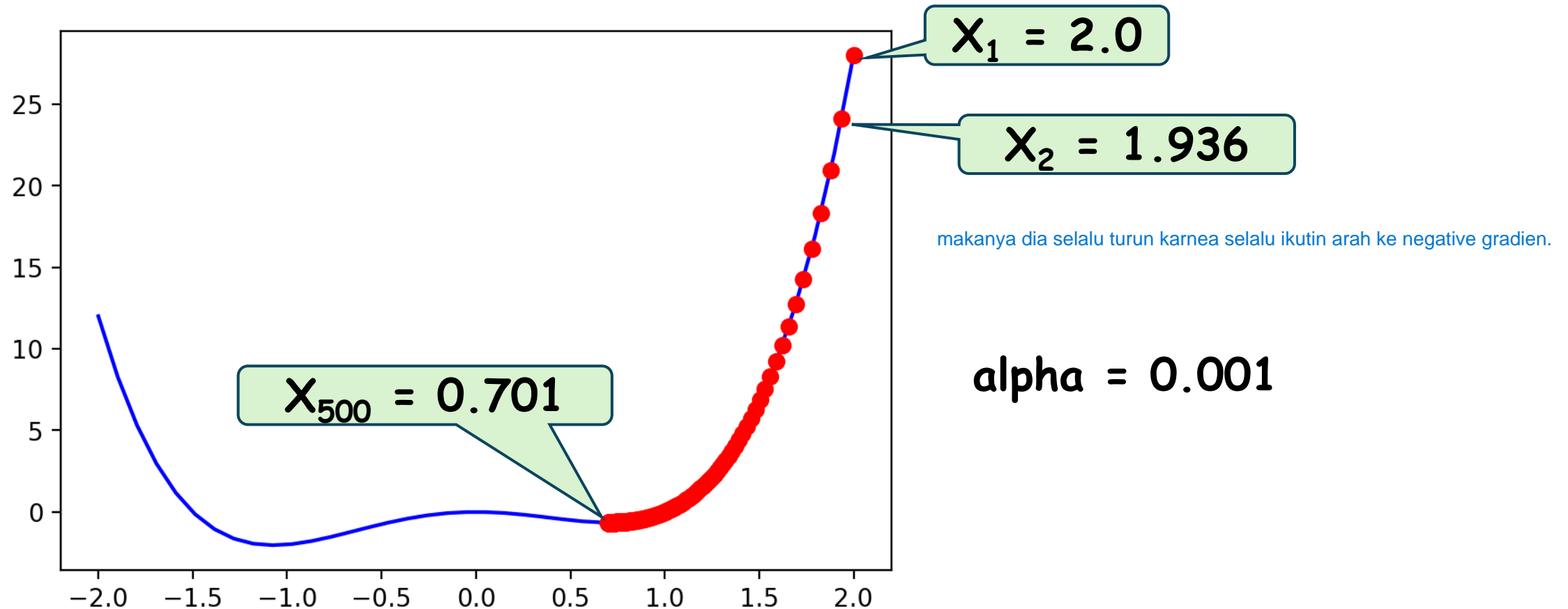
if $|f'(x_{t+1})| < \epsilon$ then return "converged on critical point"

if $|x_t - x_{t+1}| < \epsilon$ then return "converged on x value"

learning
rate

Gradient Descent

Contoh: carilah parameter x sehingga fungsi $f(x) = 2x^4 + x^3 - 3x^2$ memberikan hasil minimal (local minima)!



```
def poly(x):  
    return (2*x*x*x*x) + (x*x*x) - (3*x*x)  
  
def deriv_poly(x):  
    return (8*x*x*x) + (3*x*x) - (6*x)  
  
EPOCH = 500  
ALPHA = 0.001  
xs = [0.] * (EPOCH + 1)  
xs[0] = 2.0 # kita coba mulai dari point x = 2  
  
for t in range(EPOCH):  
    xs[t + 1] = xs[t] - ALPHA * deriv_poly(xs[t])  
  
print("optimum at x = {} ---> y = {}".format(xs[-1], poly(xs[-1])))
```

https://colab.research.google.com/drive/1_jPMRCngeBY73rhrAAWOLHd9vHJsGdvS#scrollTo=zF7ychipCyvg

Dengan beberapa populer tools seperti PyTorch, perhitungan gradient lebih mudah.

```
import torch

def poly(x):
    return (2*x*x*x*x) + (x*x*x) - (3*x*x)

EPOCH = 100
ALPHA = 0.01

x = torch.tensor(2.0, requires_grad = True)

for i in range(EPOCH):
    y = poly(x)
    y.backward() # hitung gradient dengan AUTODIF

    with torch.no_grad(): # operasi di dalam with tidak akan track gradient
        grad = x.grad
        x -= grad * ALPHA
        print ('y = {:.3f}, x = {:.3f}, grad = {:.3f}'.\
                format(y.data, x.data, grad.data))
    x.grad.zero_() # set gradient kembali ke nol
```

AUTODIF!

Kita akan belajar nanti nanti!

Gradient Descent

Konvergenzi gradient descent

Convergence of gradient descent??

First, you need to understand local minima, global minima, saddle point, convexity, etc.

Definition D.2.9 (Minimum) *A local minimum of $f(\mathbf{x})$ is a point \mathbf{x}^+ such that the following is true for some $\varepsilon > 0$:*

$$f(\mathbf{x}^+) \leq f(\mathbf{x}) \quad \forall \mathbf{x} : \|\mathbf{x} - \mathbf{x}^+\| < \varepsilon$$

Ball of size ε centered in \mathbf{x}^+

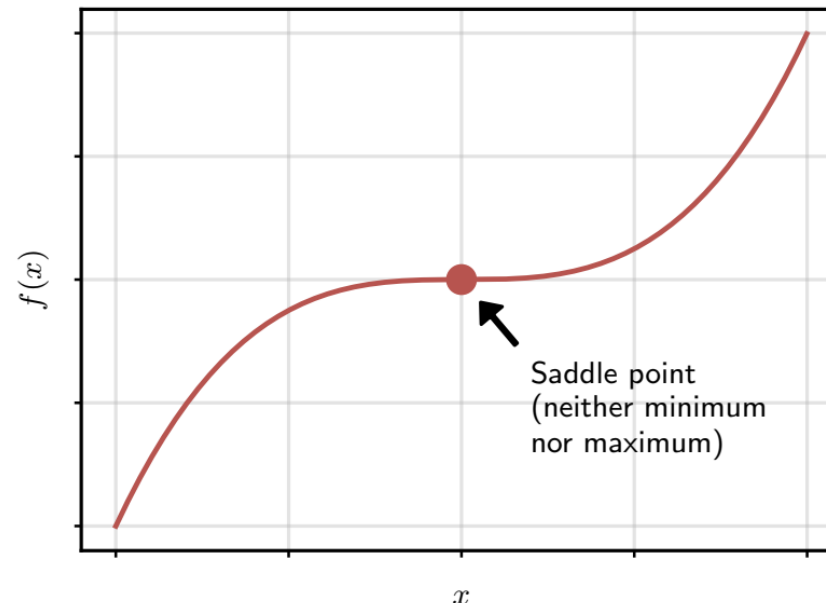
In this point, the slope of the tangent is zero.

Gradient Descent

Convergence of gradient descent??

Definition D.2.10 (Stationary points) A stationary point of $f(\mathbf{x})$ is a point \mathbf{x}^+ such that $\nabla f(\mathbf{x}^+) = 0$. titik stasioner dimana gradiennya itu 0

Stationary points are not limited to minima: they can be maxima (the minima of $-f(\mathbf{x})$) or **saddle points**.



Saddle poin itu, gak mesti minima/minimum.

Gradien descent kebanyakan kasus brhenti di saddle point.

Gradient Descent

Convergence of gradient descent??

Definition D.2.11 (Global minimum) A *global minimum* of $f(\mathbf{x})$ is a point \mathbf{x}^* such that $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for any possible input \mathbf{x} .

Definition D.2.12 (Convex function) A function $f(\mathbf{x})$ is *convex* if for any two points \mathbf{x}_1 and \mathbf{x}_2 and $\alpha \in [0, 1]$ we have:

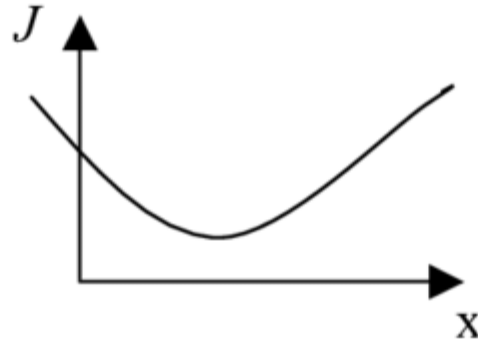
$$f(\alpha \mathbf{x}_1 + (1 - \alpha) \mathbf{x}_2) \leq \alpha f(\mathbf{x}_1) + (1 - \alpha) f(\mathbf{x}_2)$$

Interval from \mathbf{x}_1 to \mathbf{x}_2

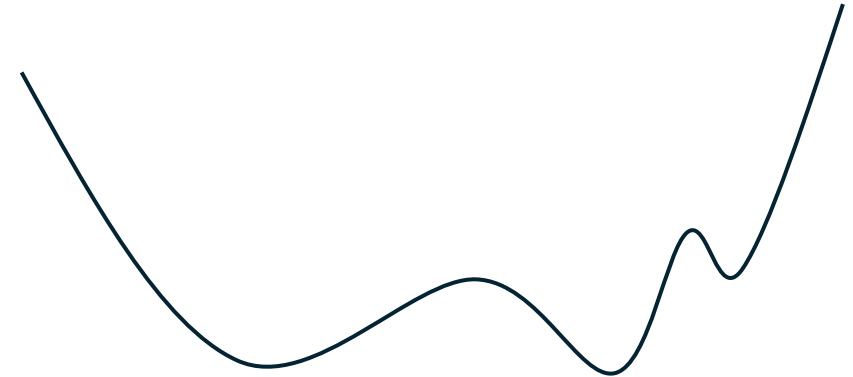
Line segment from $f(\mathbf{x}_1)$ to $f(\mathbf{x}_2)$

Gradient Descent

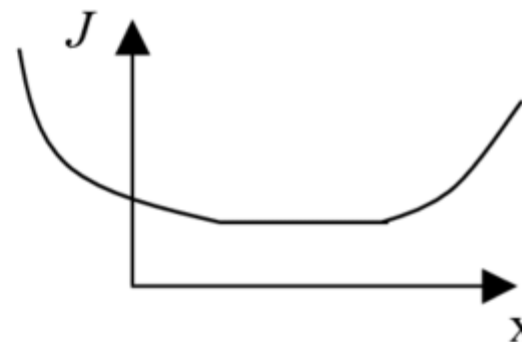
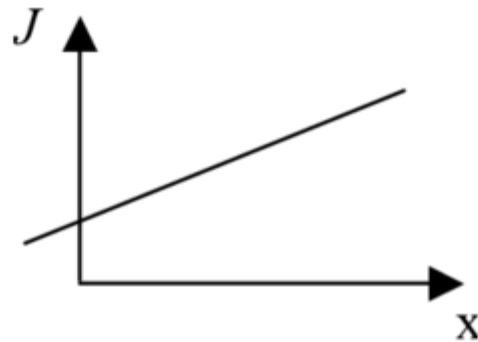
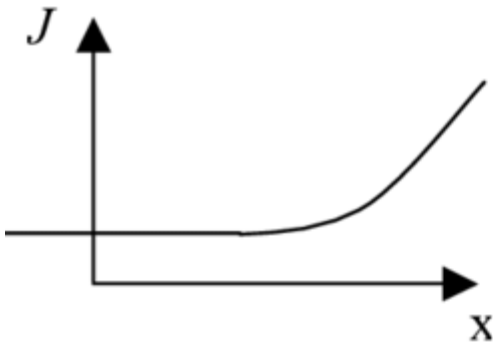
Convergence of gradient descent??



a) convex (and strictly convex)



Non-convex



b) convex (but not strictly convex)

Gradient Descent

Convergence of gradient descent??

1. For a generic **non-convex function**, gradient descent converges to a **stationary point**;
2. For a **convex function**, gradient descent will converge to a **global minimum**, irrespective of initialization;
3. If a convex function is also **strict**, the global minimizer will also be **unique**. kayak kertas dilipet.

Accelerating Gradient Descent --> Momentum

When dealing with **large models**, steepest descent can be extremely noisy (especially when using **stochastic optimization**).

kalau kayak ada beberapa miliar baris

Kita harus pisah baris baris tersebut menjadi beberapa batch (stochastic)

A variety of techniques have been developed to accelerate convergence of the optimization algorithm by selecting better descent directions.

nah buat bikin convergence nya itu, kita mau lebih speech, kita bisa pakai momentum (heavy ball).

For computational reasons, we are especially interested in methods that do not require **higher-order derivatives (e.g., the Hessian)**, or multiple calls to the function.

Accelerating Gradient Descent --> Momentum

Momentum is our choice.

If you picture **gradient descent** as a ball “**rolling down a hill**”, the movement is relatively erratic, because each gradient can point in a completely different direction.

We can **smooth this behavior** by introducing a “**momentum**” term that conserves some direction from the previous gradient iteration:

The diagram illustrates the momentum gradient descent equation:
$$\mathbf{g}_t = -\eta_t \nabla f(\mathbf{x}_{t-1}) + \lambda \mathbf{g}_{t-1}$$
$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{g}_t$$
Annotations include:

- A red arrow labeled "Normal gradient descent iteration" points to the first term $-\eta_t \nabla f(\mathbf{x}_{t-1})$.
- A blue arrow labeled "Additional momentum term" points to the second term $\lambda \mathbf{g}_{t-1}$.
- An orange box contains the initial condition $\mathbf{g}_0 = 0$.
- A light blue box contains the text: "The coefficient λ determines how much the previous term is dampened."

Accelerating Gradient Descent --> Momentum

Normal gradient descent iteration

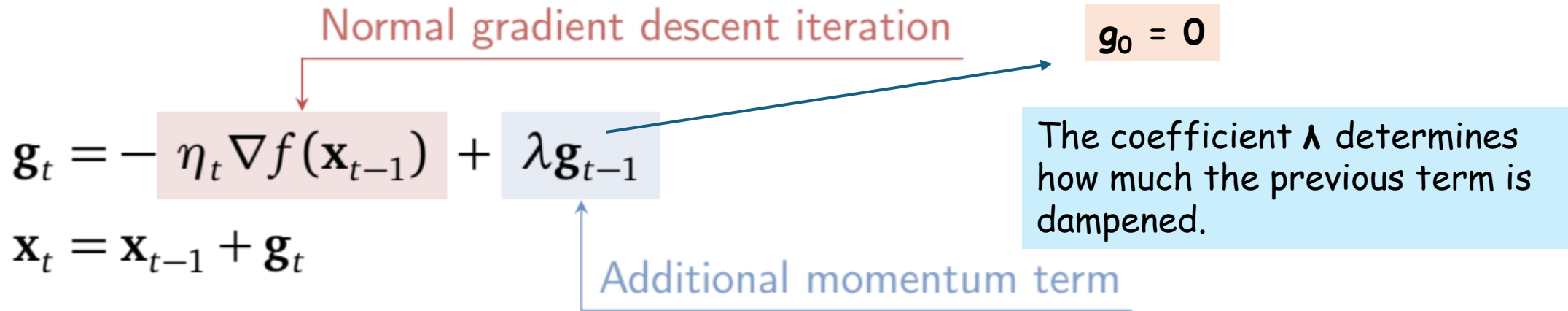
$$\mathbf{g}_t = -\eta_t \nabla f(\mathbf{x}_{t-1}) + \lambda \mathbf{g}_{t-1}$$

$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{g}_t$

$\mathbf{g}_0 = 0$

The coefficient λ determines how much the previous term is dampened.

Additional momentum term



If we unroll:

$$\begin{aligned}\mathbf{g}_t &= -\eta_t \nabla f(\mathbf{x}_{t-1}) + \lambda(-\eta_t \nabla f(\mathbf{x}_{t-2}) + \lambda \mathbf{g}_{t-2}) \\ &= -\eta_t \nabla f(\mathbf{x}_{t-1}) - \lambda \eta_t \nabla f(\mathbf{x}_{t-2}) + \lambda^2 \mathbf{g}_{t-2}\end{aligned}$$

kalau pakai momentum gradien sebelumnya perlu disimpan.

One disadvantage of using accelerated optimization algorithms can be increased storage requirements: for example, momentum requires us to **store the previous gradient iteration in memory, doubling the space** needed by the optimization algorithm.

PRACTICE: Jika Anda mengerjakan ini, Anda mendapat 500 Point

To become proficient with all three frameworks (NumPy, JAX, PyTorch), I suggest to replicate the exercise below thrice – each variant should only take a few minutes if you know the syntax. Consider a 2D function $f(\mathbf{x})$, $\mathbf{x} \sim (2)$, where we take the domain to be $[0, 10]$:¹²

$$f(\mathbf{x}) = \sin(x_1) \cos(x_2) + \sin(0.5x_1) \cos(0.5x_2)$$

Before proceeding in the book, repeat this for each framework:

1. Implement the function in a **vectorized** way, i.e., given a matrix $\mathbf{X} \sim (n, 2)$ of n inputs, it should return a vector $f(\mathbf{X}) \sim (n)$ where $[f(\mathbf{X})]_i = f(\mathbf{X}_i)$.
2. Implement another function to compute its gradient (hard-coded – we have not touched automatic differentiation yet).
3. Write a basic gradient descent procedure and visualize the paths taken by the optimization process from multiple starting points.
4. Try adding a momentum term and visualizing the norm of the gradients, which should converge to zero as the algorithm moves towards a stationary point.

Datasets & Losses

What is Supervised Datasets?

We consider a scenario in which manually coding a certain function is unfeasible (e.g., recognizing objects from real-world images), but gathering **examples** of the desired behaviour is sufficiently easy.

Sebuah data yang mempunyai n instances/datapoints

Definition D.3.1 (Dataset) A supervised dataset \mathcal{S}_n of size n is a set of n pairs $\mathcal{S}_n = \{(x_i, y_i)\}_{i=1}^n$, where each (x_i, y_i) is an example of an input-output relationship we want to model. We further assume that each example is an **identically and independently distributed (i.i.d.)** draw from some unknown (and unknowable) probability distribution $p(x, y)$. Harus identically and idendently distributed

Identically distributed → stable & unchanging through time.

Independently distributed → has no bias in its collection; sufficiently representative of the entire distribution.

What is Supervised Datasets?

harus dilatih terus menerus.

More on the i.i.d. property

Importantly, ensuring the i.i.d. property is not a one-shot process, and it must be checked constantly during the lifetime of a model. In the case of car classification, if unchecked, subtle changes in the distribution of cars over time will degrade the performance of a machine learning model, an example of **domain shift**. As another example, a recommender system will change the way users interact with a certain app, as they will start reacting to suggestions of the recommender system itself. This creates **feedback loops** [CMMB22] that require constant re-evaluation of the performance of the system and of the app.

Variants of Supervised Learning - Pre-trained Models

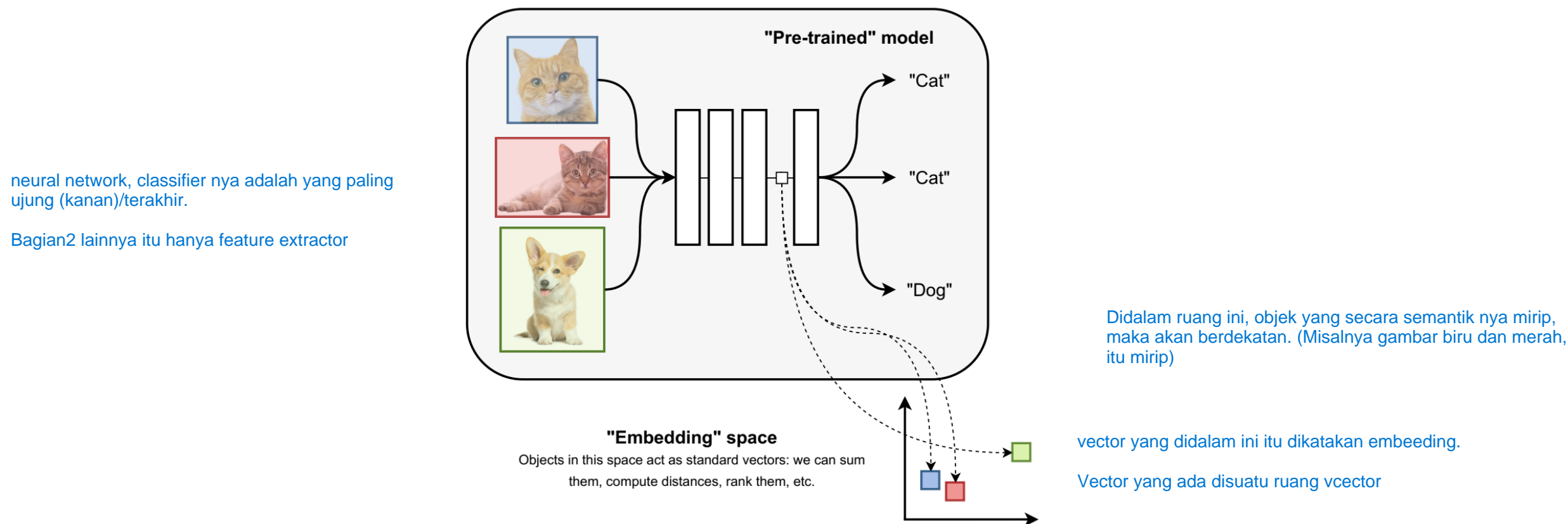


Figure F.3.1: Differentiable models process data by transforming it sequentially via linear algebra operations. In many cases, after we optimize these programs, the internal representations of the input data of the model (what we call a **pre-trained** model) have geometric properties: for example, semantically similar images are projected to points that are close in this "latent" space. Transforming data from a non-metric space (original input images) to a metric space (bottom right) is called **embedding** the data.

Variants of Supervised Learning - Pre-trained Models

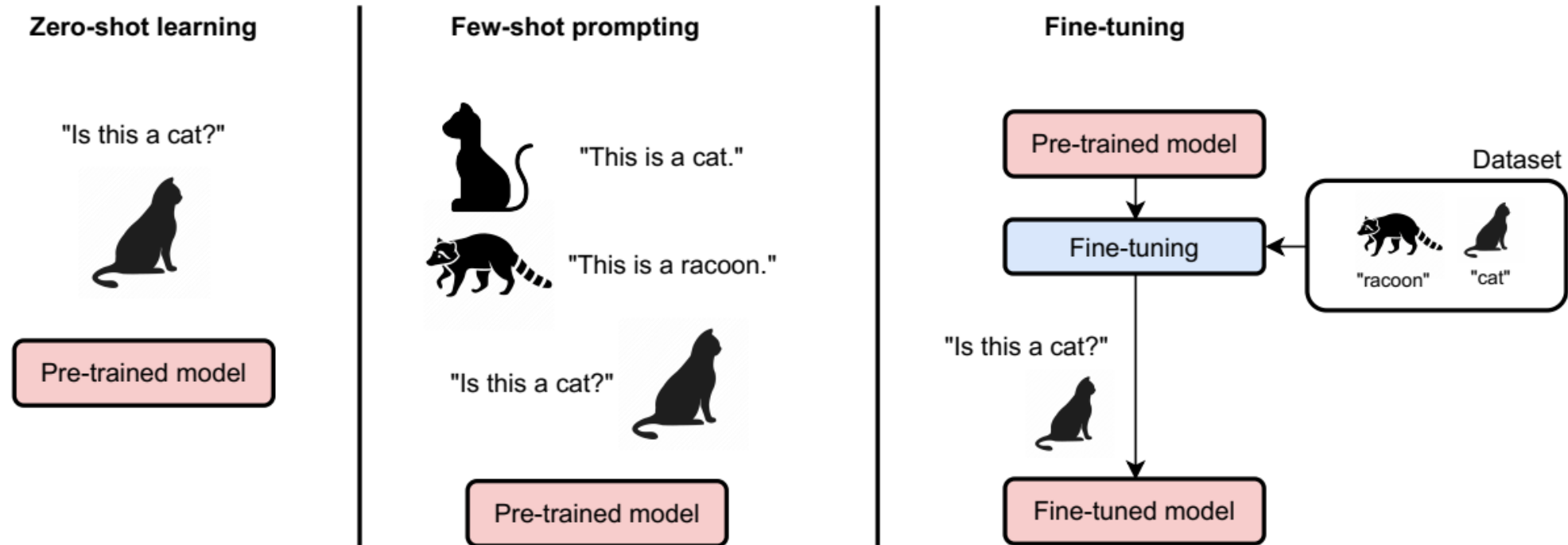


Figure F.3.2: Three ways of using trained models. **Zero-shot:** a question is directly given to the model. This can be achieved with generative language models (introduced in Chapter 8). **Few-shot prompting** is similar, but a few examples are provided as input. Both techniques can be employed only if the underlying model shows a large amount of generalization capabilities. **Fine-tuning:** the model is optimized via gradient descent on a small dataset of examples. This proceeds similarly to training the model from scratch.

Loss Functions

Once data has been gathered, we need to formalize our idea of “**approximating**” the desired behavior, which we do by introducing the concept of **loss functions**.

Seberapa dekat y dengan y topi. Kita mau dapetin yang loss function lebih kecil. Ada info dibawah.

Definition D.3.2 (Loss function) *Given a desired target y and the predicted value $\hat{y} = f(x)$ from a model f , a **loss function** $l(y, \hat{y}) \in \mathbb{R}$ is a scalar, differentiable function whose value correlates with the performance of the model, i.e., $l(y, \hat{y}_1) < l(y, \hat{y}_2)$ means that the prediction \hat{y}_1 is better than the prediction \hat{y}_2 when considering the reference value (target) y .*

1. buat function harus differentiable
- 2.

Loss Functions

n buah data point (dimana setaip data point itu x, y)

To this end, given a dataset $\mathcal{S}_n = \{(x_i, y_i)\}$ and a loss function $l(\cdot, \cdot)$, a sensible optimization task to solve is the minimum average loss on the dataset achievable by any possible *differentiable* model f :

$$f^* = \arg \min_f \frac{1}{n} \sum_{i=1}^n l(y_i, f(x_i)) \quad (\text{E.3.1})$$

Average over the dataset

menghitung loss untuk setiap data point

Prediction of f on the i -th sample from the dataset

ini untuk mendapatkan rata-rata loss setiap data point

Gradien descent yang kita minimalkan itu loss function (l), bukan F nya. Didalam L ada f.

For historical reasons, (E.3.1) is referred to as **empirical risk minimization (ERM)**, where risk is used as a generic synonym for loss.

Loss Functions

To this end, given a dataset $\mathcal{S}_n = \{(x_i, y_i)\}$ and a loss function $l(\cdot, \cdot)$, a sensible optimization task to solve is the minimum average loss on the dataset achievable by any possible *differentiable* model f :

$$\mathbf{w}^* = \arg \min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^n l(y_i, f(x_i, \mathbf{w}))$$

It's almost impossible to minimize across the space of all possible functions. In practical, $f(\mathbf{x})$ is fixed, but with **a set of parameters**.

Minimization is done by searching for the optimal value of these parameters via numerical optimization, which we denote by **$f(\mathbf{x}, \mathbf{w})$** .

Examples of Loss Functions

Consider a **binary classification** task where $y \in \{-1, +1\}$. We define a **0/1 loss function**:

$$l(y, \hat{y}) = \begin{cases} 0 & \text{if } \text{sign}(\hat{y}) = y \\ 1 & \text{otherwise} \end{cases}$$

0/1 loss function adalah loss function yang masuk akal (kalau nilainya sama, maka 1, kalau beda = 0)

y topi itu prediction.

While this aligns with our intuitive notion of “**being right**”, it is **useless** as loss function since its gradient will almost always be zero.

gak berguna karena gak bisa diturunin differentiable function kita.

A possible loss function in this case is the **hinge loss**:

$$l(y, \hat{y}) = \max(0, 1 - y \hat{y})$$

y topi itu row output dari classifier

kalau sign itu prediction

Linear Models

Least-squares Regression

A supervised learning problem can be defined by choosing the **input type x** , the **output type y** , the **model f** , and the **loss function l** .

Now, we consider a simple case:

- The input is a vector **$x \sim (c)$** , corresponding to a set of features;

sepanjang c (1 vector, ada 1 baris kolomnya sebanyak c)

- The output is a single real value **$y \in \mathbb{R}$ (regression task)**;

hasilnya \mathbb{R} (float)

- We take **f** to be a linear model, providing us with simple closed-form solutions.

Least-squares Regression

Regression losses: squared loss and variants

We do not care about the sign of the prediction error, a common choice is the **squared loss**:

$$l(\hat{y}, y) = (\hat{y} - y)^2$$

penaltinya gede

Here and in the following we use the symbol \hat{y} to denote the **prediction of a generic model**.

Drawback: higher errors will be penalized with a strength that grows quadratically in the error, which may **provide undue influence to outliers**.

Least-squares Regression

Regression losses: squared loss and variants

Other choices that diminish the influence of outliers can be the **absolute value loss**:

$$l(\hat{y}, y) = |\hat{y} - y|$$

gak differentiable

limit kanan sama limit kirinya beda kalo diturunin.

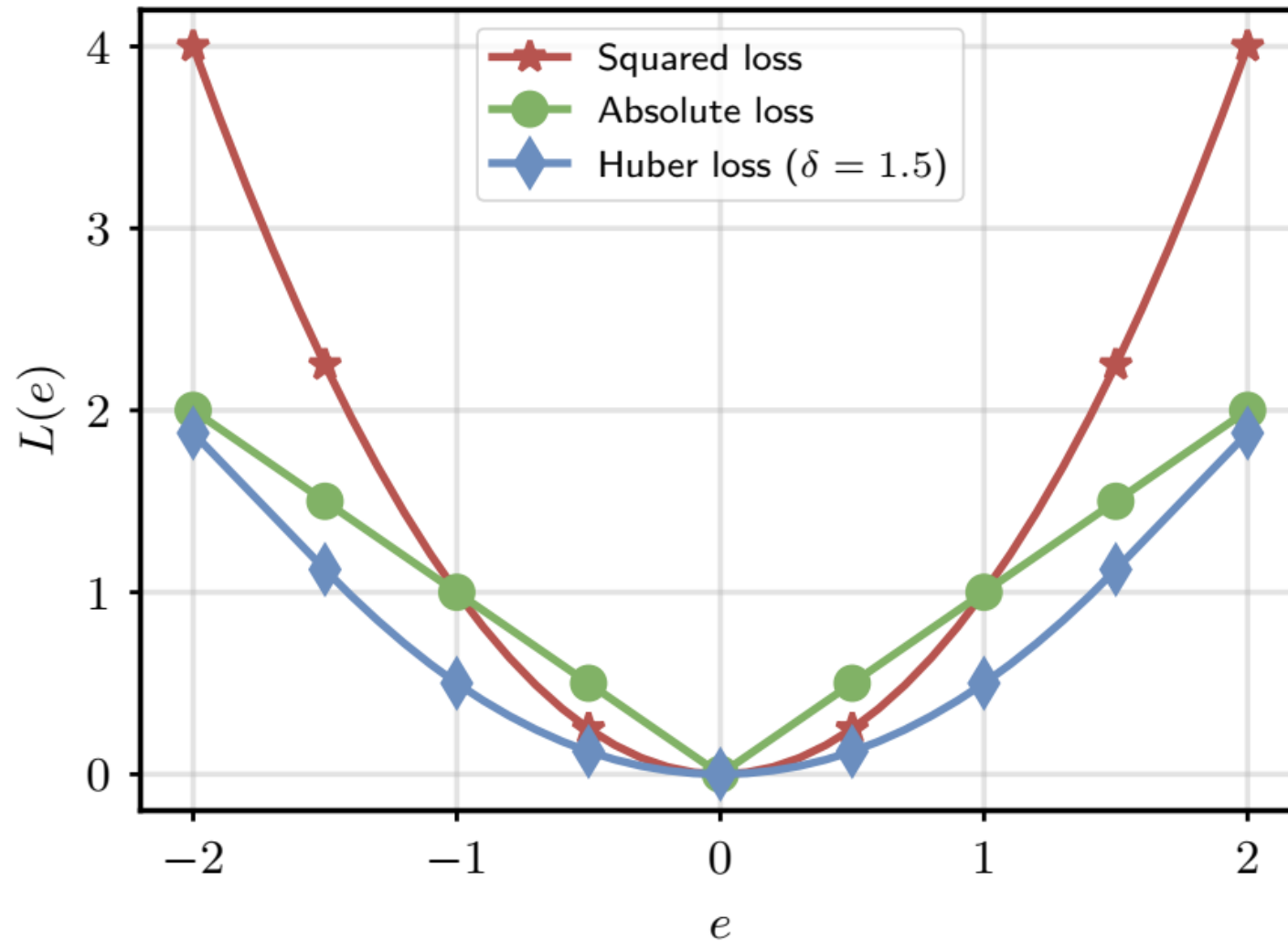
or the **Huber loss** (a combination of the squared loss and the absolute loss):

$$L(y, \hat{y}) = \begin{cases} \frac{1}{2} (y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq 1 \\ (|y - \hat{y}| - \frac{1}{2}) & \text{otherwise} \end{cases}$$

quadratic in the promixity of 0 error, and linear otherwise (with the -1/2 term added to ensure continuity)

Least-squares Regression

Regression losses: squared loss and variants



Least-squares Regression

With a loss function in hand, we consider the following model (**a linear model**) to complete the specification of our first supervised learning problem.

Definition D.4.1 (Linear models) *A linear model on an input \mathbf{x} is defined as:*

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

dot product

where $\mathbf{w} \sim (c)$ and $b \in \mathbb{R}$ (the bias) are trainable parameters.

Least-squares Regression

Combining the **linear model**, the **squared loss**, and an **empirical risk minimization** problem we obtain the **least-squares optimization problem**.

Definition D.4.2 (Least-squares) *The least-squares optimization problem is given by:*

$$\mathbf{w}^*, b^* = \arg \min_{\mathbf{w}, b} \frac{1}{n} \sum_{i=1}^n \overbrace{(y_i - \mathbf{w}^\top \mathbf{x}_i - b)}^{\text{true - prediction}}^2 \quad (\text{E.4.4})$$

y sebanyak data point (sebanyak n)

kalau w sebanyak fitur ()

How would you compute E.4.4? Do you use a “loop” that iterates all instances in the training data one-by-one?

selalu pake vectorized
approach karena ada GPU
sekarang

Least-squares Regression

For the sake of efficiency, we rewrite the least-squares in a **vectorized form** that only involves matrix operations.

Modern code for training differentiable models is built around n -dimensional arrays, with **optimized hardware to perform matrix operations** on them.

Input Matrix:

X^T itu panjangnya $x \sim (c)$

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix} \sim (n, c)$$

Size of dataset
(num instances)

Num features

Output Vector:

$$\mathbf{y} = [y_1, \dots, y_n]^T$$

Batched Model Output:

$$f(\mathbf{X}) = \mathbf{X}\mathbf{w} + \mathbf{1}b$$

x (ini x kecil) itu awal2

$[x_1]$
 $[x_2]$
 $[x_3]$
 $[x_4]$

kalo di transpose jadi $[x_1 \ x_2 \ x_3 \ x_4]$

Same bias b for all n predictions

Least-squares Regression

Definition D.4.2 (Least-squares) *The least-squares optimization problem is given by:*

$$\mathbf{w}^*, b^* = \arg \min_{\mathbf{w}, b} \frac{1}{n} \sum_{i=1}^n (y_i - \mathbf{w}^\top \mathbf{x}_i - b)^2 \quad (\text{E.4.4})$$

The **vectorized** least-squares optimization problem



$$\text{LS}(\mathbf{w}, b) = \frac{1}{n} \|\mathbf{y} - \mathbf{X}\mathbf{w} - \mathbf{1}b\|^2$$

$$\|\mathbf{e}\|^2 = \sum_i e_i^2$$

Least-squares Regression: How to solve?

Suppose we ignore the bias (bias can be easily incorporated into \mathbf{w}):

$$LS(\mathbf{w}) = \frac{1}{n} \|\mathbf{y} - X\mathbf{w}\|^2$$

The gradient w.r.t \mathbf{w} is:

$$\nabla LS(\mathbf{w}) = X^T (X\mathbf{w} - \mathbf{y})$$

Can you show this?

Least-squares Regression: How to solve?

We can actually find optimal \mathbf{w} using a closed form formula:

$$\mathbf{w}_* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

Can you show this?

Performing the inversion in $(\mathbf{X}^T \mathbf{X})^{-1}$ is not always possible. for example, if **one feature is a scalar multiple of the other**, the matrix \mathbf{X} does not have **full rank** (this is called **collinearity**).

Least-squares Regression: How to solve?

But, we are more interested in Gradient Descent method:

```
def least_squares_gd(X: Float[Tensor, "n c"],
                    y: Float[Tensor, "n"],
                    learning_rate=1e-3) \
    -> Float[Tensor, "c"]:
    # Initializing the parameters
    w = torch.randn((X.shape[1], 1))

    # Fixed number of iterations
    for i in range(15000):
        # Note the sign: the derivative has a minus!
        w = w + learning_rate * X.T @ (y - X @ w)

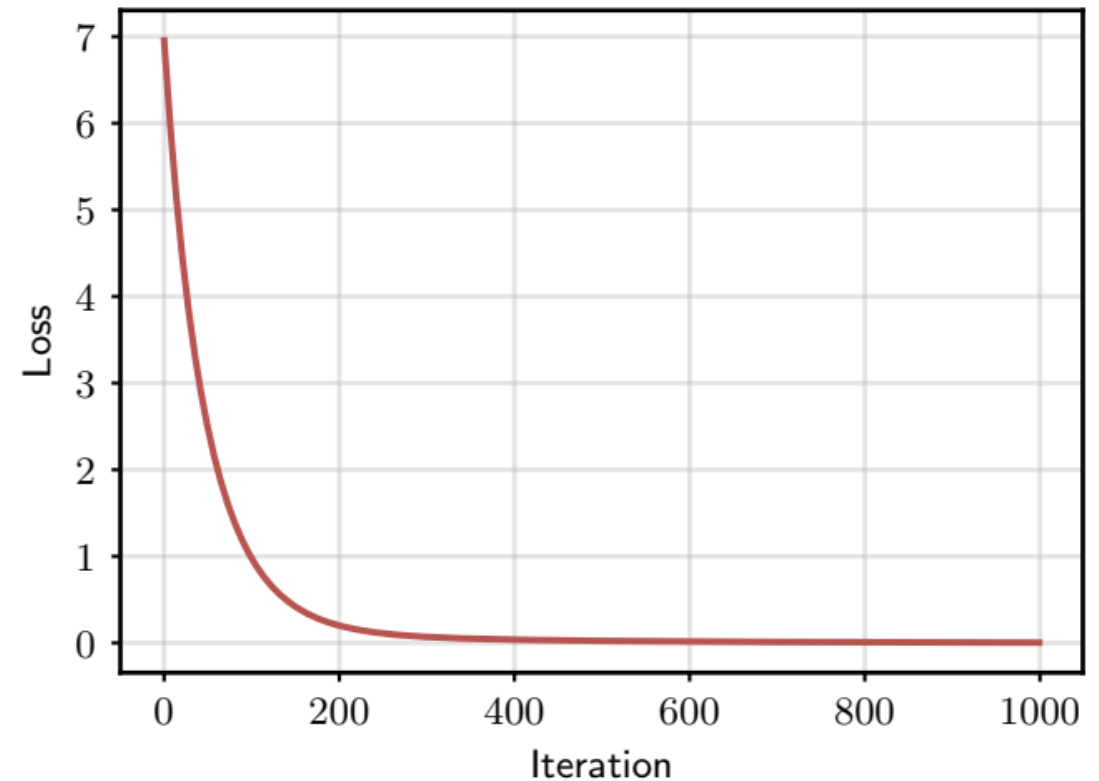
    return w
```

Fixed learning rate = 0.001

Least-squares Regression: How to solve?

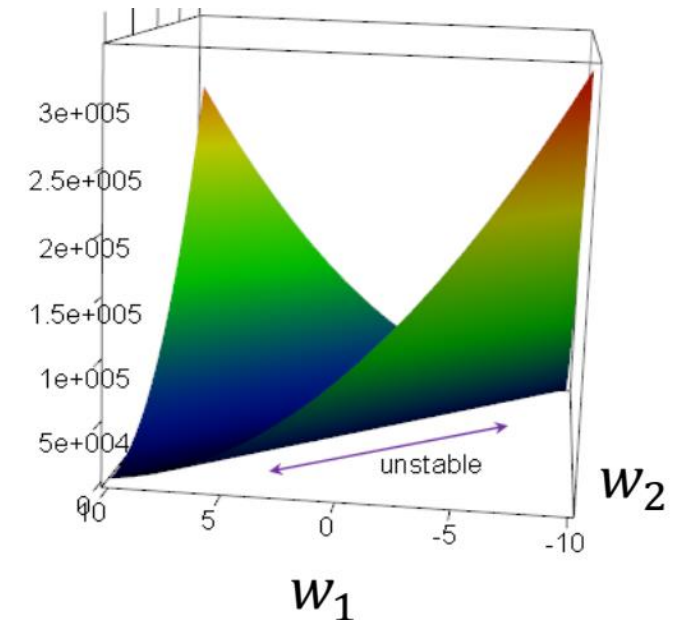
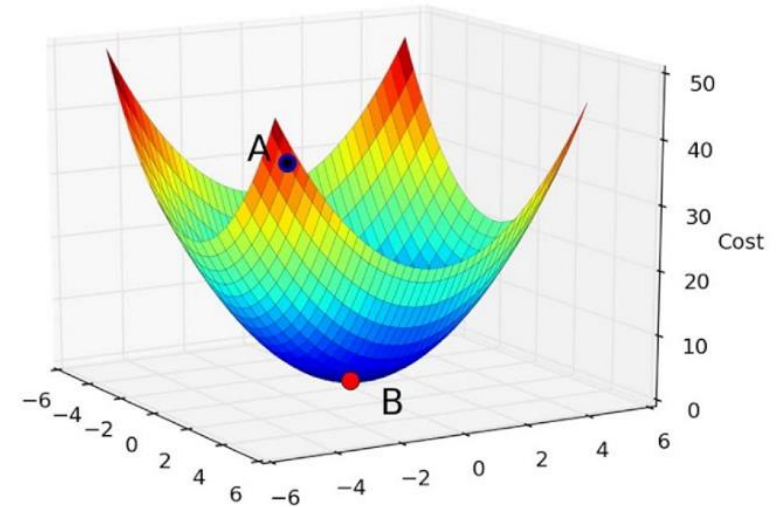
But, we are more interested in Gradient Descent method:

Figure E.4.2: *An example of running code from Box C.4.2, where the data is composed of $n = 10$ points drawn from a linear model $\mathbf{w}^\top \mathbf{x} + \varepsilon$, with $w_i \sim \mathcal{N}(0, 1)$ and $\varepsilon \sim \mathcal{N}(0, 0.01)$. Details apart, note the very smooth descent: each step provides a decrease in loss.*



Least-squares Regression: How to solve?

- **Good News:** jika tidak ada fitur-fitur yang irrelevant, loss function-nya bersifat “strictly convex” (seperti mangkuk).
- **Bad News:** jika ada fitur-fitur yang irrelevant, permasalahan bersifat “convex but not strictly” (seperti *ridge*).



solusi: regularization

Least-squares Regression: How to solve? --- irrelevant features!

Contoh: model yang diterapkan pada data dengan 3 buah fitur, namun fitur pertama dan kedua sama.

x1 adalah irrelevant feature
(atau x2)

$$f(\mathbf{x}) = w_1x_1 + w_2x_2 + w_3x_3 + b$$

x1	x2	x3
8	8	98
43	43	21
12	12	16
87	87	44

Misalkan, solusi-nya adalah: $[\hat{w}_1, \hat{w}_2, \hat{w}_3, \hat{b}]$

Namun sayangnya, karena **x1** dan **x2** sama, solusi di atas **tidak unik**.

contoh: $[\hat{w}_1 + 0.5, \hat{w}_2 - 0.5, \hat{w}_3, \hat{b}]$

$[\hat{w}_1 - 0.1, \hat{w}_2 + 0.1, \hat{w}_3, \hat{b}]$

Dua solusi ini akan
memberikan nilai
 $f(\mathbf{x})$ yang sama

Least-squares Regression: How to solve? --- irrelevant features!

Contoh: model yang diterapkan pada data dengan 3 buah fitur, namun fitur pertama dan kedua sama.

x1 adalah

x1		
8		
43		
12		
87	87	44

Solusi yang tidak unik:

- *lack of interpretability*. Sebenarnya, nilai parameter itu bisa digunakan untuk mengetahui **effect size** sebuah fitur. Namun, karena nilainya tidak unik, hal ini jadi tidak berguna.
- Proses optimization menjadi **ill-posed problem**.

$$\left[\hat{w}_1 - 0.1, \hat{w}_2 + 0.1, \hat{w}_3, \hat{b} \right]$$

di

ini akan
an nilai
g sama

Irrelevant Features: Multi-Collinearity

Sebelumnya adalah **kasus ekstrim**. Secara umum, fitur-fitur yang irrelevant adalah fitur-fitur yang **collinear**.

Fitur x_j **irrelevant** jika x_j adalah **kombinasi linear** dari fitur lain.

$$x_j = \sum_{i \neq j} \alpha_i x_i$$

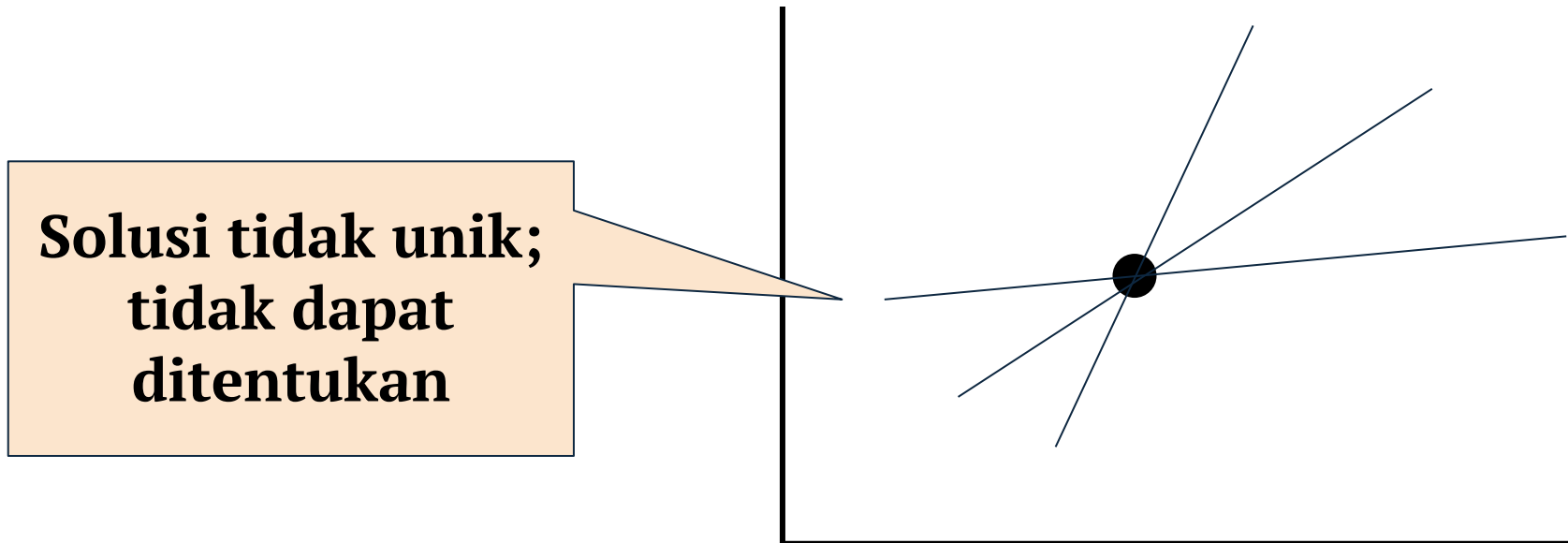
Hal-hal seperti ini bisa saja terjadi terutama ketika **fitur sangat banyak** (high-dimensional)

Ill-posed problem berikutnya: model terlalu kompleks

Model lebih kompleks dibandingkan data.

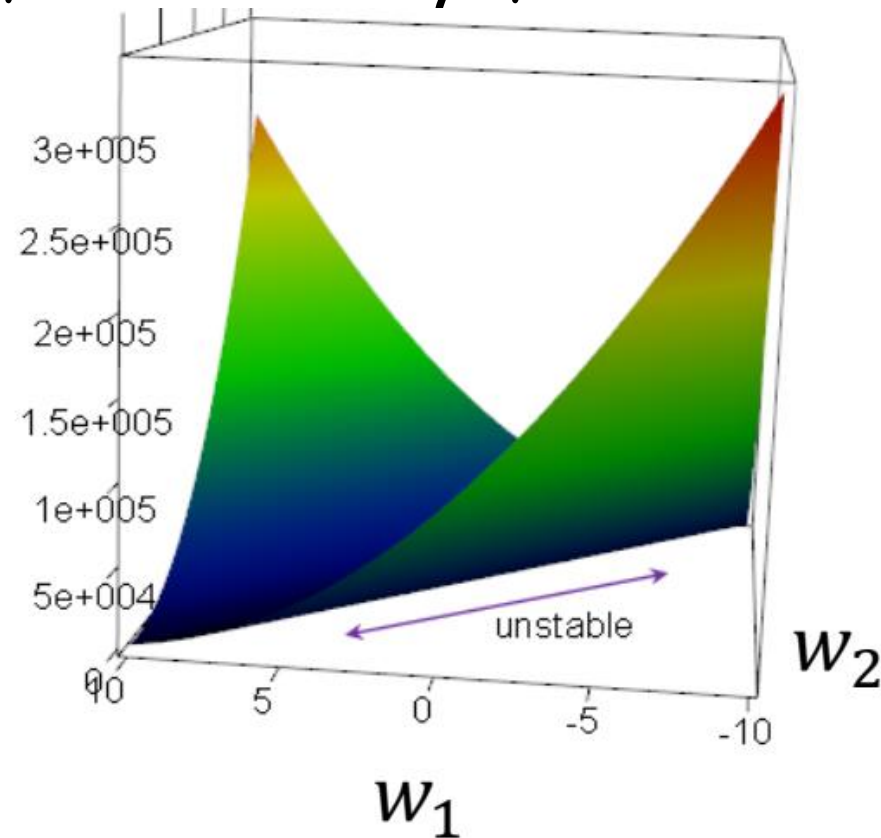
Kasus ekstrim:

- model terdiri dari 2 parameter: slope dan intercept
- hanya ada sebuah data point



Ill-posed problem

Artinya, solusi permasalahan (optimization) tidak dapat didefinisikan karena, salah satunya, **tidak unik**.



Least-squares Regression: How to solve? --- irrelevant features!

Solutions for irrelevant features?

It is possible to slightly modify the problem to achieve a solution which is "as close as possible" to the original one, while being feasible to compute.

For example, a known trick is to add a small multiple, $\lambda > 0$, of the identity matrix to the **matrix being inverted**:

$$\mathbf{w}_* = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y}$$

This **pushes the matrix to be "more diagonal"** and make the **"inversion easier"**.

Least-squares Regression: How to solve? --- irrelevant features!

Solutions for irrelevant features?

Backtracking to the original problem, we note this is the closed form solution of a modified optimization problem:

$$\mathbf{w}_* = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y}$$

This problem is called
regularized least-squares
(or **ridge regression**)



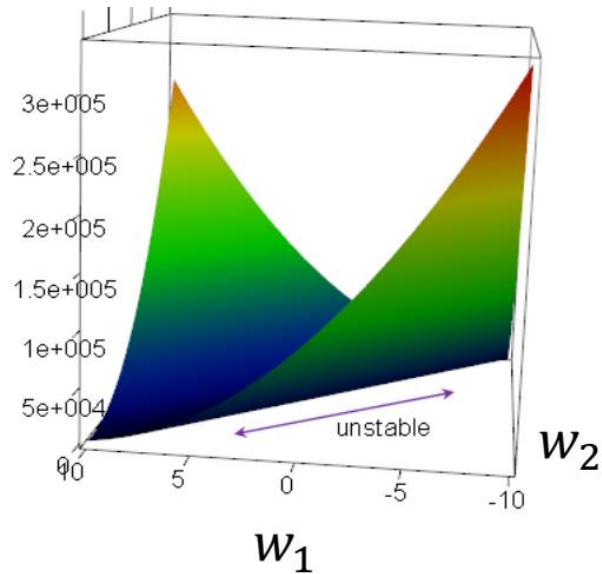
$$LS(\mathbf{w}) = \frac{2}{n} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

Regularization
hyper-parameter

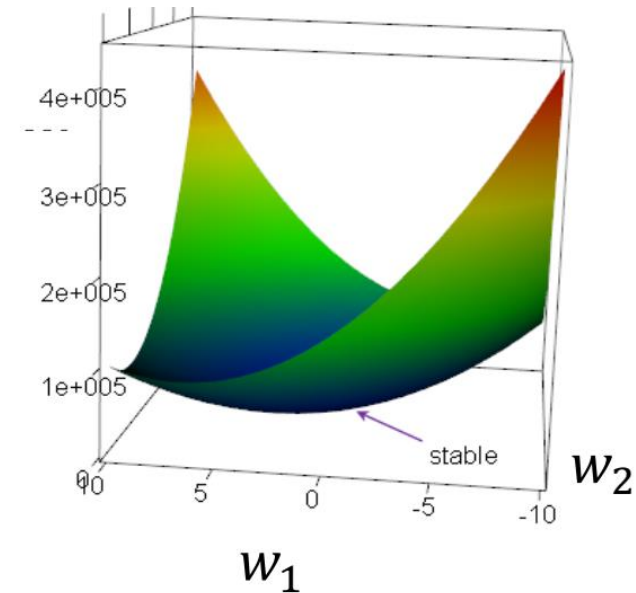
L2-Regularization

Least-squares Regression: How to solve? --- irrelevant features!

Solutions for irrelevant features?



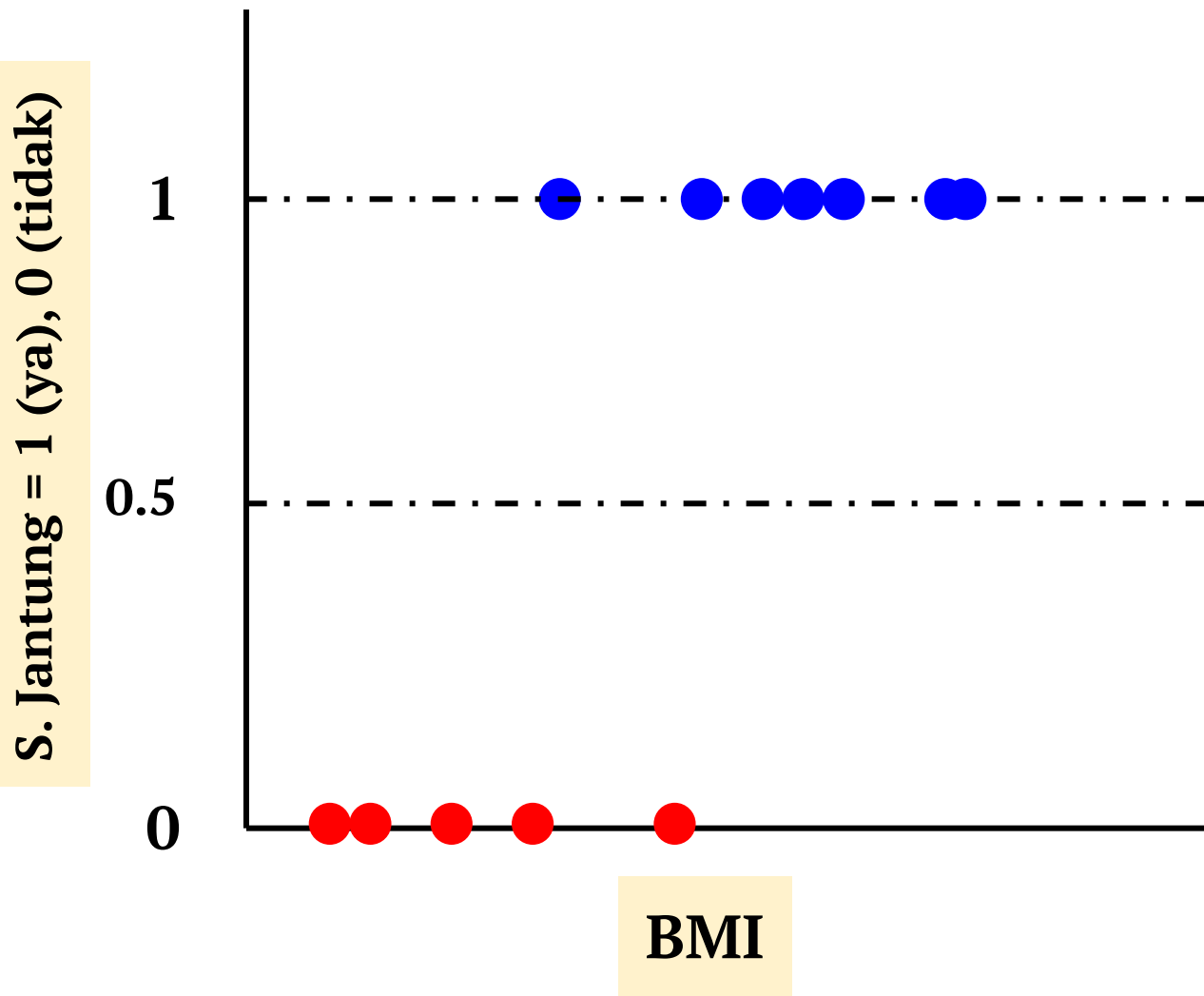
Sebelum diregularisasi



Setelah diregularisasi

Linear Models (Binary Classification)

Binary Classification

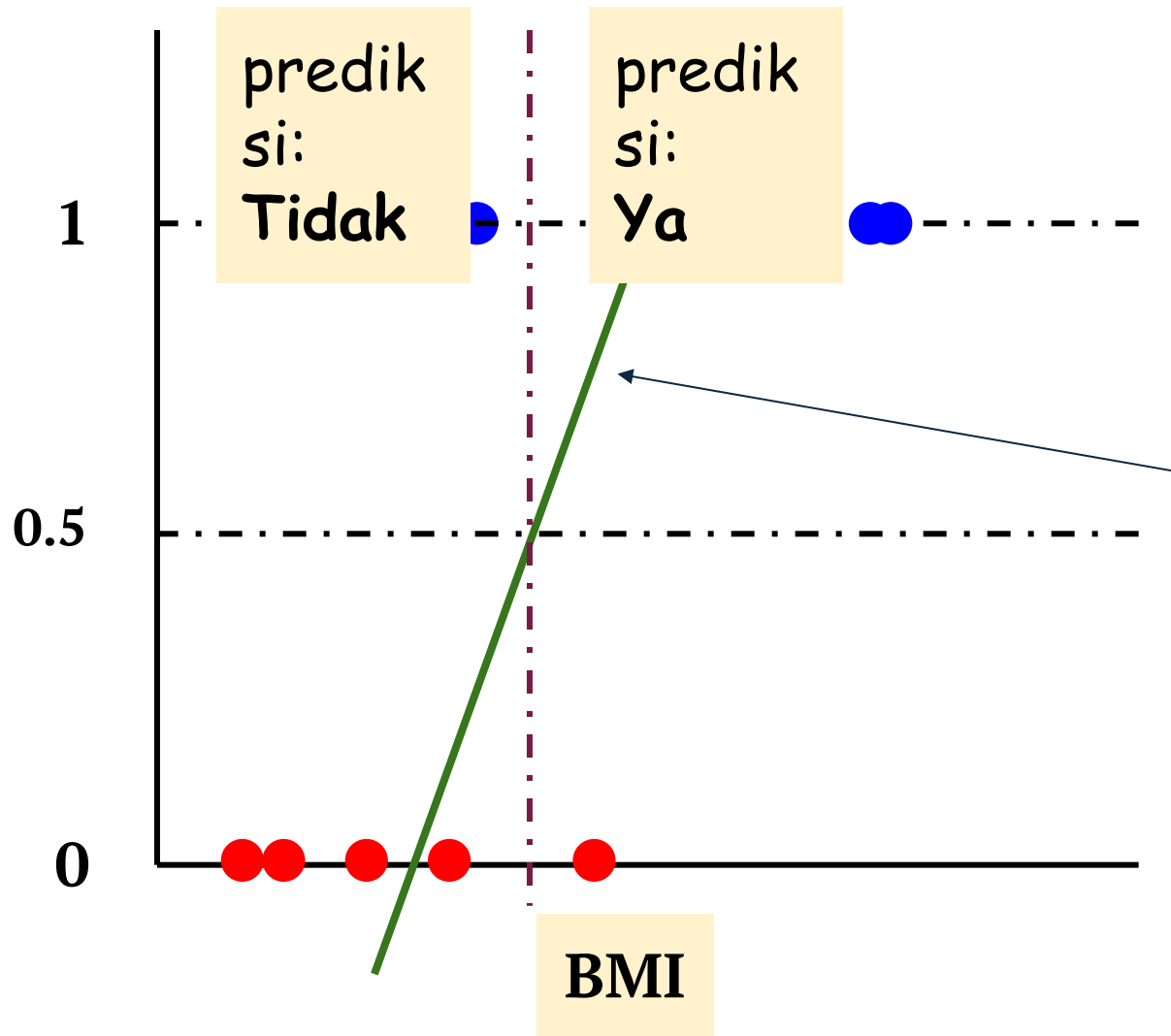


Diberikan informasi **BMI** (body mass index), coba prediksi apakah seseorang punya penyakit jantung atau tidak (**Ya**/**Tidak**)!

Bagaimana caranya membangun model klasifikasi untuk hal ini?

Binary Classification

S. Jantung = 1 (ya), 0 (tidak)

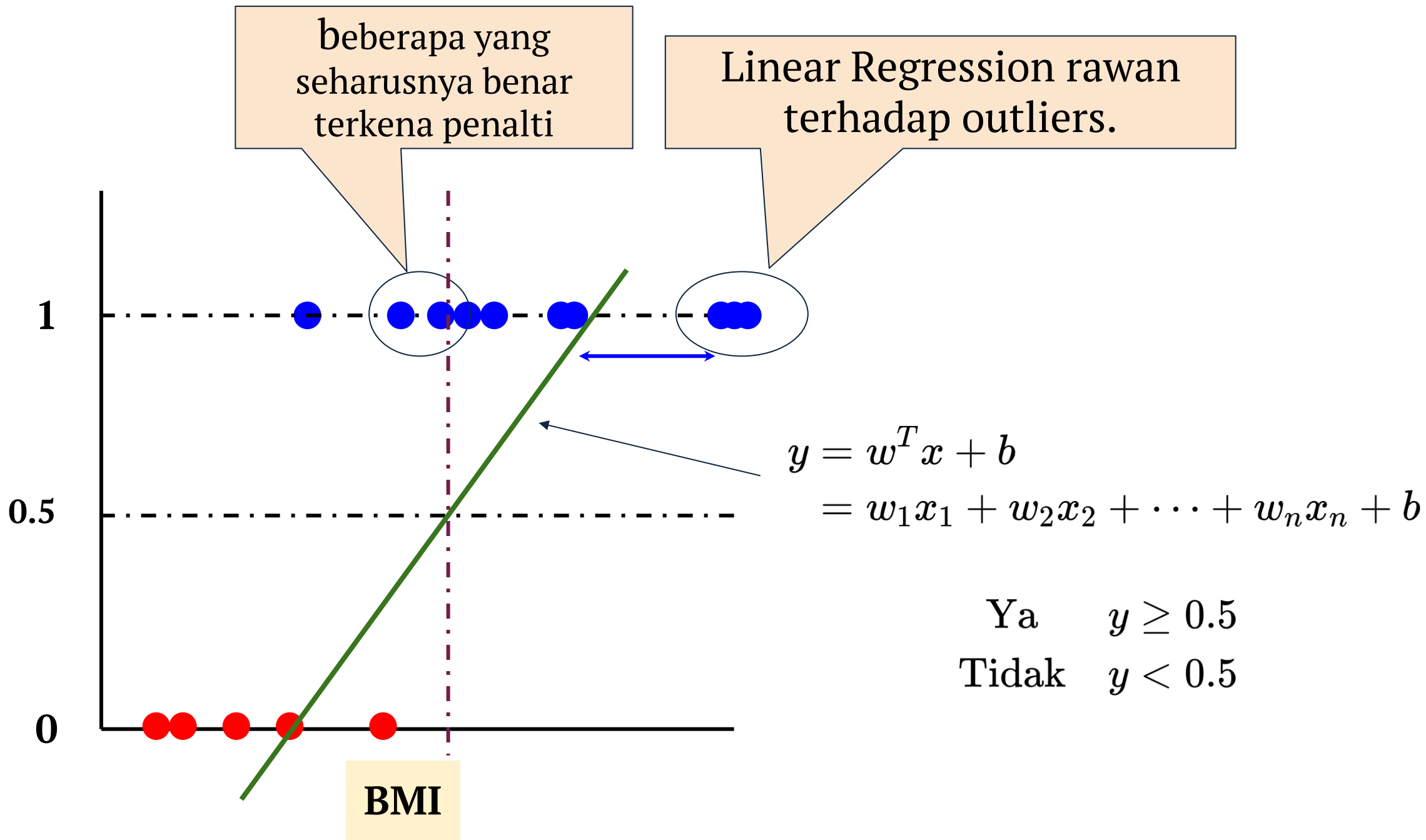


Memangnya **linear regression** biasa tidak bisa?

$$y = w^T x + b$$
$$= w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b$$

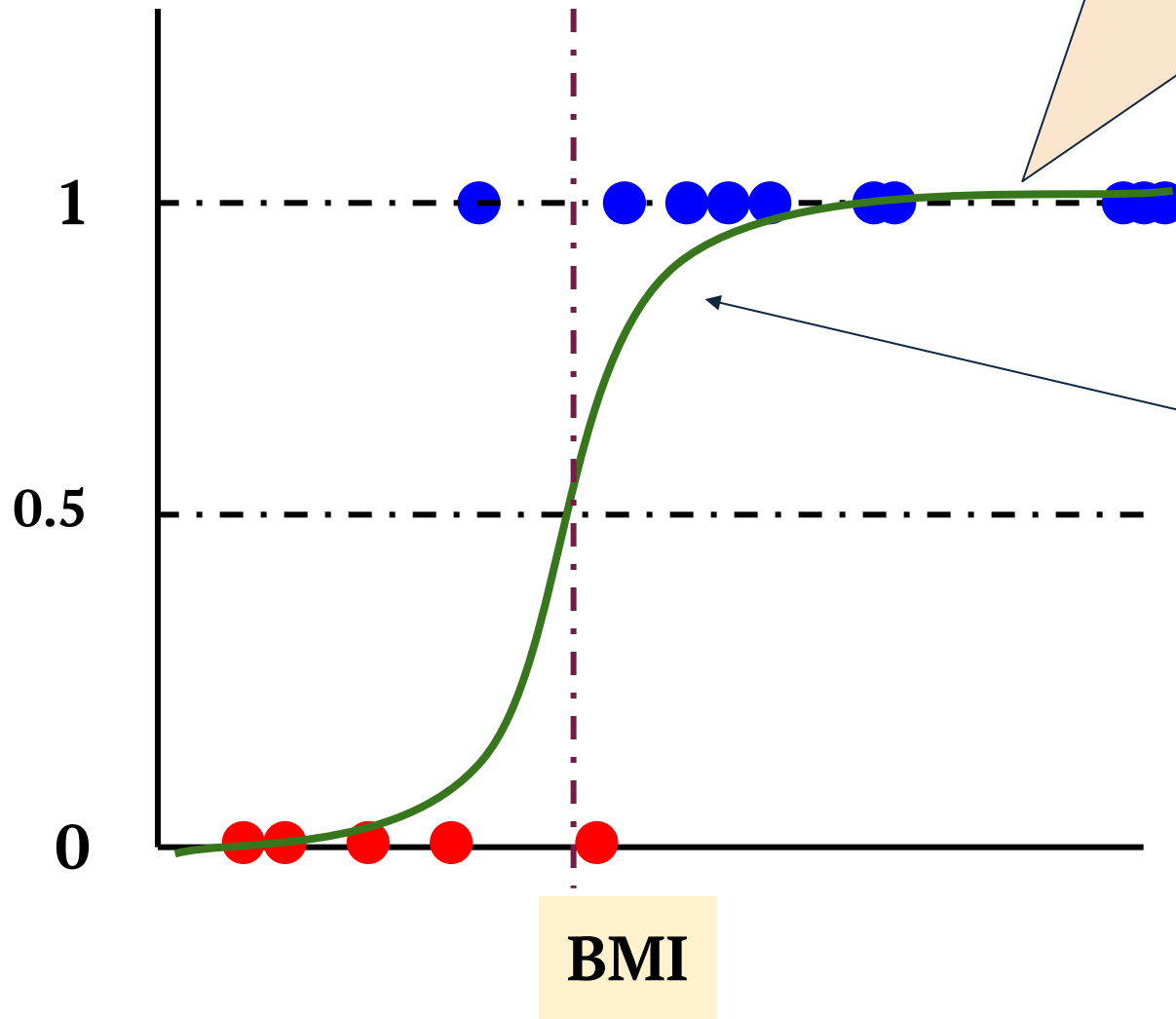
Ya $y \geq 0.5$
Tidak $y < 0.5$

S. Jantung = 1 (ya), 0 (tidak)



Binary Classification

S. Jantung = 1 (ya), 0 (tidak)



Akan lebih baik jika garis pemisah
melengkung seperti ini

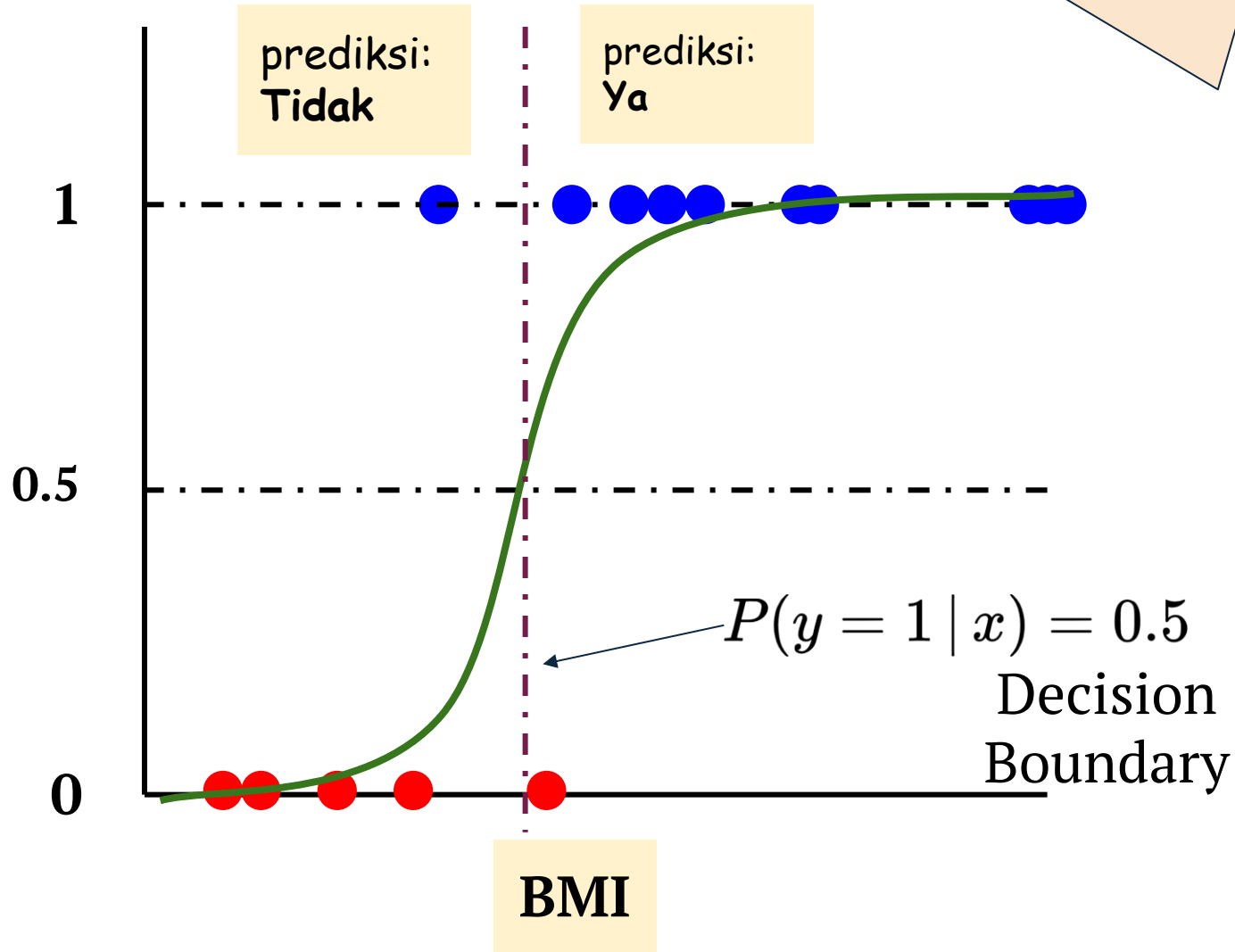
Ada fungsi yang sifatnya seperti ini, yaitu **logistic function**:

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

Binary Classification

Logistic Regression = Linear Regression + Logistic Function

S. Jantung = 1 (ya), 0 (tidak)



$$P(y = 1 | x) = \sigma(w^T x + b) = \frac{1}{1 + \exp(-w^T x - b)}$$

Disini, perlu diingat bahwa: $y \in \{0, 1\}$

Binary Classification

Loss Function: **Binary Cross Entropy** Function:

$$\text{CE}(\hat{y}, y) = \underbrace{-y \log(\hat{y})}_{\text{Loss for class 1}} \underbrace{-(1-y) \log(1-\hat{y})}_{\text{Loss for class 2}}$$

Consider the gradient of the binary logistic regression model with respect to \mathbf{w} :

$$\nabla \text{CE}(f(\mathbf{x}), y) = (f(\mathbf{x}) - y)\mathbf{x}$$

Note the similarity with the **gradient** of **a standard linear model for regression**.

Binary Classification

Note the similarity with the **gradient** of **a standard linear model for regression**.

$$\nabla \text{CE}(f(\mathbf{x}), y) = (f(\mathbf{x}) - y)\mathbf{x}$$

This similarity can be further understood by rewriting our model as:

The diagram illustrates the relationship between logits and log-odds. On the left, the expression $\mathbf{w}^\top \mathbf{x} + b$ is enclosed in a light red box. A red arrow labeled "Logits" points to this box. On the right, the expression $\log\left(\frac{y}{1-y}\right)$ is enclosed in a light blue box. A blue arrow labeled "Sigmoid inverse: $\sigma^{-1}(y)$ " points to this box. An equals sign is placed between the two boxes, indicating that the logit is equal to the log-odds.

$$\mathbf{w}^\top \mathbf{x} + b = \log\left(\frac{y}{1-y}\right)$$

This clarifies why we were referring to the model as a **"linear model" for classification**: we can always rewrite it as a **purely linear model in terms of a non-linear transformation** of the output (in this case, the inverse of the sigmoid, also known as the **log-odds**).

Materi Optional

Perhitungan gradien Binary-Cross Entropy loss function pada Logistic Regression

Logistic Regression Model:

Ingat bahwa x dan w adalah vektor!

$$\theta(x) = \frac{1}{1 + \exp(-z(x))}$$

$$z(x) = W^T x + b$$

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

Loss Function untuk training W dan b :

Banyaknya instances di training data

n = Banyaknya fitur

$$L(W, b) = \frac{1}{m} \sum_{i=1}^m -y_i \ln(\theta(x_i)) - (1 - y_i) \ln(1 - \theta(x_i))$$

Kali ini, kita coba gunakan \ln , bukan \log

Materi Optional

Perhitungan gradien Binary-Cross Entropy loss function pada Logistic Regression

Logistic Regression Model:

$$\theta(x) = \frac{1}{1 + \exp(-z(x))}$$

$$z(x) = W^T x + b$$

Goal: Cari W' dan b' yang meminimalkan L !

Loss Function untuk training W dan b :

$$W', b' = \underset{W, b}{\operatorname{argmin}} L(W, b)$$

$$L(W, b) = \frac{1}{m} \sum_{i=1}^m -y_i \ln(\theta(x_i)) - (1 - y_i) \ln(1 - \theta(x_i))$$

Materi Optional

Perhitungan gradien Binary-Cross Entropy loss function pada Logistic Regression

Logistic Regression Model:

$$\theta(x) = \frac{1}{1 + \exp(-z(x))} \quad z(x) = W^T x + b$$

$$L(W, b) = \frac{1}{m} \sum_{i=1}^m -y_i \ln(\theta(x_i)) - (1 - y_i) \ln(1 - \theta(x_i))$$

Untuk mencari **W** dan **b** yang meminimalkan **L** dengan framework Gradient Descent, perlu hitung:

$$\frac{\partial L}{\partial W_j} = ?$$

dan

$$\frac{\partial L}{\partial b} = ?$$

Materi Optional

Perhitungan gradien Binary-Cross Entropy loss function pada Logistic Regression

Logistic Regression Model:

$$\theta(x) = \frac{1}{1 + \exp(-z(x))} \quad z(x) = W^T x + b$$

$$L(W, b) = \frac{1}{m} \sum_{i=1}^m -y_i \ln(\theta(x_i)) - (1 - y_i) \ln(1 - \theta(x_i))$$

Untuk mencari **W** dan **b** yang meminimalkan **L** dengan framework Gradient Descent, perlu hitung:

$$\frac{\partial L}{\partial W_j} = \frac{\partial L}{\partial \theta} \times \frac{\partial \theta}{\partial z} \times \frac{\partial z}{\partial W_j} \quad \text{dan} \quad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial \theta} \times \frac{\partial \theta}{\partial z} \times \frac{\partial z}{\partial b}$$

Materi Optional

Perhitungan gradien Binary-Cross Entropy loss function pada Logistic Regression

$$z(x_i) = W^T x_i + b$$

$$\theta(x_i) = \frac{1}{1 + \exp(-z(x_i))}$$

Kita sederhanakan: untuk satu buah instance!

$$L(W, b) = \frac{1}{m} (-y_i \ln(\theta(x_i)) - (1 - y_i) \ln(1 - \theta(x_i)))$$

$$\frac{\partial L}{\partial \theta} = \frac{1}{m} \left(-\frac{y_i}{\theta} + \frac{1 - y_i}{1 - \theta} \right) \quad \frac{\partial \theta}{\partial z} = \frac{\exp(z)}{(1 + \exp(z))^2} = \theta(1 - \theta) \quad \frac{\partial z}{\partial W_j} = x_j$$

$$\frac{\partial L}{\partial W_j} = \frac{\partial L}{\partial \theta} \times \frac{\partial \theta}{\partial z} \times \frac{\partial z}{\partial W_j} = \frac{1}{m} (\theta - y_i) x_j = \frac{1}{m} (y_i^{pred} - y_i^{true}) x_j$$

Materi Optional

Perhitungan gradien Binary-Cross Entropy loss function pada Logistic Regression

$$z(x_i) = W^T x_i + b$$

$$\theta(x_i) = \frac{1}{1 + \exp(-z(x_i))}$$

Kita sederhanakan: untuk satu buah instance!

$$L(W, b) = \frac{1}{m} (-y_i \ln(\theta(x_i)) - (1 - y_i) \ln(1 - \theta(x_i)))$$

$$\frac{\partial L}{\partial \theta} = \frac{1}{m} \left(-\frac{y_i}{\theta} + \frac{1 - y_i}{1 - \theta} \right) \quad \frac{\partial \theta}{\partial z} = \frac{\exp(z)}{(1 + \exp(z))^2} = \theta(1 - \theta) \quad \frac{\partial z}{\partial b} = 1$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \theta} \times \frac{\partial \theta}{\partial z} \times \frac{\partial z}{\partial b} = \frac{1}{m} (\theta - y_i) = \frac{1}{m} (y_i^{pred} - y_i^{true})$$

Materi Optional

Perhitungan gradien Binary-Cross Entropy loss function pada Logistic Regression

untuk satu buah instance:

$$\frac{\partial L}{\partial W_j} = \frac{1}{m} (\theta - y_i) x_j = \frac{1}{m} (y_i^{pred} - y_i^{true}) x_j$$

$$\frac{\partial L}{\partial b} = \frac{1}{m} (\theta - y_i) = \frac{1}{m} (y_i^{pred} - y_i^{true})$$

untuk semua instances di sebuah batch training set (ada m):

$$\frac{\partial L}{\partial W_j} = \frac{1}{m} \sum_{i=1}^m (\theta - y_i) x_j = \frac{1}{m} \sum_{i=1}^m (y_i^{pred} - y_i^{true}) x_j \quad \frac{\partial L}{\partial b} = (\theta - y_i) = \frac{1}{m} \sum_{i=1}^m (y_i^{pred} - y_i^{true})$$