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* Some slides were originally made by Alfan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.

Supervised Learning

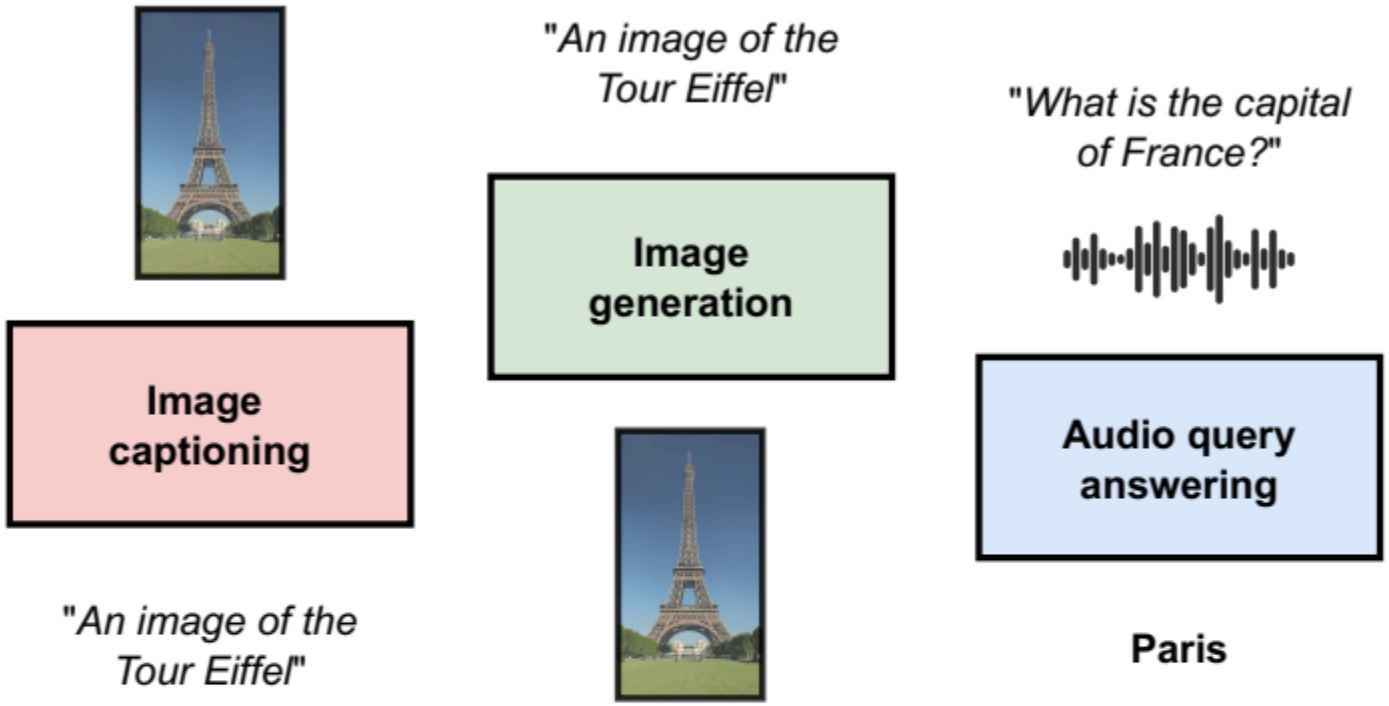


Figure F.1.2: *Most tasks can be categorized based on the desired input - output we need: image generation wants an image (an ordered grid of pixels) from a text (a sequence of characters), while the inverse (image captioning) is the problem of generating a caption from an image. As another example, 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.*

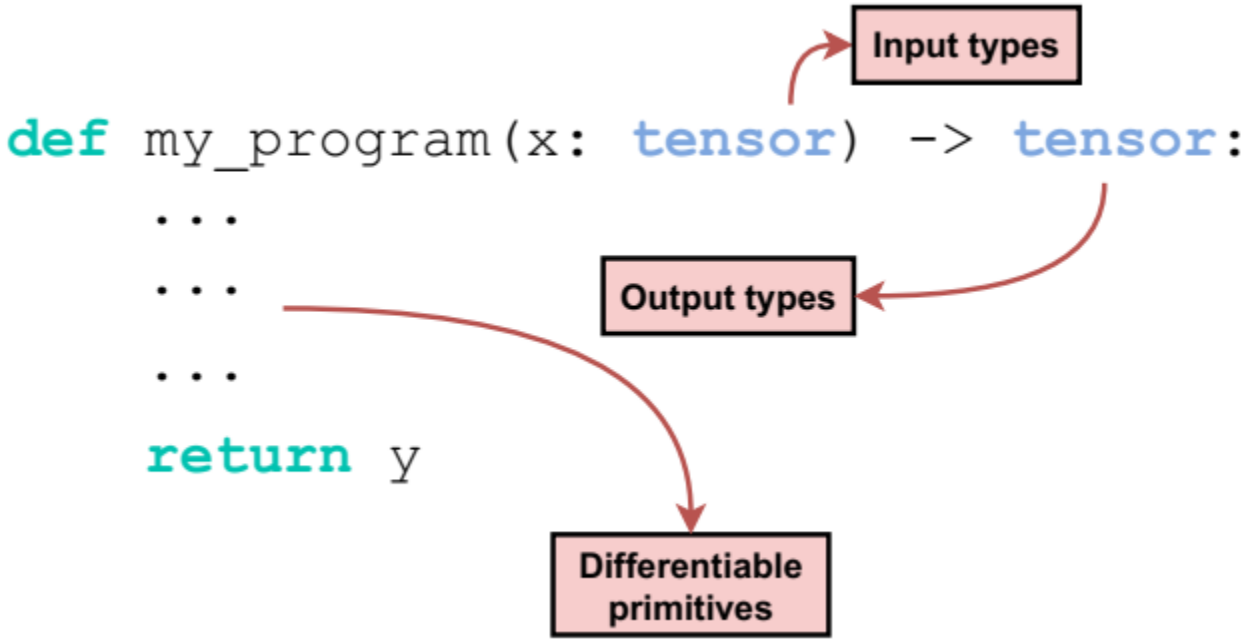
Learning is a **search** problem

- We start by defining a program with a large number of degree-of-freedoms (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

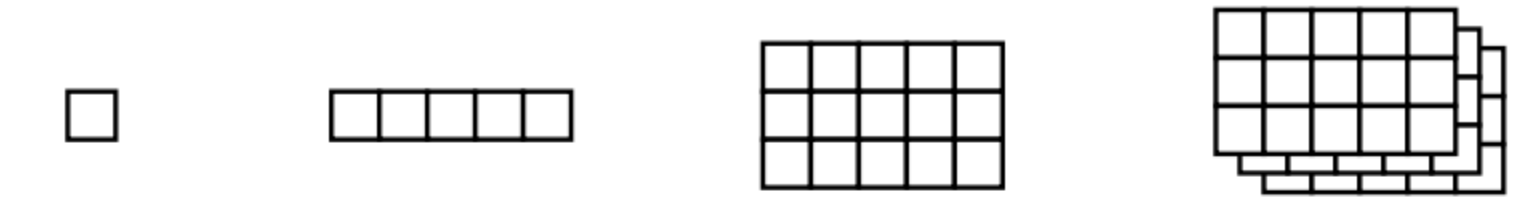
Figure F.1.3: *Neural networks are sequences of differentiable* ***primitives*** *which operate on structured arrays (****tensors****): each primitive can be categorized based on its input/output signature, which in turn defines the rules for composing them.*



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.



$n = 0$ $n = 1$ $n = 2$ $n = 3$

Scalar Vector Matrix *n*-dimensional array

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

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

- 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 $X \sim (s_1, s_2, \ldots, s_n)$ to quickly denote the **shape** of the tensor.*

- Vectors **$\mathbf{x} \sim (d)$** are examples of 1-dimensional tensors.
- In math, we distinguish between **column vectors $\mathbf{x}$** and **row vectors $\mathbf{x}^\top$**.
- 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]])
```

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 (**ℓ2**) norm:

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

Tensors and Their Operators

(**Inner Product**)

The inner product between two vectors **x,y ~ (d)** is given by:

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

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

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

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

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

We say that **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:*

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

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* $\mathbf{Y}$.

Tensors and Their Operators

(**Matrix-Vector Product**)

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

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

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

$$\mathbf{X}\mathbf{W}^\top$$

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([[ 0.7710, -0.4868,  0.6181],
        [ 0.2370,  0.8303,  0.2545]])

>>> torch.sum(a, dim=0)
tensor([1.0080, 0.3435, 0.8726])

>>> torch.sum(a, dim=1)
tensor([0.9023, 1.3218])

>>> torch.sum(a)
tensor(2.2240)
```

Tensors and Their Operators

Batched Matrix Multiplication (BMM)

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

$$[\mathrm{BMM}(X, Y)]_i = \mathbf{X}_i \mathbf{Y}_i \sim (n, a, c)$$

```
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**.

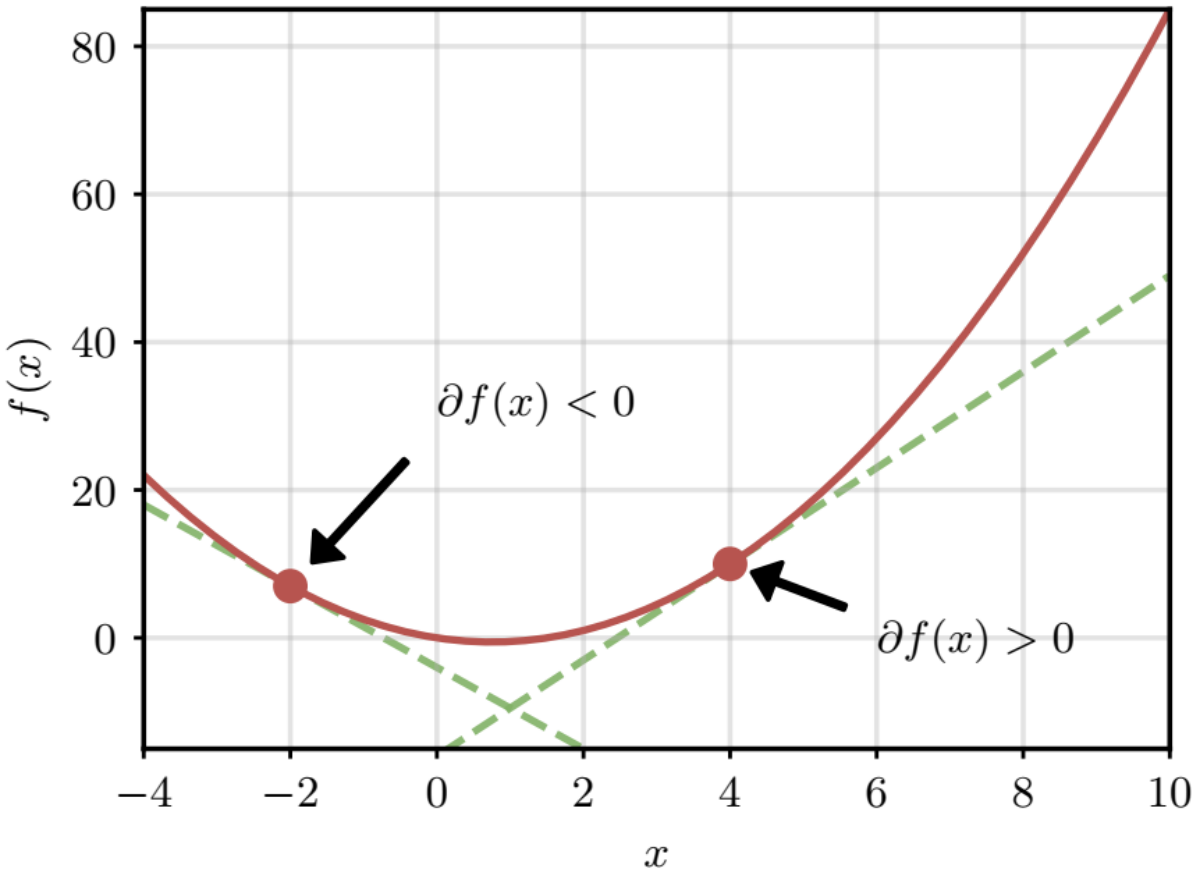
Definition D.2.5 (Derivative) *The **derivative** of $f(x)$ is defined as:*

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{E.2.14}$$

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

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

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

Partial derivatives

Consider a function **y = f(x)** taking a vector **x ~ (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**.

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

This is called **partial derivatives**.

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

Partial 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 v**? In this case we obtain the **directional derivative**:

$$\mathrm{D}_{\mathbf{v}} f(\mathbf{x}) = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h},$$

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

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

Displacement on the i-th axis

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

Jacobians

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

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

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

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

Jacobians

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

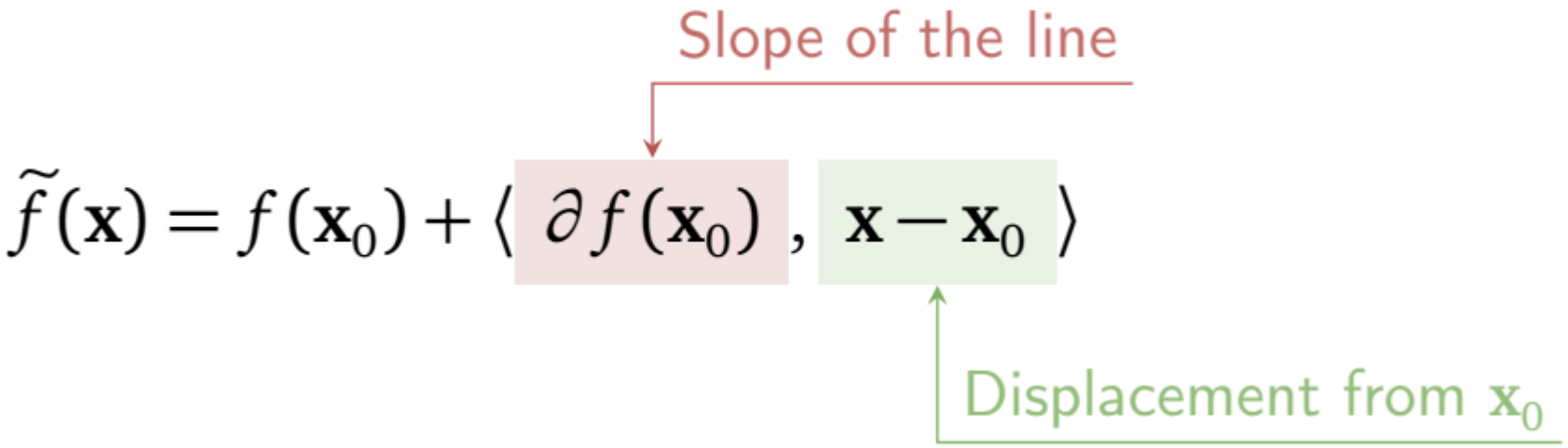
$$\partial\,[f(g(\mathbf{x}))] = [\partial f(\bullet)]\,\partial g(\mathbf{x})$$

where the first derivative is evaluated in **g(x) ~ (h)**.

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 $\mathbf{f}(\mathbf{x}_0)$ is given by:

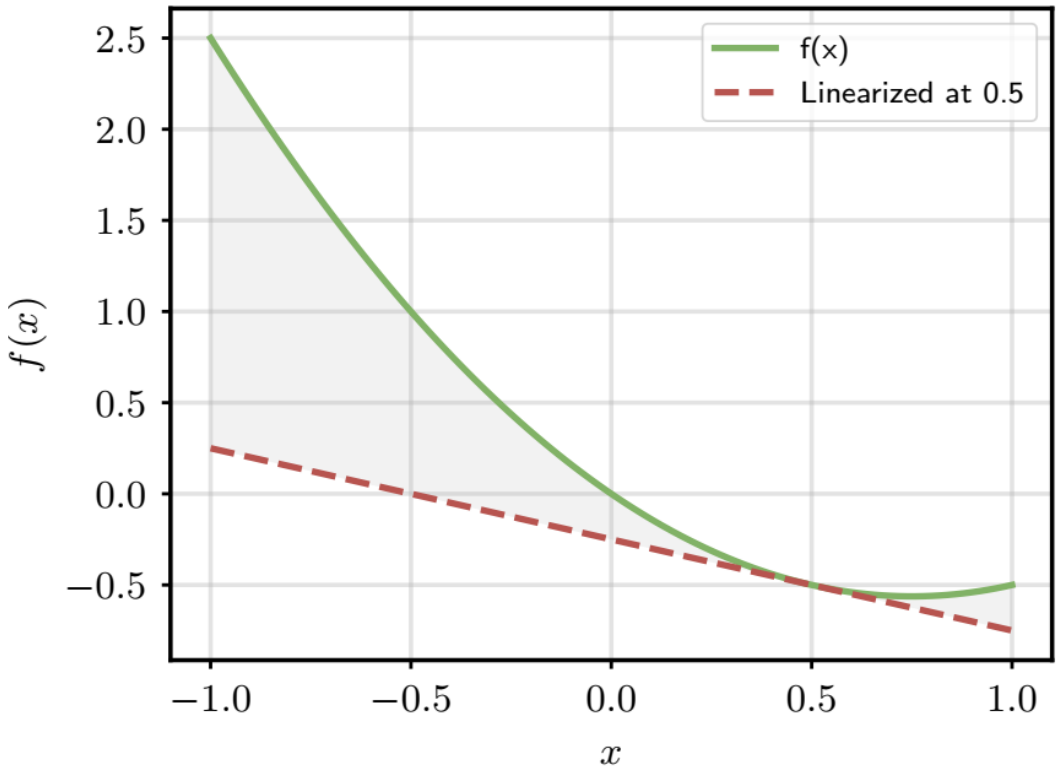


Jacobians

Taylor's Theorem: The gradient is the best **"first-order approximation"**.

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

```python
# 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 **x ~ (d), y ~ (o), W ~ (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 ~ (d)**:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$$

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)**.

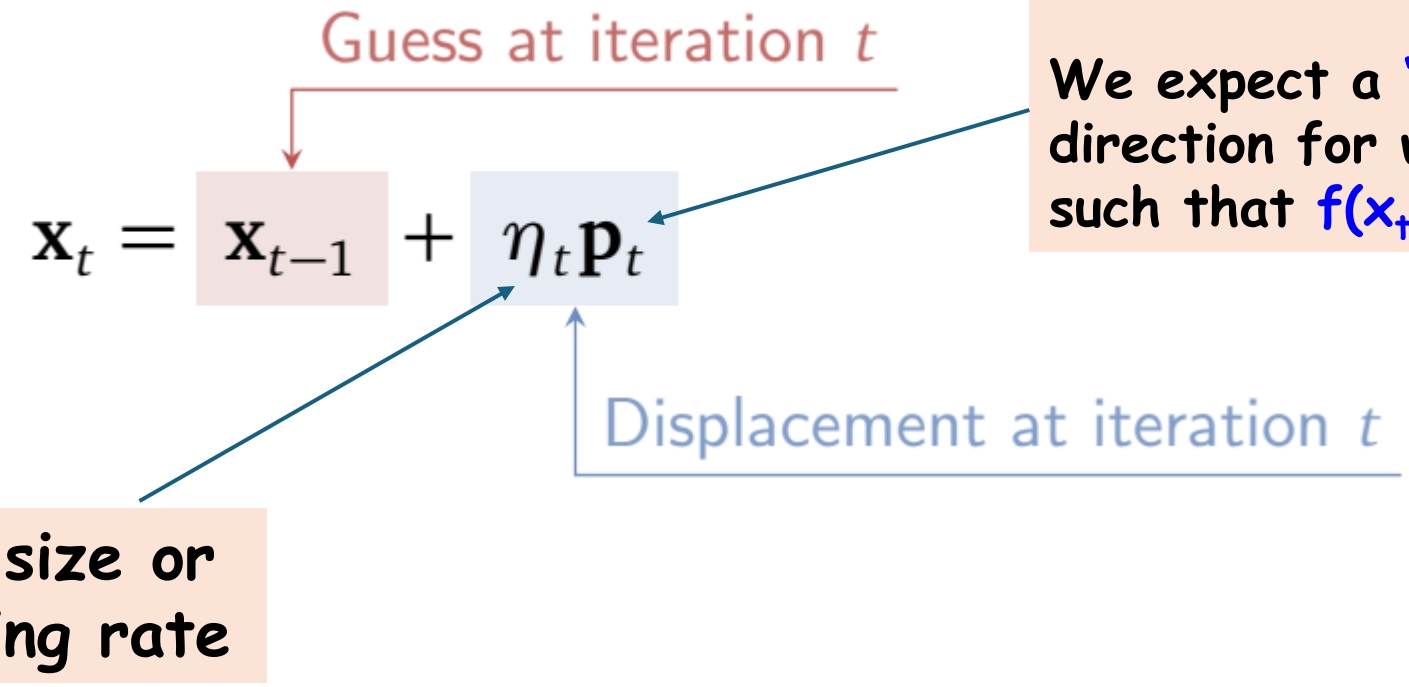
We assume:

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

Optimization Problems

How? Closed-form? → **very rare!**

In general, however, we are forced to resort to iterative procedures, starting from random guess $\mathbf{x}_0$.



A direction.

We expect a "descent direction": a direction for which there exists an η_t such that $f(x_t) \leq f(x_{t-1})$.

Step size or learning rate

Optimization Problems

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

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

We have learned that:

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

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

This is called steepest descent direction (negative gradient)

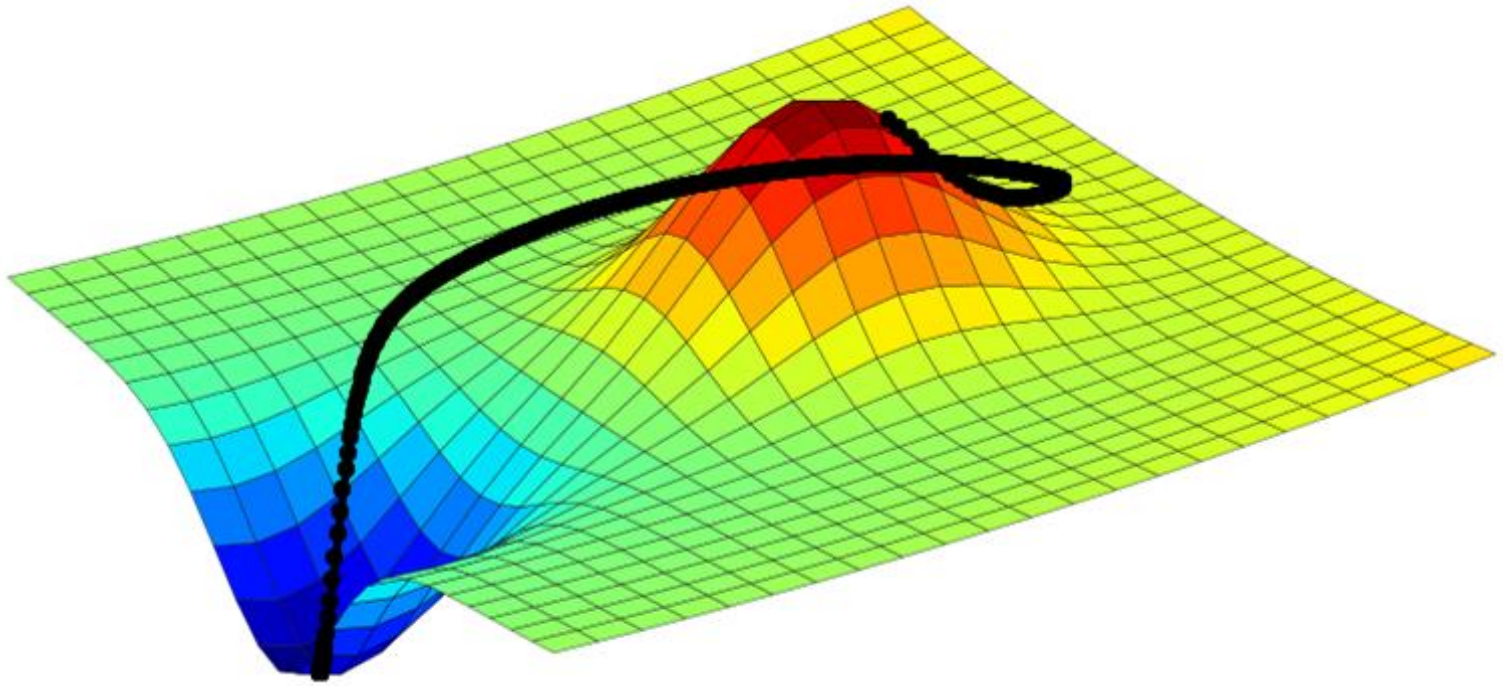
Gradient Descent

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

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).



https://github.com/joshdk/pygradesc

Gradient Descent

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

epoch: max iteration

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

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

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

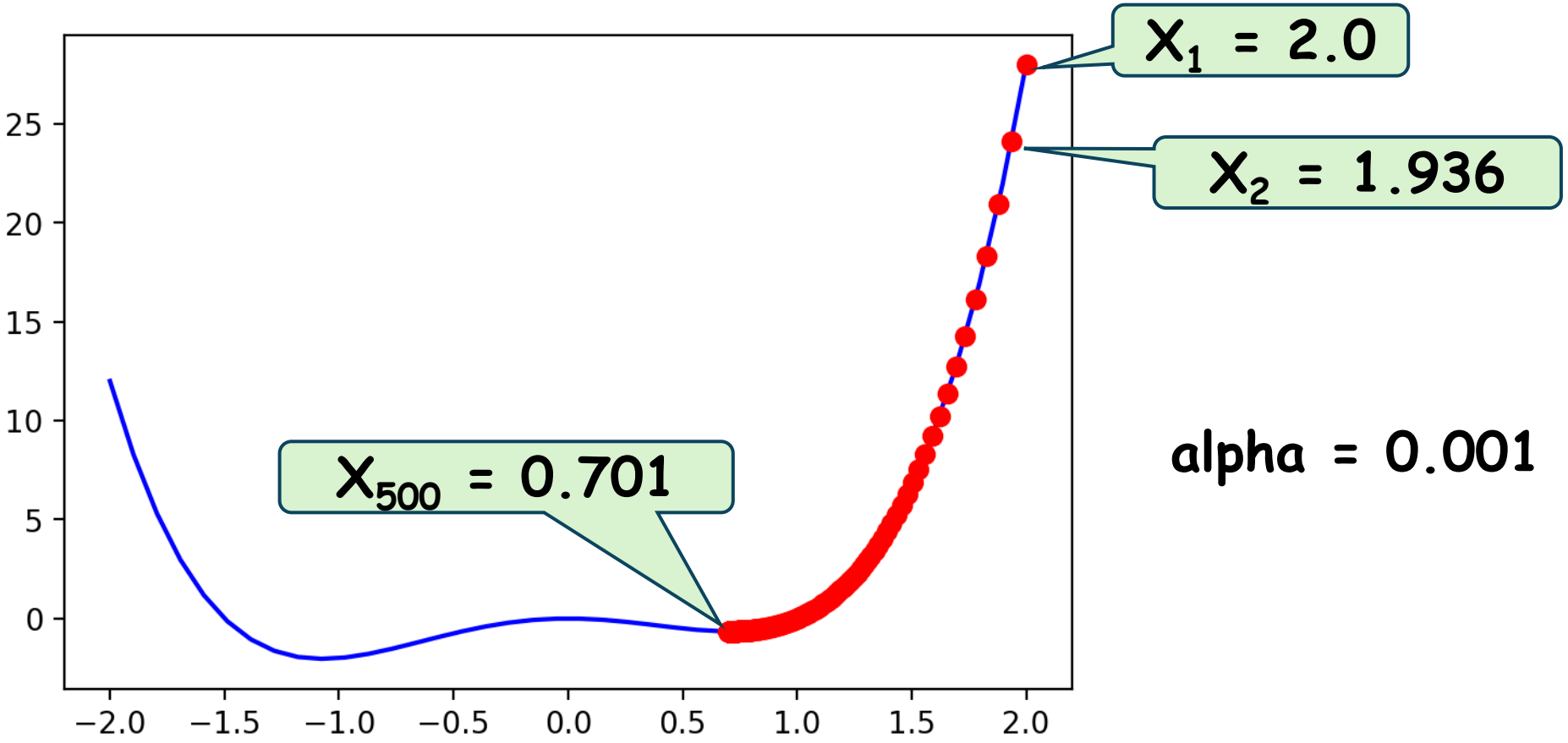
$$\text{if } \left| f'(x_{t+1}) \right| < \epsilon \text{ then return "converged on critical point"}$$

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

learning rate

Gradient Descent

Contoh: carilah parameter **x** sehingga fungsi $\mathbf{f(x) = 2x^4 + x^3 - 3x^2}$ 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 popular 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

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* $\mathbf{x}^+$

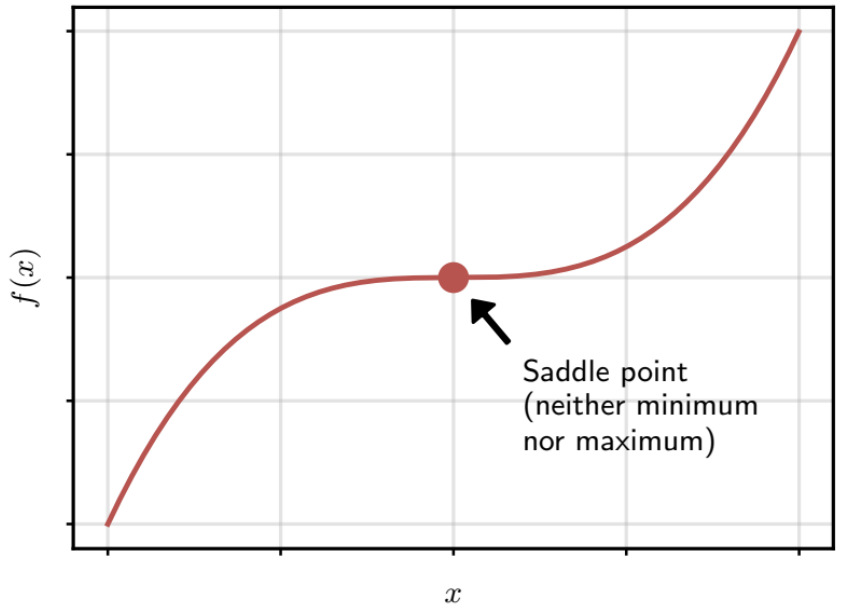
In this point, the slope of the tangent is zero.

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* $\nabla f(\mathbf{x}^+) = 0$.

Stationary points are not limited to minima: they can be maxima (the minima of **−f(x)**) or **saddle points**.



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:*

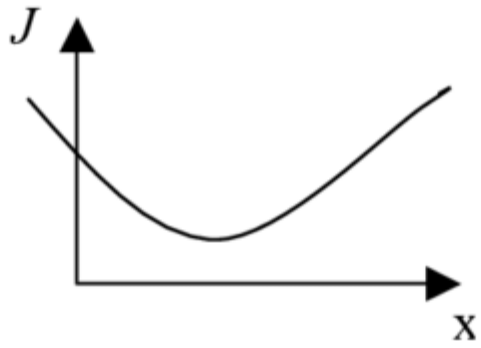
$$f(\alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2) \leq \alpha f(\mathbf{x}_1) + (1-\alpha)f(\mathbf{x}_2)$$

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

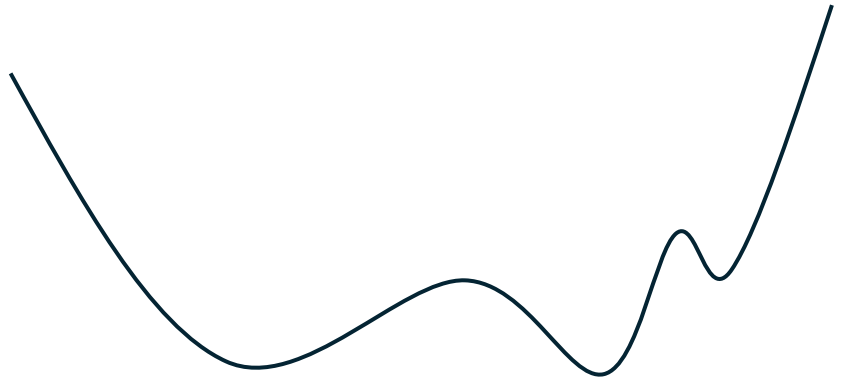
Interval from $\mathbf{x}_1$ *to* $\mathbf{x}_2$

Gradient Descent

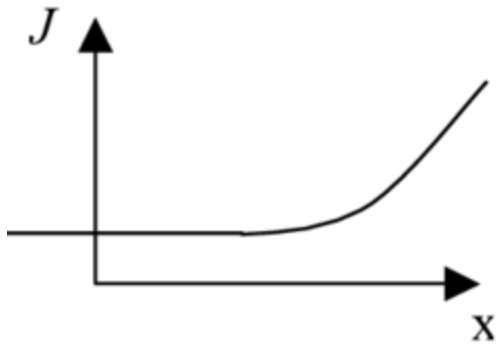
Convergence of gradient descent??

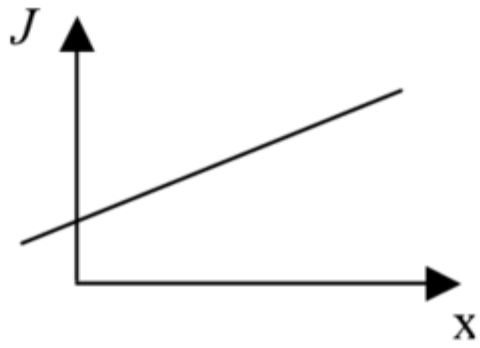


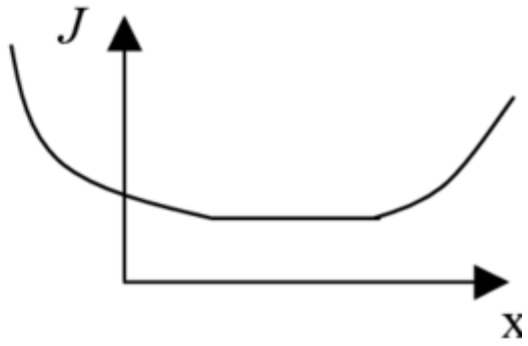
a) convex (and strictly convex)



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**.

Accelerating Gradient Descent --> Momentum

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

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

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:

Normal gradient descent iteration

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

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

$g_0 = 0$

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

Additional momentum term

Accelerating Gradient Descent --> Momentum

Normal gradient descent iteration

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

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

$g_0 = 0$

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

Additional momentum term

If we unroll:

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

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.

> **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 $p(x, y)$.*

Identically distributed → stable & unchanging through time.
Independently distributed → has no bias in its collection; sufficiently representative of the entire distribution.

What is Supervised Datasets?

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

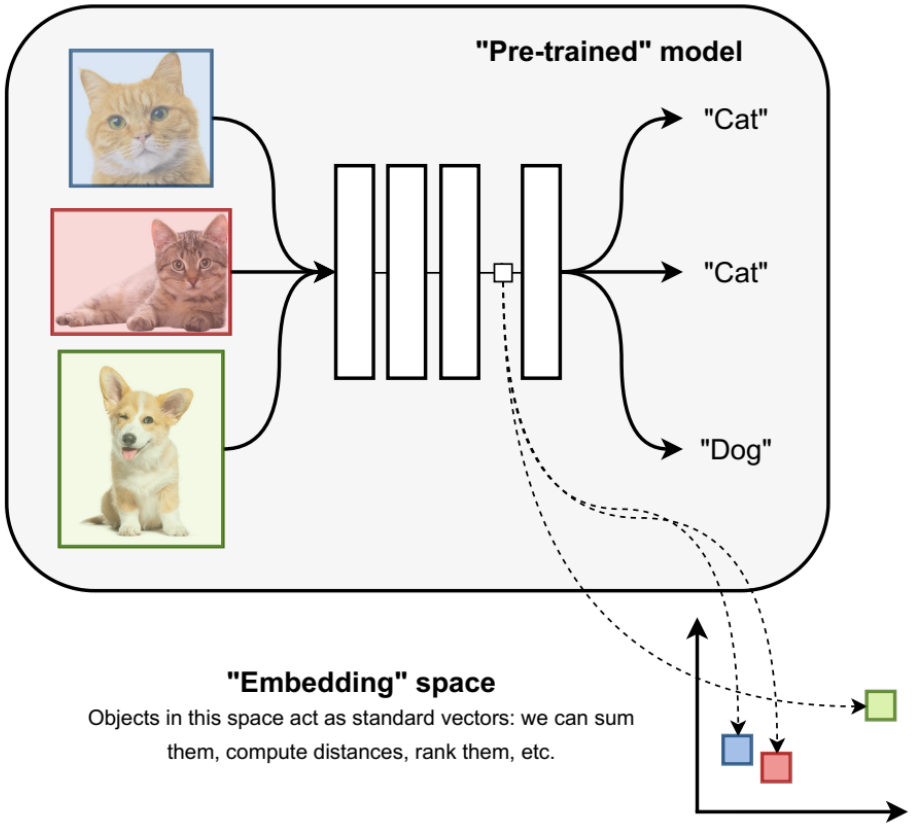


Figure F.3.1: *Differentiable models process data by transforming it sequentially via linear algebra operations. In many cases, after we optimize these programs, the internal representations of the input data of the model (what we call a* ***pre-trained*** *model) have geometric properties: for example, 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

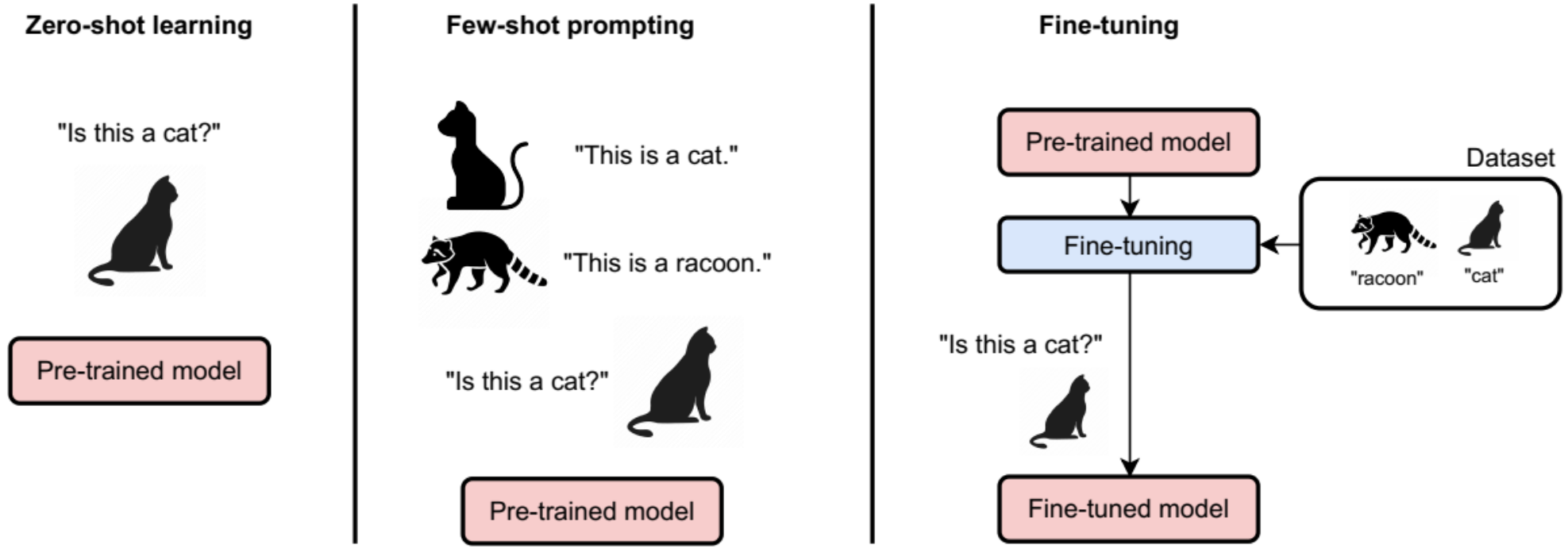


Figure F.3.2: *Three ways of using trained models.* ***Zero-shot****: a question is directly given to the model. This can be achieved with generative language models (introduced in Chapter 8).* ***Few-shot prompting*** *is similar, but a few examples are provided as input. Both techniques can be employed 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**.

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.*

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:

Average over the dataset

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

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

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:

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

It's almost impossible to minimize across the space of all possible functions. In practical, f(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(x, w)**.

Examples of Loss Functions – Binary Classification

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

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

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

A possible loss function in this case is the **hinge loss (Support Vector Machine)**:

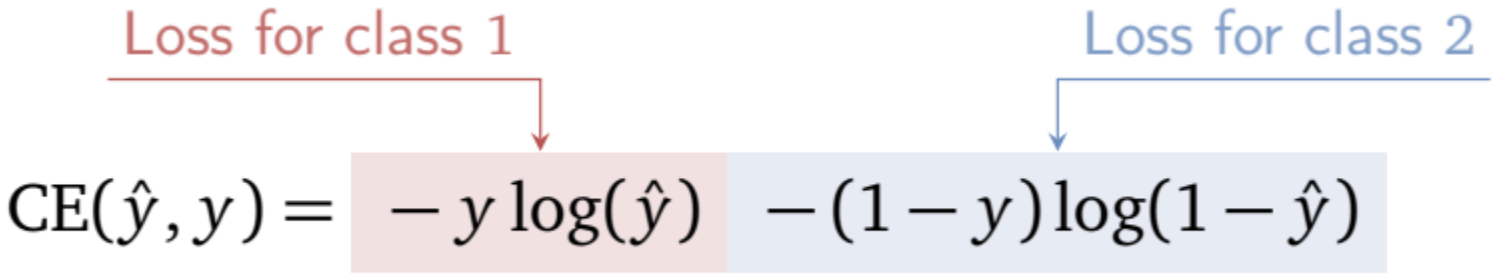
$$l(y, \hat{y}) = \max(0, 1 - y\hat{y})$$

Note that $\hat{y}$ is the classifier output and is not the predicted label!

Examples of Loss Functions – Binary Classification

Other possible loss functions:

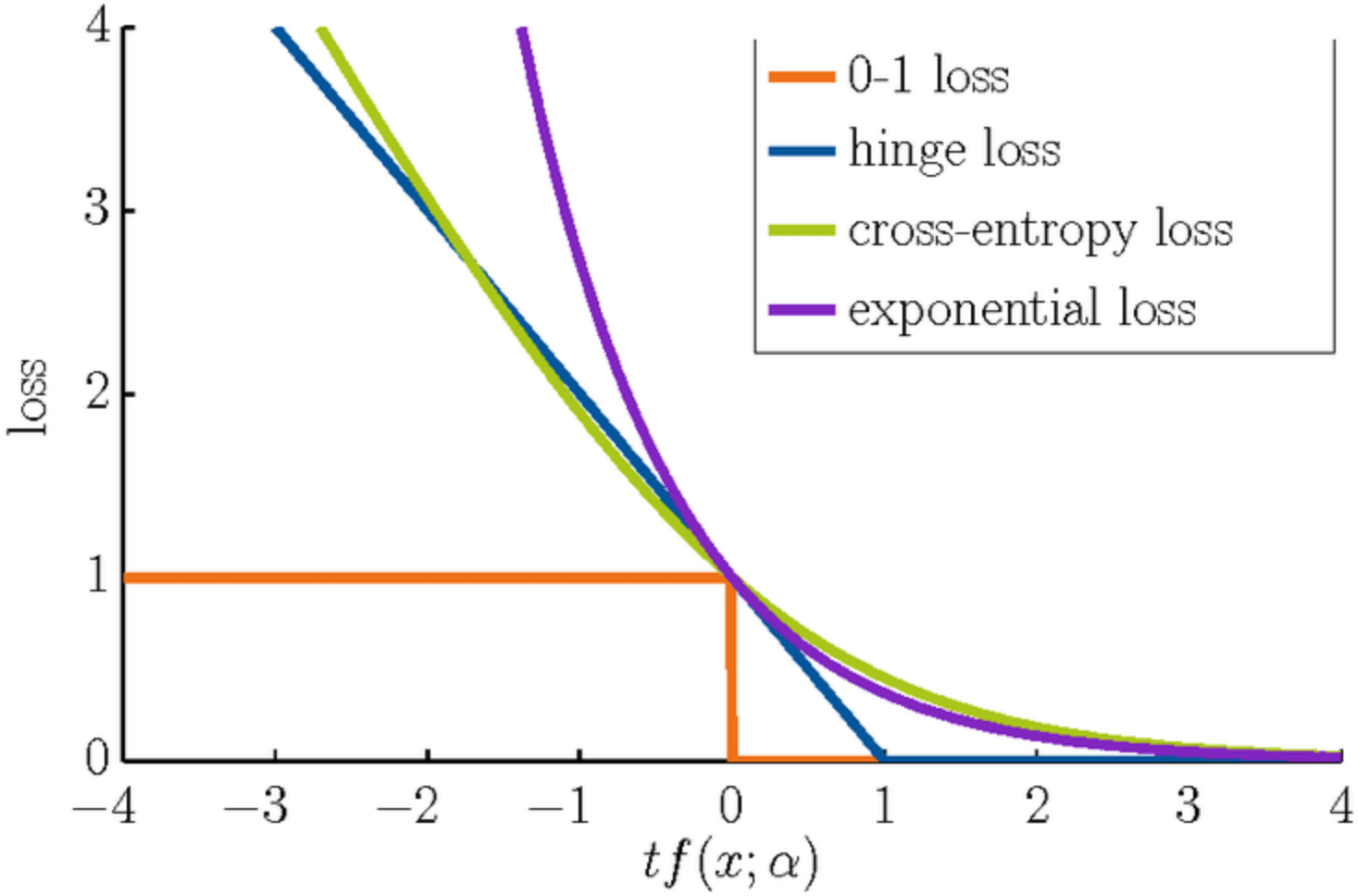
Binary Cross Entropy (Logistic Regression):



Exponential Loss (ADAboost):

$$l(y, \hat{y}) = \exp[-y\hat{y}]$$

Examples of Loss Functions – Binary Classification



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 ~ (c),** corresponding to a set of features;
- The output is a single real value **y ∈ R** (**regression task**);
- 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**:

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

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**:

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

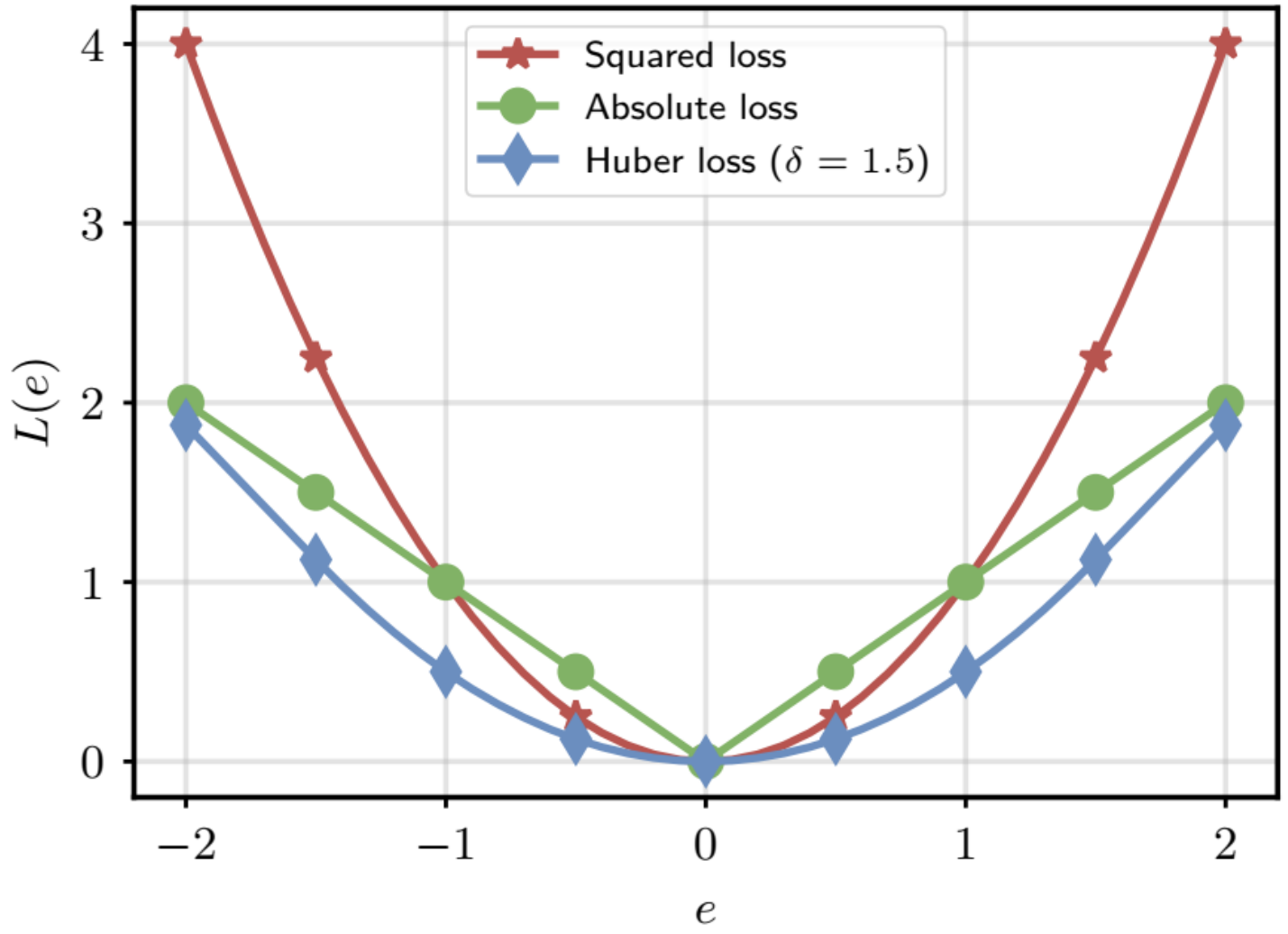
or the **Huber loss** (a combination of the squared loss and the absolute loss):

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

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:*

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

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:*

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

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

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:

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

Size of dataset (num instances)

Num features

Output Vector:

$$y = [y_1, \ldots, y_n]^\top$$

Batched Model Output:

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

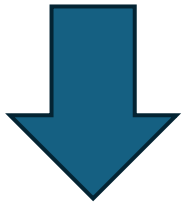
Same bias b for all n predictions

Least-squares Regression

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

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

The vectorized least-squares optimization problem



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

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

Least-squares Regression: How to solve?

Suppose we ignore the bias (bias can be easily incorporated into **W**):

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

The gradient w.r.t **w** is:

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

Can you show this?

Least-squares Regression: How to solve?

We can actually find optimal **w** using a closed form formula:

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

Can you show this?

Performing the inversion in $(\boldsymbol{X}^T\boldsymbol{X})^{-1}$ is not always possible. for example, if **one feature is a scalar multiple of the other**, the matrix **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:

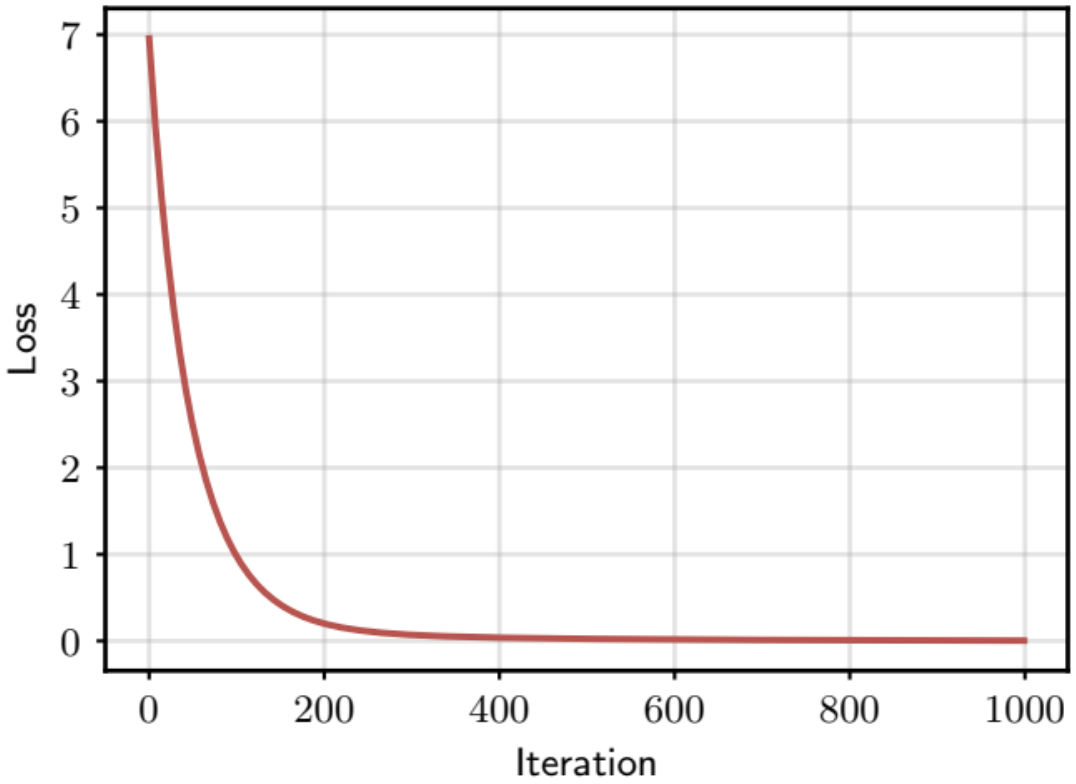
```
def least_squares_gd(X: Float[Tensor, "n c"],
                     y: Float[Tensor, "n"],
                     learning_rate=1e-3) \
                     -> Float[Tensor, "c"]:
    # Initializing the parameters
    w = torch.randn((X.shape[1], 1))

    # 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
```

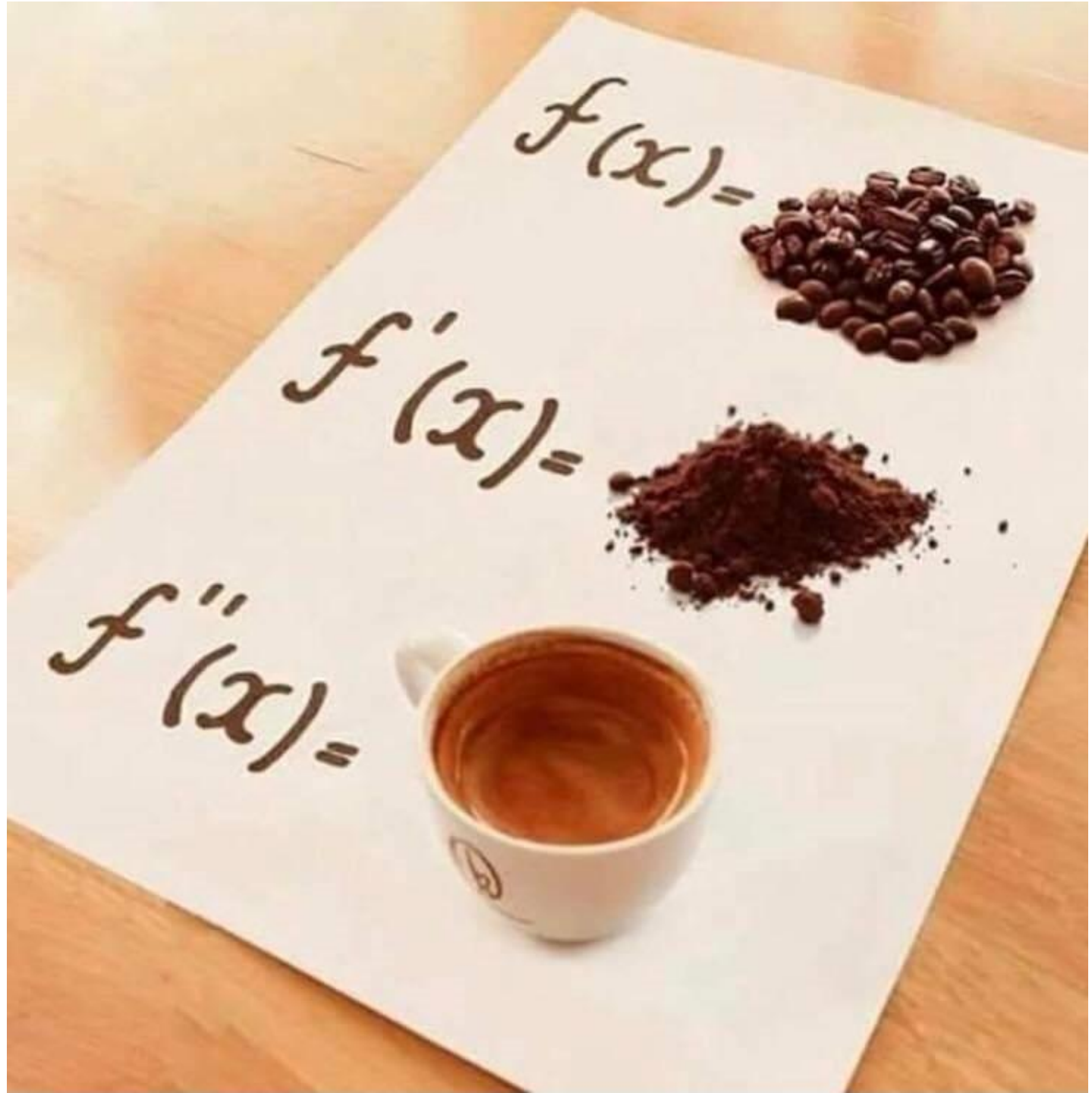
Fixed learning rate = 0.001

Least-squares Regression: How to solve?

But, we are more interested in Gradient Descent method:

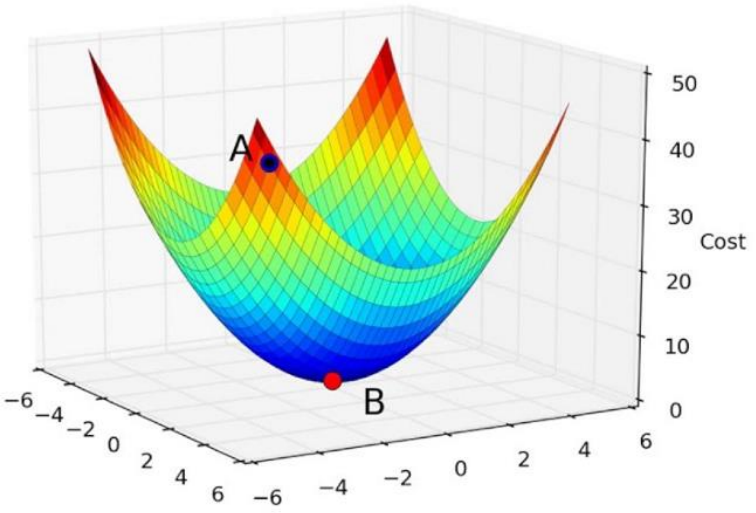
Figure F.4.2: *An example of running code from Box C.4.2, where the data is composed of $n = 10$ points drawn from a linear model $\mathbf{w}^\top \mathbf{x} + \varepsilon$, with $w_i \sim \mathcal{N}(0,1)$ and $\varepsilon \sim \mathcal{N}(0, 0.01)$. Details apart, note the very smooth descent: each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




f(x)=
f'(x)=
f''(x)=

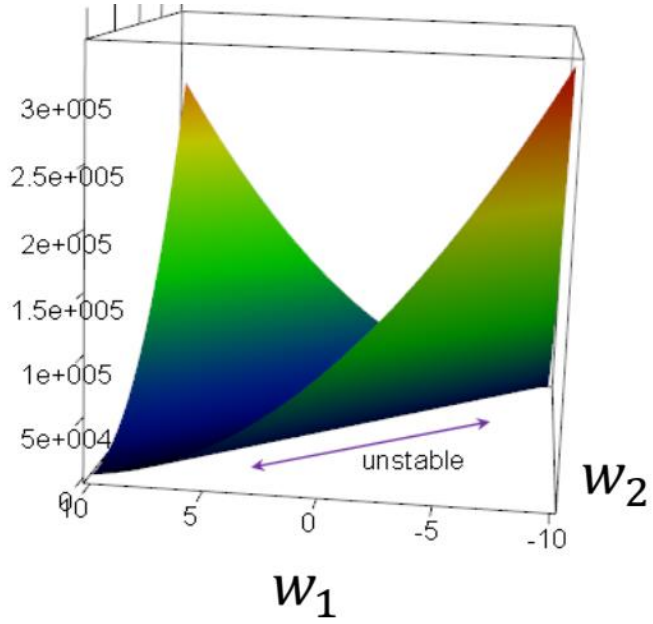
Least-squares Regression: How to solve?

- **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



https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b

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**)

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

$$f(\boldsymbol{x}) = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

Misalkan, solusi-nya adalah: $\left[\hat{w_1}, \hat{w_2}, \hat{w_3}, \hat{b}\right]$

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

contoh: $\left[\hat{w_1} + 0.5, \hat{w_2} - 0.5, \hat{w_3}, \hat{b}\right]$

$\left[\hat{w_1} - 0.1, \hat{w_2} + 0.1, \hat{w_3}, \hat{b}\right]$

Dua solusi ini akan memberikan nilai f(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

di

ini akan
an nilai
g sama

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**.

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

Irrelevant Features: Multi-Collinearity

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

Fitur $\mathbf{x_j}$ **irrelevant** jika $\mathbf{x_j}$ adalah **kombinasi linear** dari fitur lain.

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

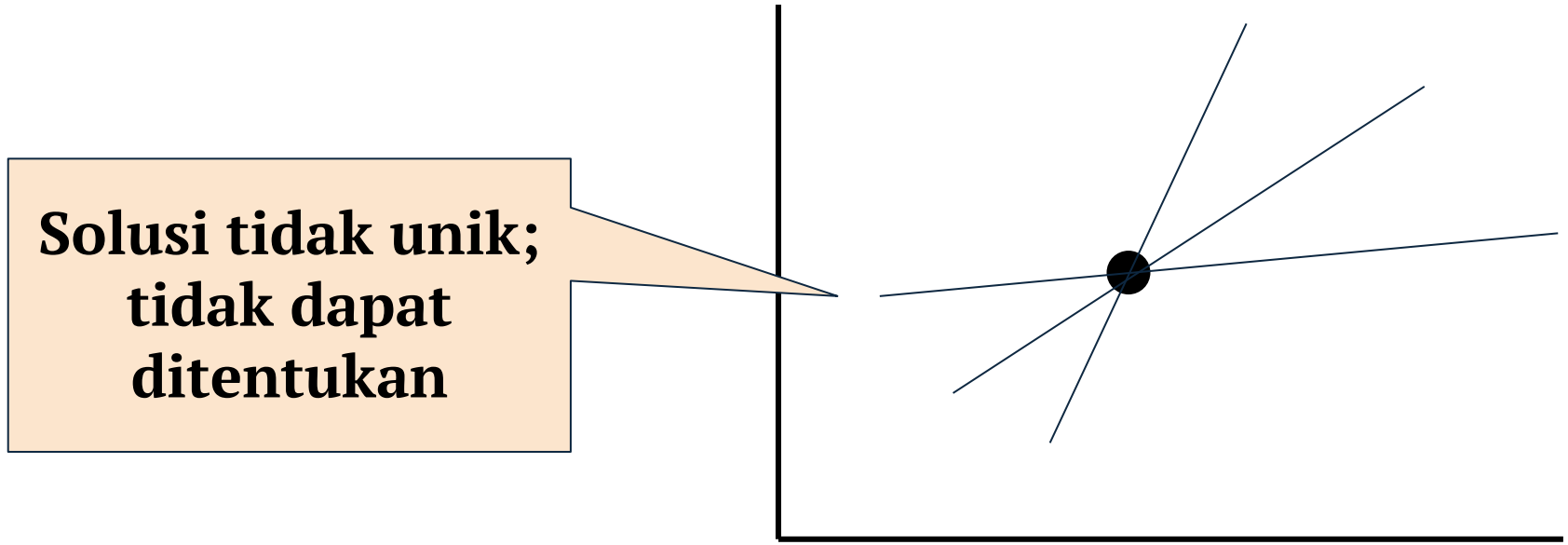
Hal-hal seperti ini bisa saja terjadi terutama ketika **fitur sangat banyak** (high-dimensional)

Ill-posed problem berikutnya: model terlalu kompleks

Model lebih kompleks dibandingkan data.

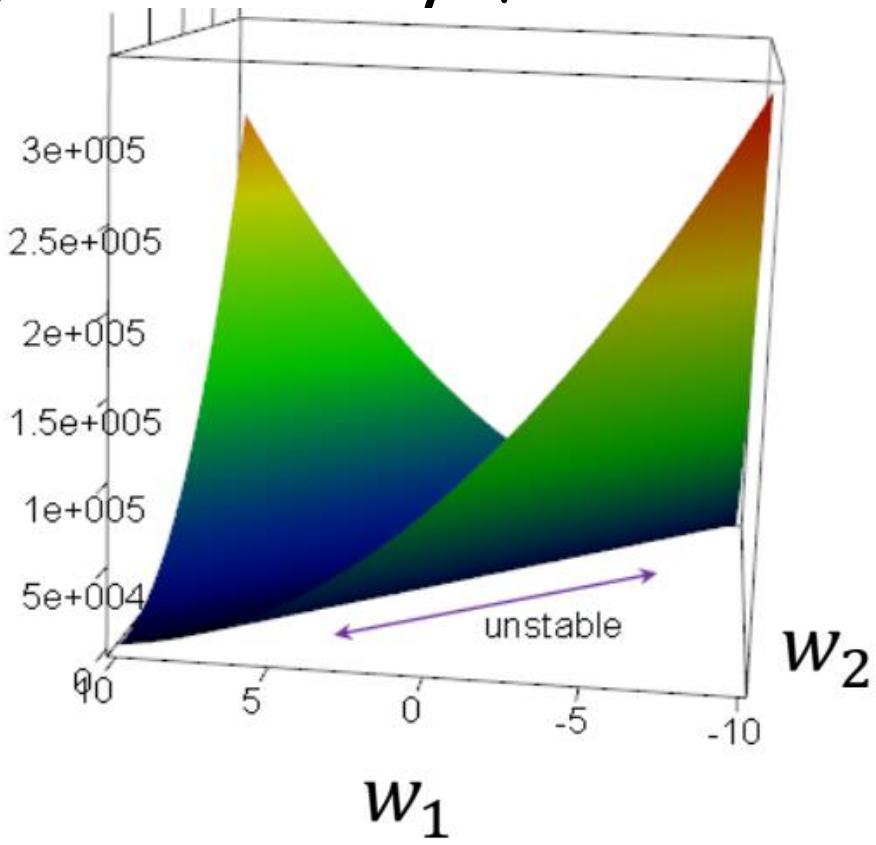
Kasus ekstrim:

- 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**:

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

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:

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

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

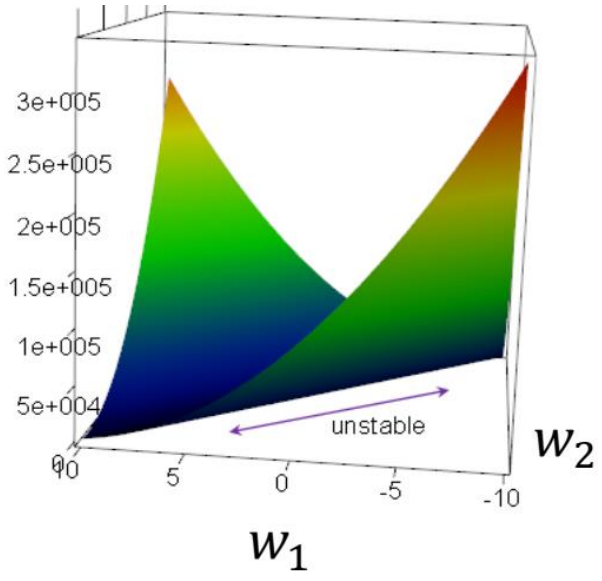
$$LS(\boldsymbol{w}) = \frac{2}{n}\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

Regularization hyper-parameter

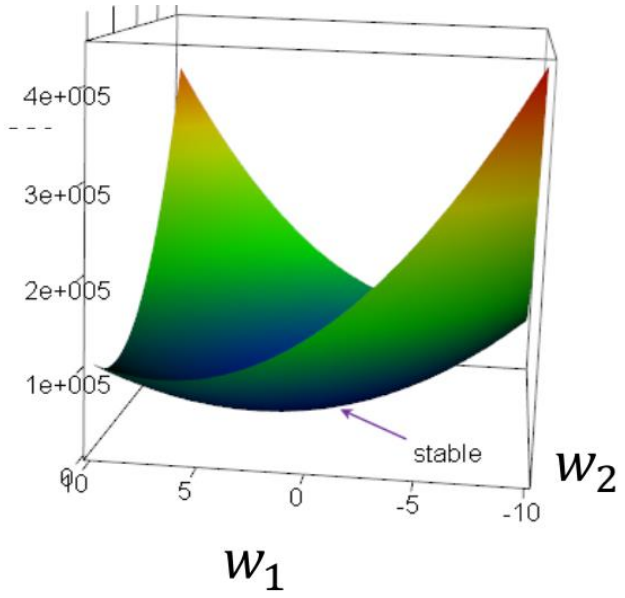
L2-Regularization

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

Solutions for irrelevant features?



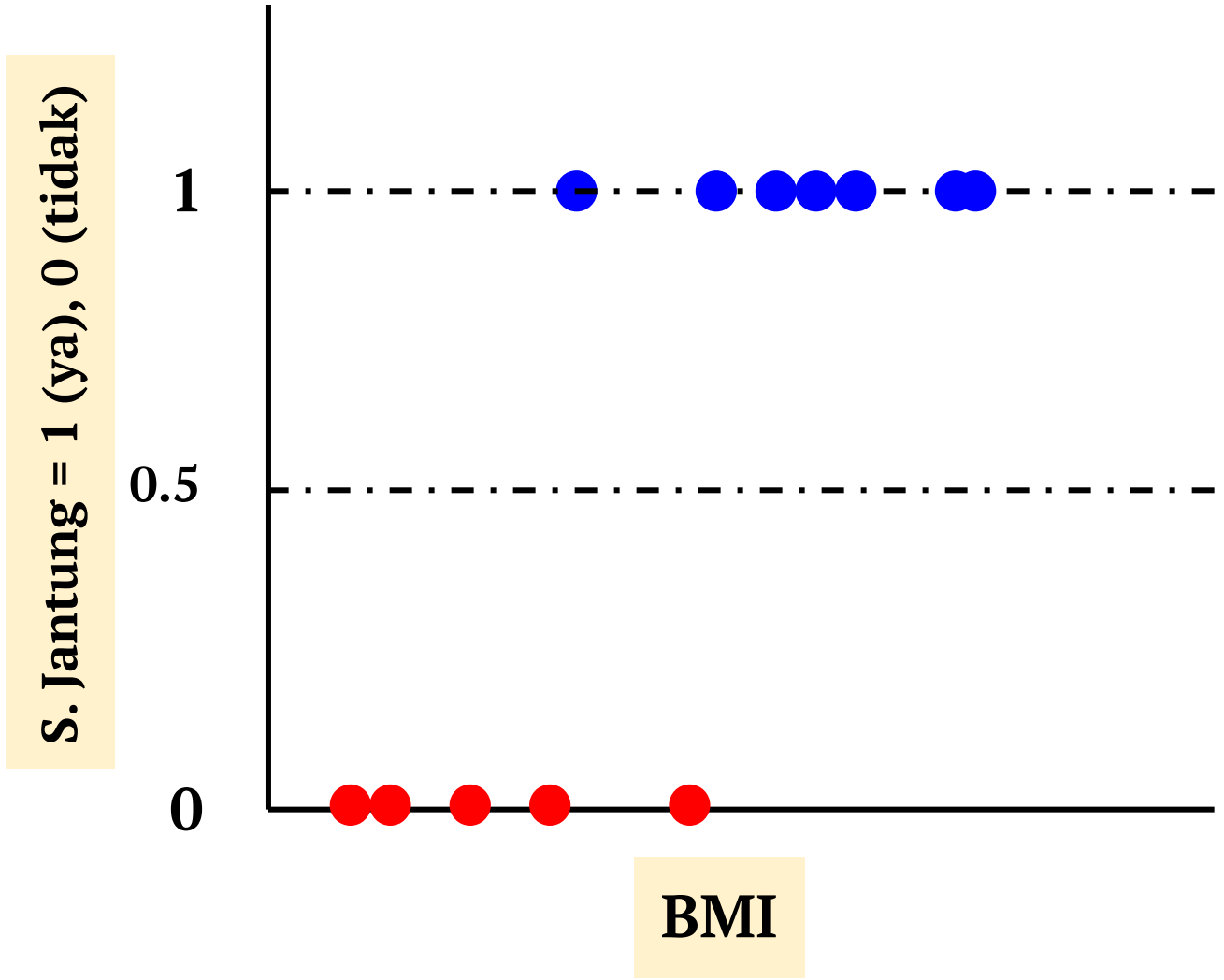
Sebelum diregularisasi



Setelah diregularisasi

Linear Models
(Binary Classification)

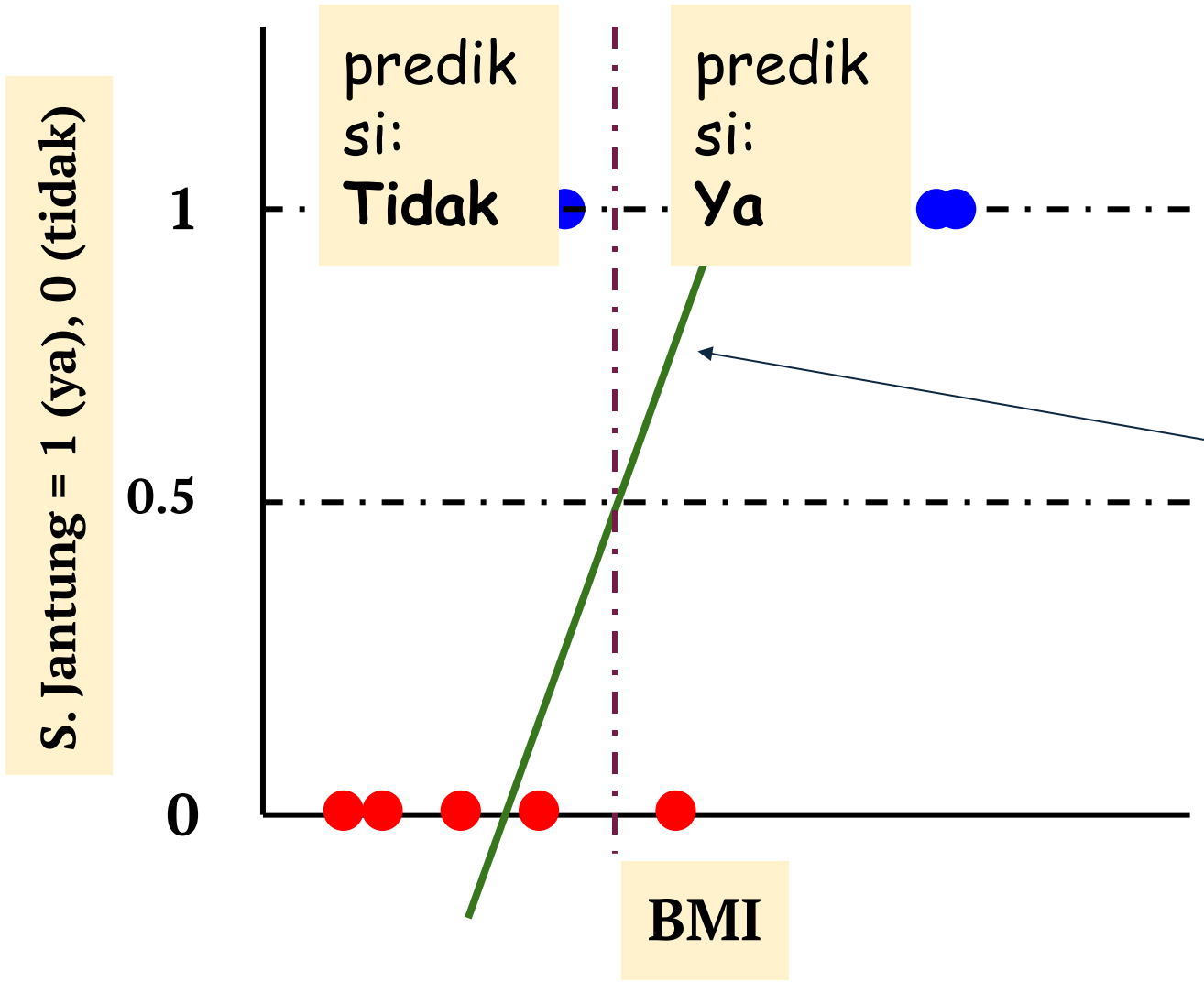
Binary Classification



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?

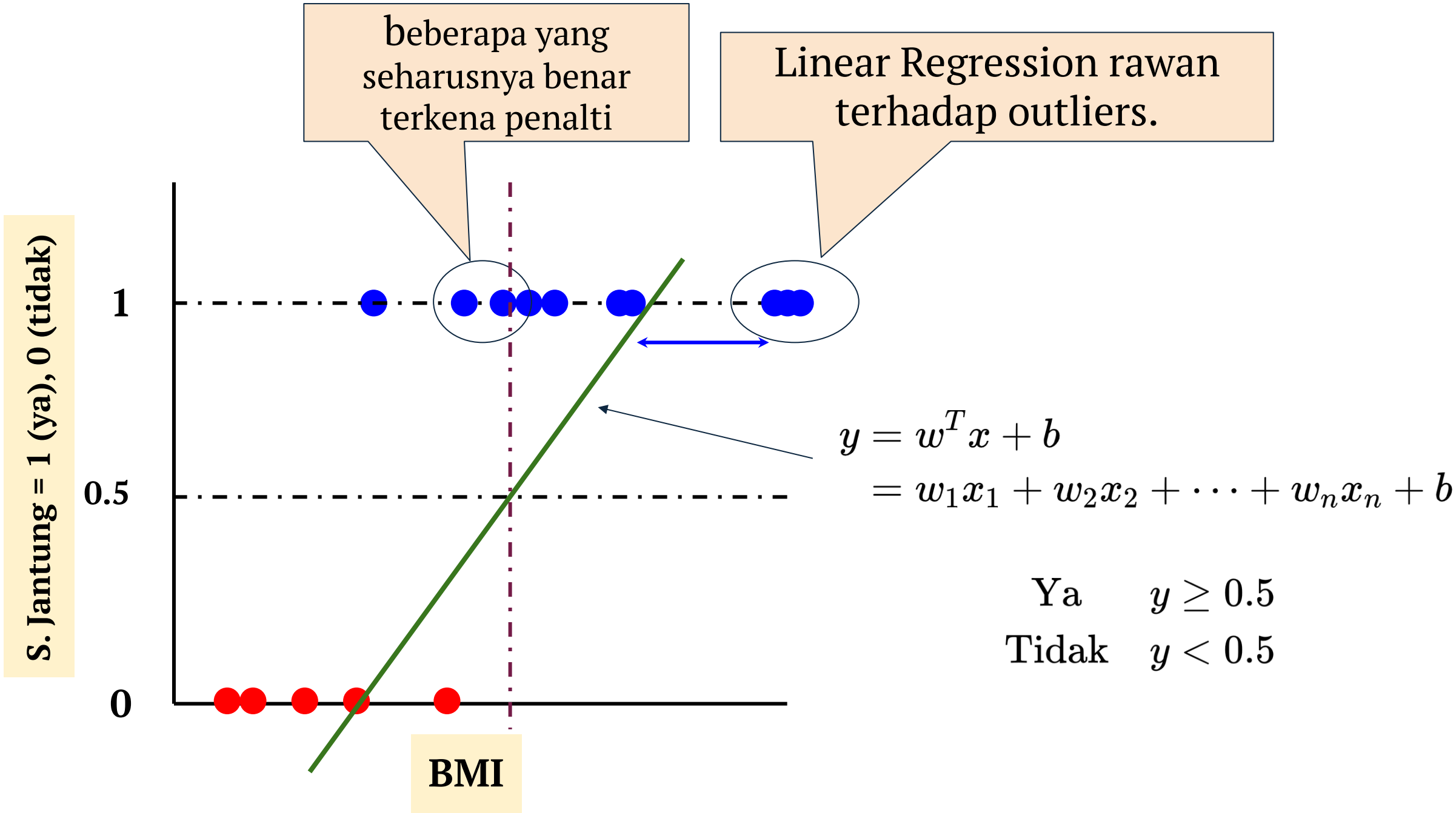
Binary Classification



Memangnya **linear regression** biasa tidak bisa?

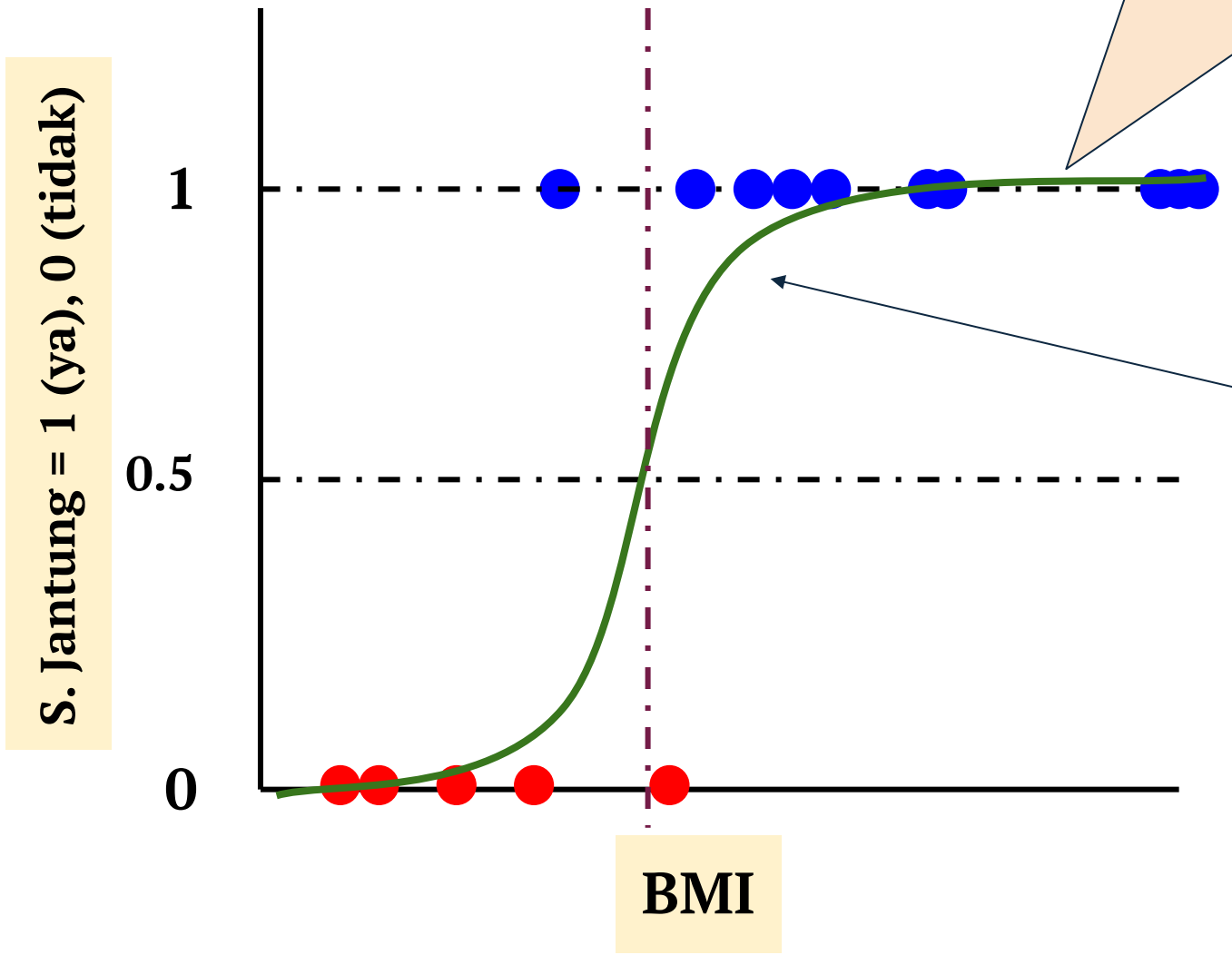
$$y = w^T x + b$$
$$= w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

$$\begin{array}{ll} \text{Ya} & y \geq 0.5 \\ \text{Tidak} & y < 0.5 \end{array}$$


beberapa yang seharusnya benar terkena penalti
Linear Regression rawan terhadap outliers.
S. Jantung = 1 (ya), 0 (tidak)
1
0.5
0
BMI
$y = w^T x + b$
$= w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$
Ya $y \geq 0.5$
Tidak $y < 0.5$

Binary Classification

Akan lebih baik jika garis pemisah **melengkung** seperti ini

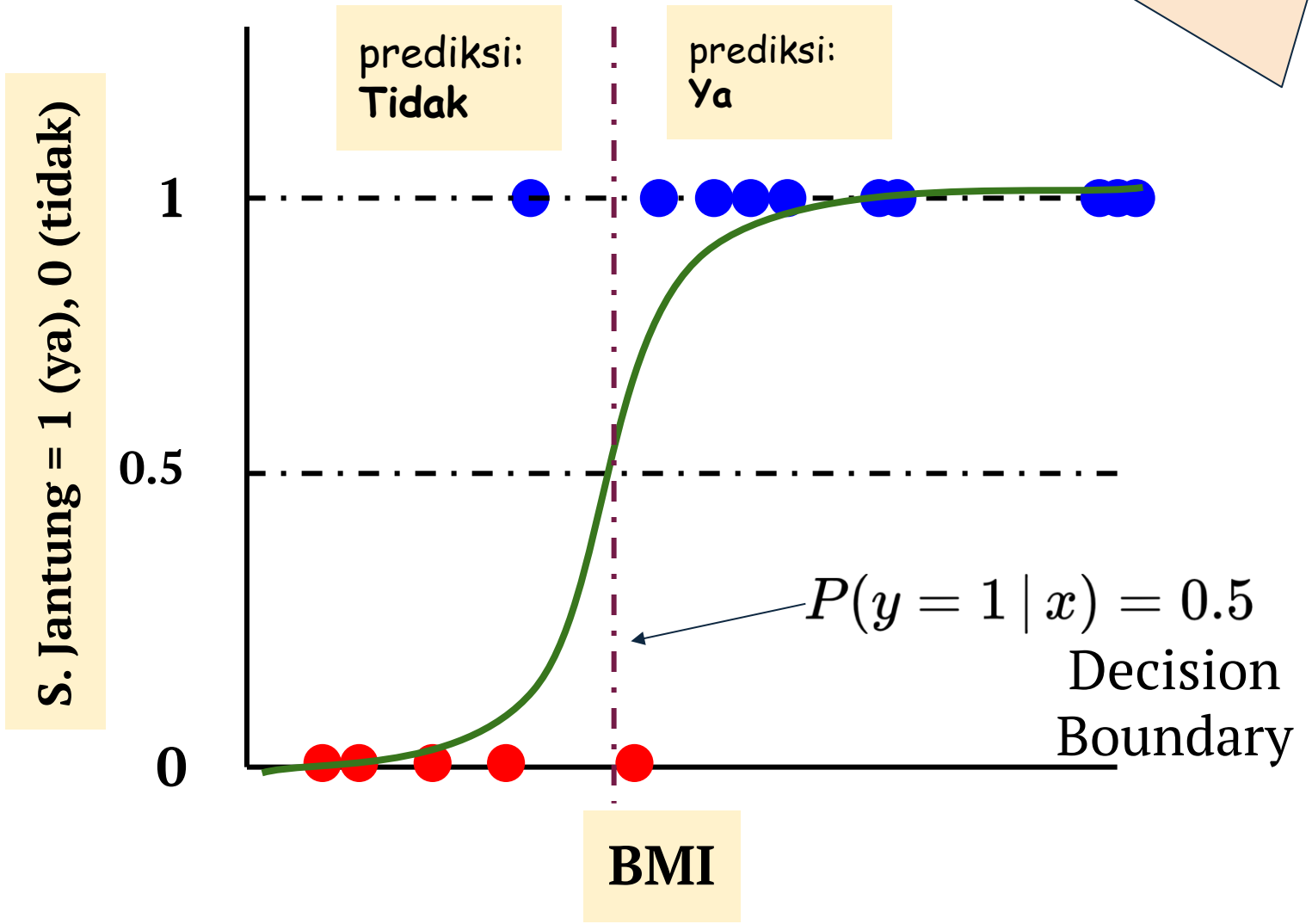


Ada fungsi yang sifatnya seperti ini, yaitu **logistic function**:

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

Binary Classification

Logistic Regression = Linear Regression + Logistic Function



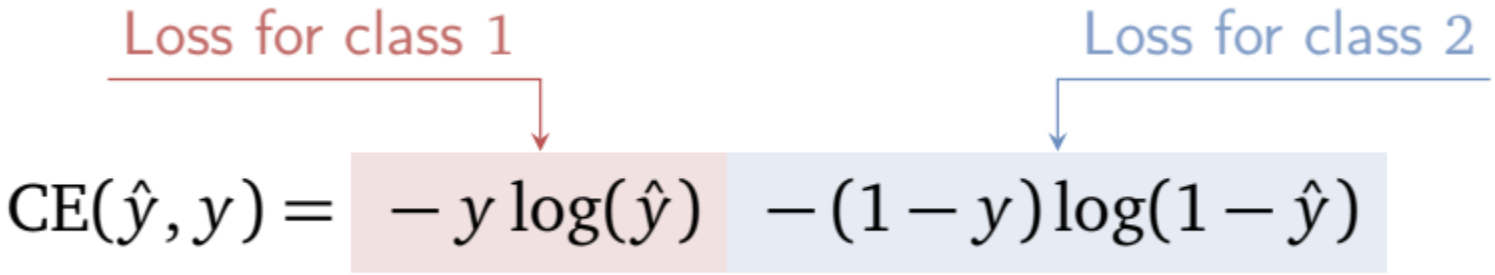
$$P(y = 1|x) = \sigma(w^T x + b)$$

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

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

Binary Classification

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



Consider the gradient of the binary logistic regression model with respect to **w** :

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

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**.

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

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

Logits

Sigmoid inverse: $\sigma^{-1}(y)$

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

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!

$$\theta(x) = \frac{1}{1 + \exp(-z(x))} \qquad z(x) = W^T x + b \qquad x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad w = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

Loss Function untuk training W dan b:

n = Banyaknya fitur

Banyaknya instances di training data

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

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

Materi Optional

Perhitungan gradien Binary-Cross Entropy loss function pada Logistic Regression

Logistic Regression Model:

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

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

Loss Function untuk training W dan b:

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

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

Materi Optional

Perhitungan gradien Binary-Cross Entropy loss function pada Logistic Regression

Logistic Regression Model:

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

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

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

$$\frac{\partial L}{\partial W_j} = ? \qquad \text{dan} \qquad \frac{\partial L}{\partial b} = ?$$

Materi Optional

Perhitungan gradien Binary-Cross Entropy loss function pada Logistic Regression

Logistic Regression Model:

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

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

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

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

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!

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

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

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

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!

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

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

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

Materi Optional

Perhitungan gradien Binary-Cross Entropy loss function pada Logistic Regression

untuk satu buah instance:

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

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

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

Implementasi Logistic Regression "from the scratch"

Misal, kita mempunyai **dummy training data**:

```
import numpy as np

# representasi data atau features
X = np.array([[0.4, 0.5, 0.8, 0, 0, 0],
              [0.2, 0.7, 0.4, 0, 0, 0],
              [0.7, 0.2, 0.7, 0, 0, 0],
              [0, 0, 0, 0.6, 0.9, 0.4],
              [0, 0, 0, 0.5, 0.4, 0.2],
              [0, 0, 0, 0.9, 0.8, 0.5],
              [0, 0, 0, 0.8, 0.3, 0.6]])

# labels
Y = np.array([[0], [0], [0], [1], [1], [1], [1]])
```

Implementasi Logistic Regression "from the scratch"

Ubah dummy data ke bentuk **tensor** PyTorch & definisikan parameters **W** dan **b**

```
import torch

X_tensor = torch.from_numpy(X).float()
Y_tensor = torch.from_numpy(Y).float()

num_features = X_tensor.size()[1]

# matrix 1 x 6. Mengapa 6? dari mana 6?
W = torch.randn(1, num_features, requires_grad = True)
b = torch.randn(1, requires_grad = True) # bias
```

Implementasi Logistic Regression "from the scratch"

Definisikan model logistic regression & Binary Cross Entropy Loss

```
def sigmoid(z):
  return 1 / (1 + torch.exp(-z))

def logreg(X):
  # mm = matrix multiplication; W.T = W ditranspose
  return sigmoid(torch.mm(X, W.T) + b)

# Binary Cross Entropy Loss
def BCEloss(Y_hat, Y):
    m = Y.size()[0]
    return -(1 / m) * torch.sum(Y * torch.log(Y_hat) + \
                                (1 - Y) * torch.log(1 - Y_hat))
```

Implementasi Logistic Regression "from the scratch"

Gradient Descent Loop - Using **Automatic Differentiation**!

```
EPOCH = 300
ALPHA = 0.05

for i in range(EPOCH):
  preds = logreg(X_tensor)
  loss = BCEloss(preds, Y_tensor)
  loss.backward() # hitung gradients

  # update parameters
  with torch.no_grad():
      W -= W.grad * ALPHA
      b -= b.grad * ALPHA
      W.grad.zero_()
      b.grad.zero_()

  print(f"Epoch {i + 1}/{EPOCH}: Loss: {loss}")
```

Implementasi Logistic Regression "from the scratch"

Gradient Descent Loop – **Manual**!

```
EPOCH = 300
ALPHA = 0.05

m = Y_tensor.size()[0]

for i in range(EPOCH):
  preds = logreg(X_tensor)
  loss = BCEloss(preds, Y_tensor)

  # update parameters
  W_grad = torch.mm((preds - Y_tensor).T, X_tensor)
  b_grad = torch.sum(preds - Y_tensor)
  W -= (1/m) * W_grad * ALPHA
  b -= (1/m) * b_grad * ALPHA

  print(f"Epoch {i + 1}/{EPOCH}: Loss: {loss}")
```

Implementasi Logistic Regression "from the scratch"

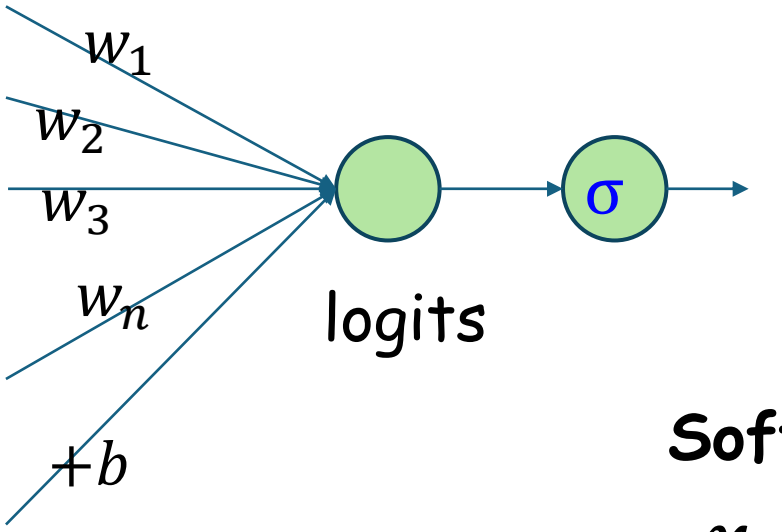
Prediksi terhadap unseen data

```
# kita buat unseen dummy data
X_unseen = np.array([[0.4, 0.8, 0.8, 0, 0, 0],
                     [0, 0, 0, 0.6, 0.8, 0.8]])
X_unseen = torch.from_numpy(X_unseen).float()

# expected label: [0, 1]

Y_pred = logreg(X_unseen)
print(Y_pred)
```

Linear Models
(Multi-class Classification)

Logistic Regression

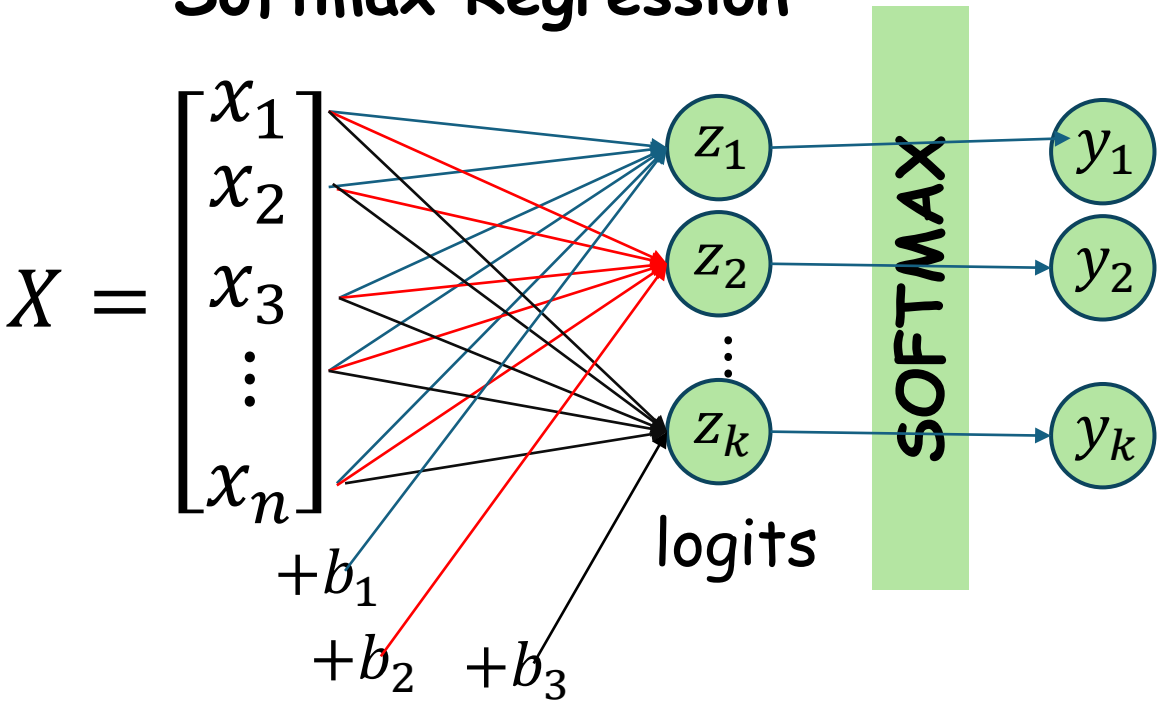
$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$



$$\theta(X) = y^{pred}$$

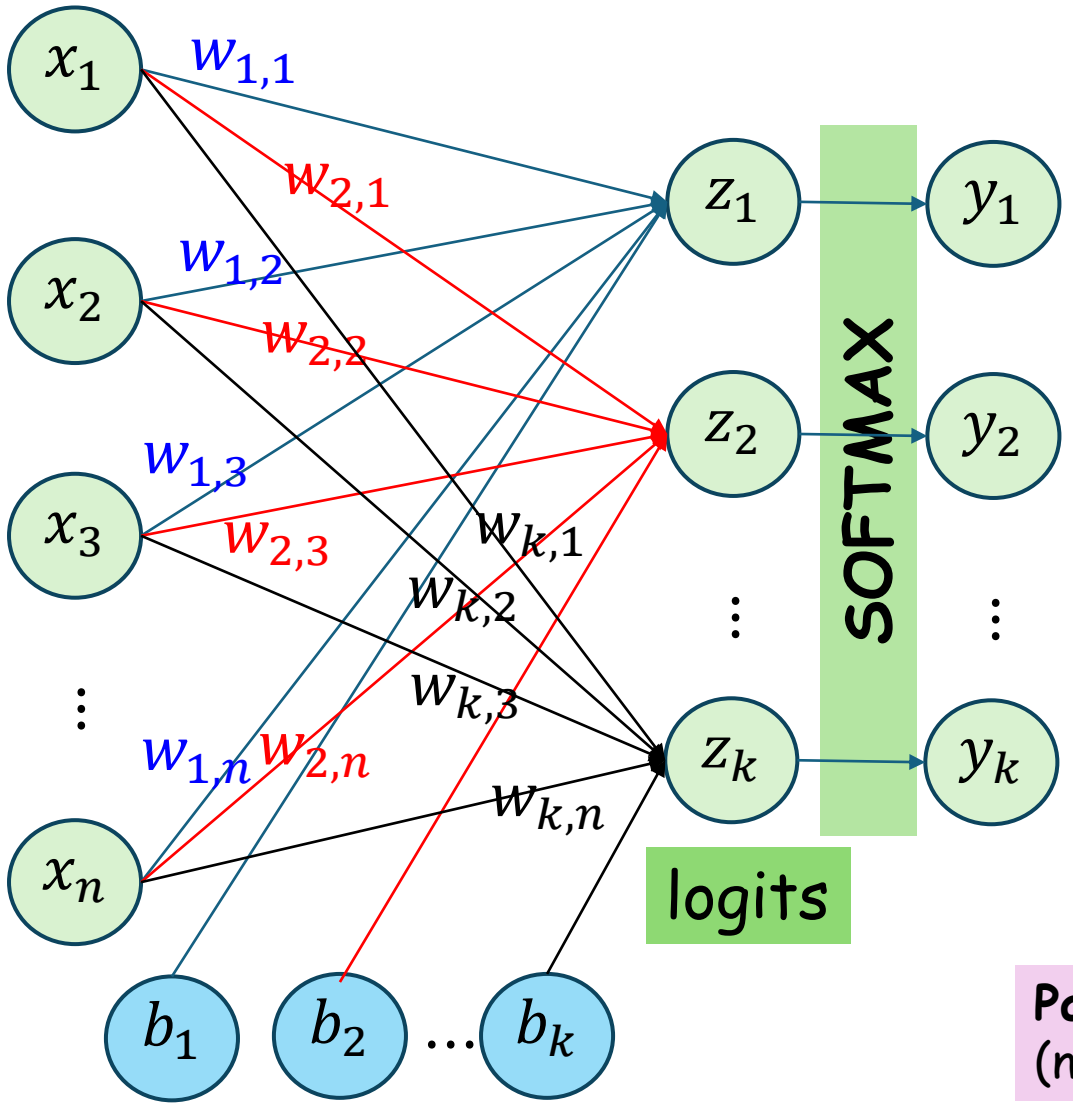
Softmax Regression

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$



$$\sum_i y_i = 1$$

Misal, permasalahan klasifikasi yang terdiri dari **k** kelas. Setiap instance direpresentasikan dengan vektor fitur berdimensi **n**.



$$y_i = softmax(z_i) = \frac{\exp(z_i)}{\exp(z_1) + \exp(z_2) + \cdots + \exp(z_k)}$$

$$= \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

$$\sum_i y_i = 1$$

$$z_i = w_{i,1}x_1 + w_{i,2}x_2 + \cdots + w_{i,n}x_n + b_i$$

$$\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & w_{2,3} & \cdots & w_{2,n} \\ \vdots & \vdots & \vdots & & \vdots \\ w_{k,1} & w_{k,2} & w_{k,3} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix}$$

Parameter W:
(num_classes x num_features)

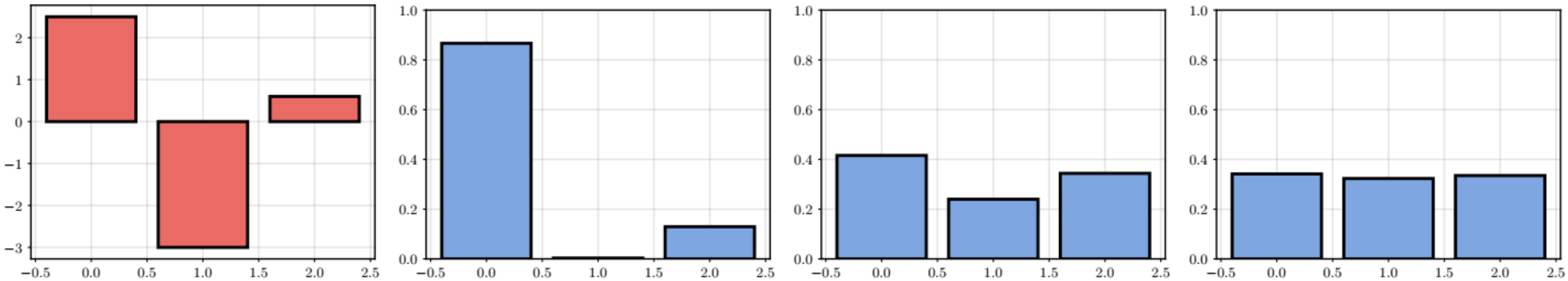
Input X (satu instance):
(num_features x 1)

Definition D.4.4 (Softmax function) *The **softmax** function is defined for a generic vector* $\mathbf{x} \sim (m)$ *as:*

$$[\text{softmax}(\mathbf{x})]_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{E.4.11}$$

Another perspective comes from considering a more general version of the softmax, where we add an additional hyper-parameter $\tau > 0$ called the **temperature**:

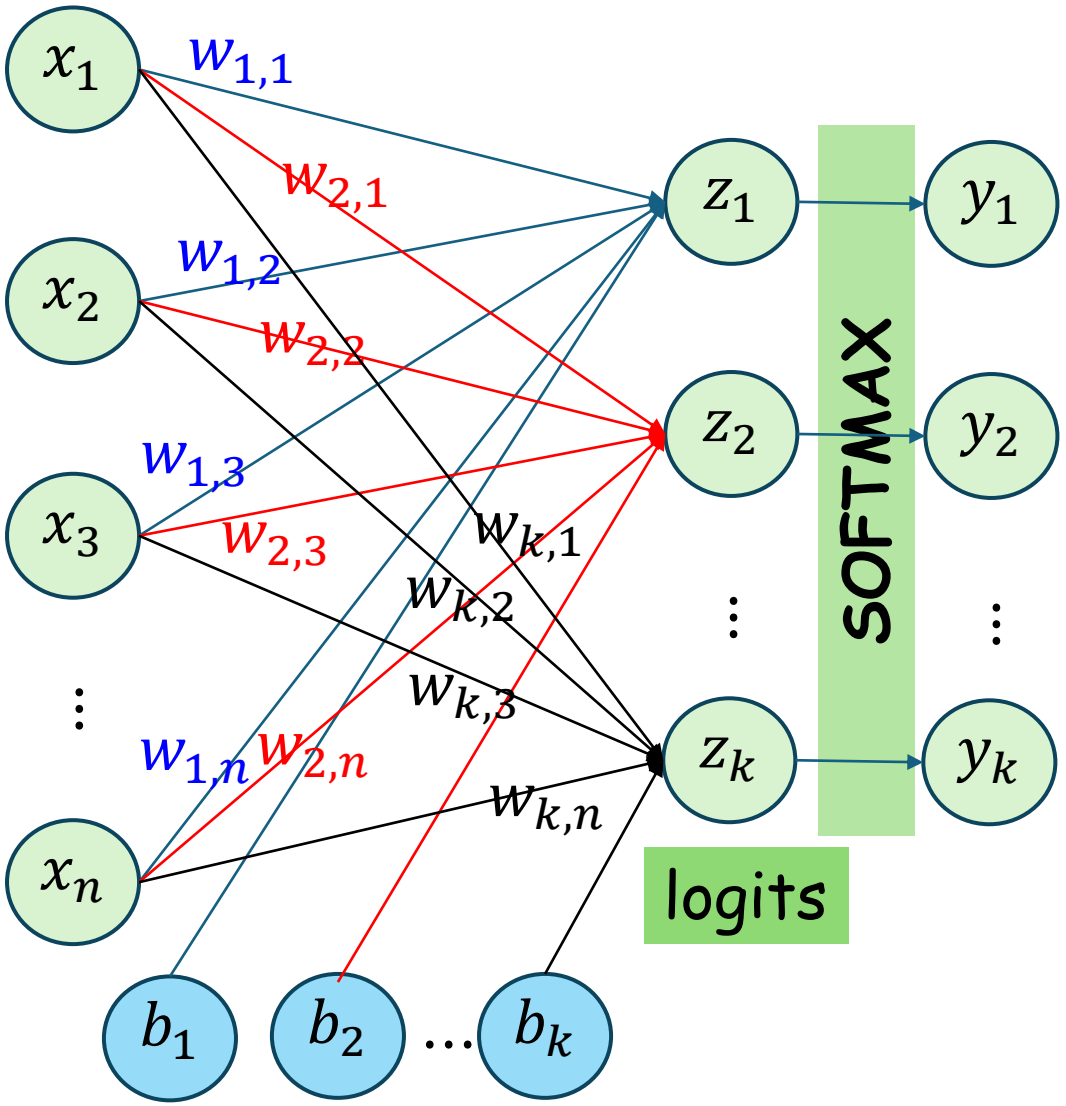
$$\text{softmax}(\mathbf{x}; \tau) = \text{softmax}(\mathbf{x}/\tau)$$



(a) *Inputs* **(b)** $\tau = 1$ **(c)** $\tau = 10$ **(d)** $\tau = 100$

Figure F.4.3: *Example of softmax applied to a three-dimensional vector (a), with temperature set to 1 (b), 10 (c), and 100 (d). As the temperature increases, the output converges to a uniform distribution. Note that inputs can be both positive or negative, but the outputs of the softmax are always constrained in* $[0, 1]$.

Misal, permasalahan klasifikasi yang terdiri dari **k** kelas. Setiap instance direpresentasikan dengan vektor fitur berdimensi



Dengan kata lain:

Bias (parameter):
(num_classes)

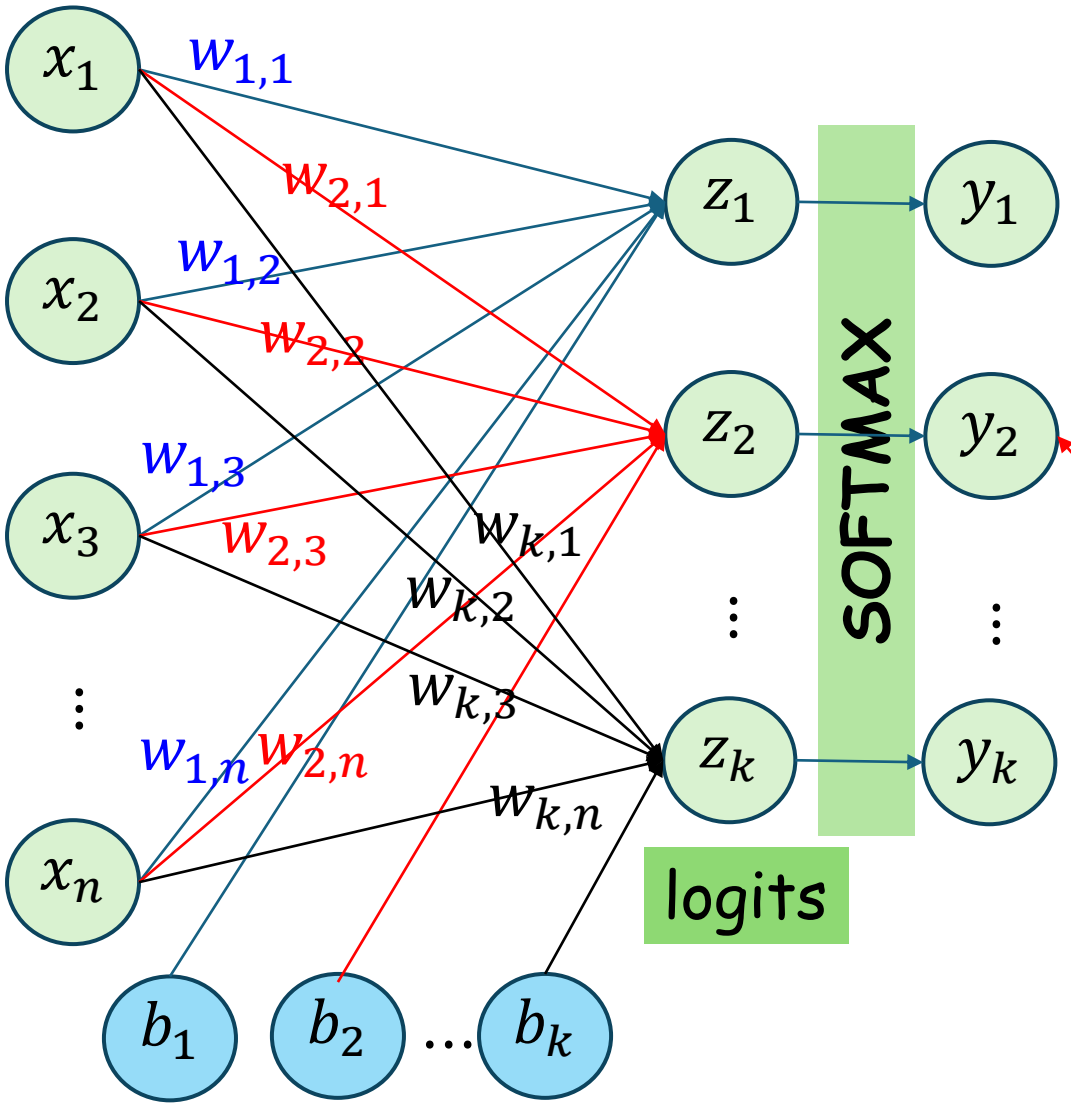
$$\boldsymbol{y} = softmax(\boldsymbol{x}\boldsymbol{W}^{\boldsymbol{T}} + \boldsymbol{b})$$

Parameter W:
(num_classes x num_features)

Input:
(batch_size x num_features)

Output:
(batch_size x num_classes)

Misal, permasalahan klasifikasi yang terdiri dari **k** kelas. Setiap instance direpresentasikan dengan vektor fitur berdimensi **n**.

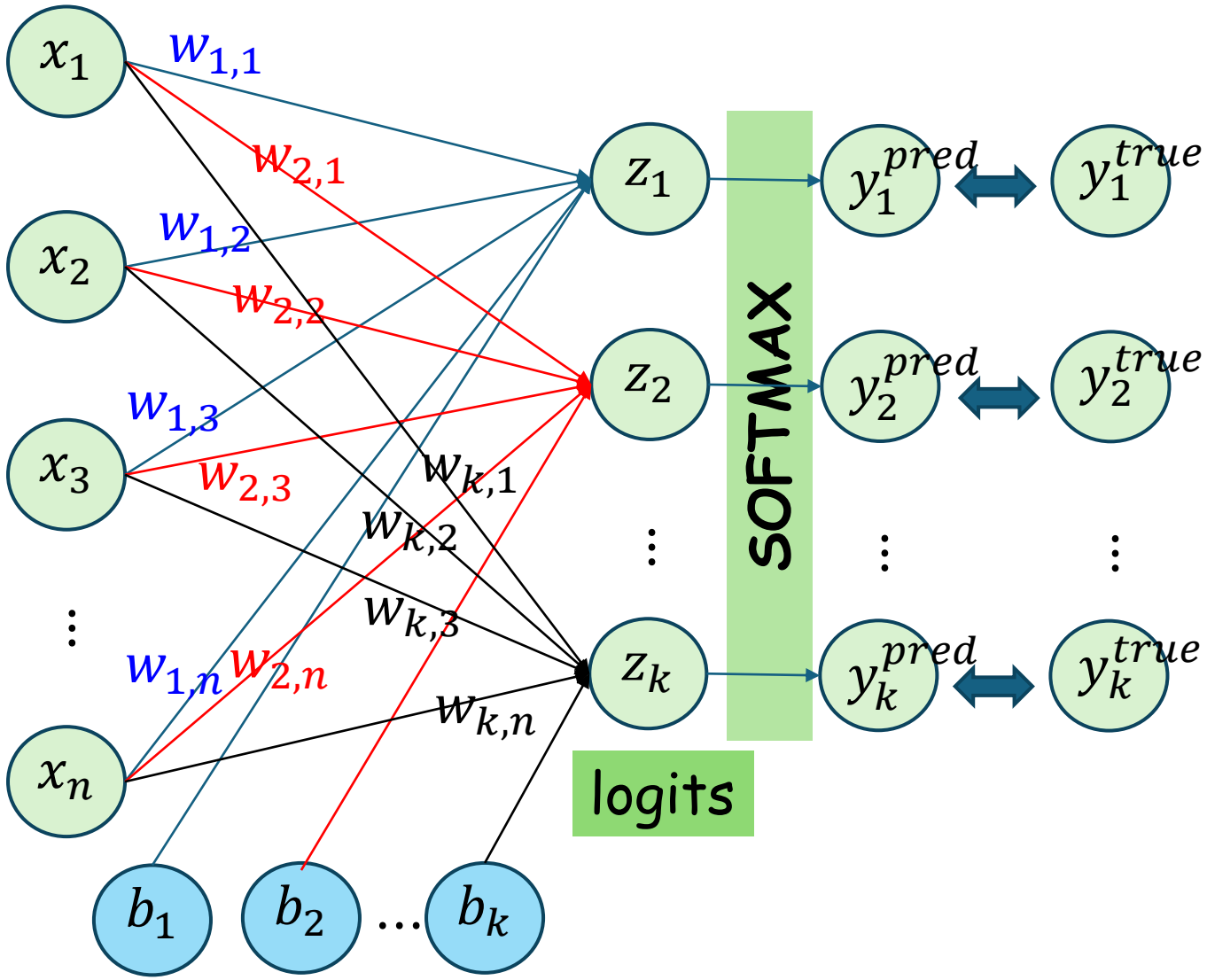


Ingat bahwa output dari softmax menyebabkan:

$$\sum_i y_i = 1$$

Jadi, kita bisa interpretasikan, misal y2, sebagai "probabilitas bahwa input x berasal dari kelas ke-2"

Misal, permasalahan klasifikasi yang terdiri dari **k** kelas. Setiap instance direpresentasikan dengan vektor fitur berdimensi **n**.



Loss function untuk multiclass classification dengan softmax:

Categorical Cross Entropy

Untuk satu instance:

$$L(X, Y^{true}|W, b) = -\sum_{j=1}^{k} y_j^{true} \log(y_j^{pred})$$

Untuk semua instance dalam sebuah batch:

$$L(\boldsymbol{X}, \boldsymbol{Y^{true}}|W, b) = \frac{1}{m}\sum_{i=1}^{m} L(X|W, b)$$

$$\sum_i y_i^{true} = 1 \qquad \sum_i y_i^{pred} = 1$$

By The Way: Stable Version of Softmax

$$softmax(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Rawan overflow atau underflow. Kapan?

$$softmax(z_i) = \frac{\exp(z_i - \max(\boldsymbol{z}))}{\sum_j \exp(z_j - \max(\boldsymbol{z}))}$$

Salah satu trik umum yang dilakukan adalah dengan mengurangi setiap elemen vektor logit dengan elemen terbesar pada vektor logit.

Pertanyaan: Kok Bisa? Apa yang terjadi? Silakan pikirkan untuk bonus ☺

```
X = np.array([[0.4, 0.5, 0.8, 0, 0, 0],
              [0.2, 0.7, 0.4, 0, 0, 0],
              [0.7, 0.2, 0.7, 0, 0, 0],
              [0, 0, 0, 0.6, 0.9, 0.4],
              [0, 0, 0, 0.5, 0.4, 0.2],
              [0, 0, 0, 0.9, 0.8, 0.5],
              [0, 0, 0, 0.8, 0.3, 0.6],
              [0, 0, 0, 0.5, 0, 0],
              [0, 0, 0, 0.9, 0, 0],
              [0, 0, 0, 0.8, 0, 0]])

Y = np.array([[0, 0, 1],
              [0, 0, 1],
              [0, 0, 1],
              [1, 0, 0],
              [1, 0, 0],
              [1, 0, 0],
              [1, 0, 0],
              [0, 1, 0],
              [0, 1, 0],
              [0, 1, 0]])
print(X)
print(Y)
```

Dummy Training Data:
Dummy data + Dummy Label

Representasi 3 buah label atau class:
[1, 0, 0]
[0, 1, 0]
[0, 0, 1]

```
# Ubah dummy data ke abstraksi Torch's Tensor

X_tensor = torch.from_numpy(X).float()
Y_tensor = torch.from_numpy(Y).float()

print(X_tensor.size())
print(Y_tensor.size())



# trainable parameters

batch_size = X_tensor.size()[0]
num_features = X_tensor.size()[1]
num_classes = Y_tensor.size()[1]

W = torch.randn(num_classes, num_features, requires_grad = True)
b = torch.randn(num_classes, requires_grad = True) # bias

print(W)
print(b)
```

```
def softmax(x):
  maxes = torch.max(x, 1, keepdim = True)[0]
  x_exp = torch.exp(x - maxes)
  x_exp_sum = torch.sum(x_exp, 1, keepdim = True)
  return x_exp / x_exp_sum



# Softmax Regression Model
def softmaxreg(X):
    return softmax(torch.mm(X, W.T) + b)

# Categorical Cross Entropy Loss
def CCEloss(Y_pred, Y_true):
    m = Y_pred.size()[0]
    return -(1 / m) * torch.sum(Y_true * torch.log(Y_pred))
```

```
# training loop - gradient descent

EPOCH = 300
ALPHA = 0.09

for i in range(EPOCH):
  preds = softmaxreg(X_tensor)
  loss = CCEloss(preds, Y_tensor)
  loss.backward() # hitung gradients

  # update parameters
  with torch.no_grad():
      W -= W.grad * ALPHA
      b -= b.grad * ALPHA
      W.grad.zero_()
      b.grad.zero_()

  print(f"Epoch {i + 1}/{EPOCH}: Loss: {loss}")
```

Kita pakai **Automatic Differentiation**!
Kita akan belajar hal tersebut!

Fully-Connected Models

The Limitations of Linear Models

Linear models are fundamentally limited, in the sense that by definition they **cannot model non-linear relationships across features**.

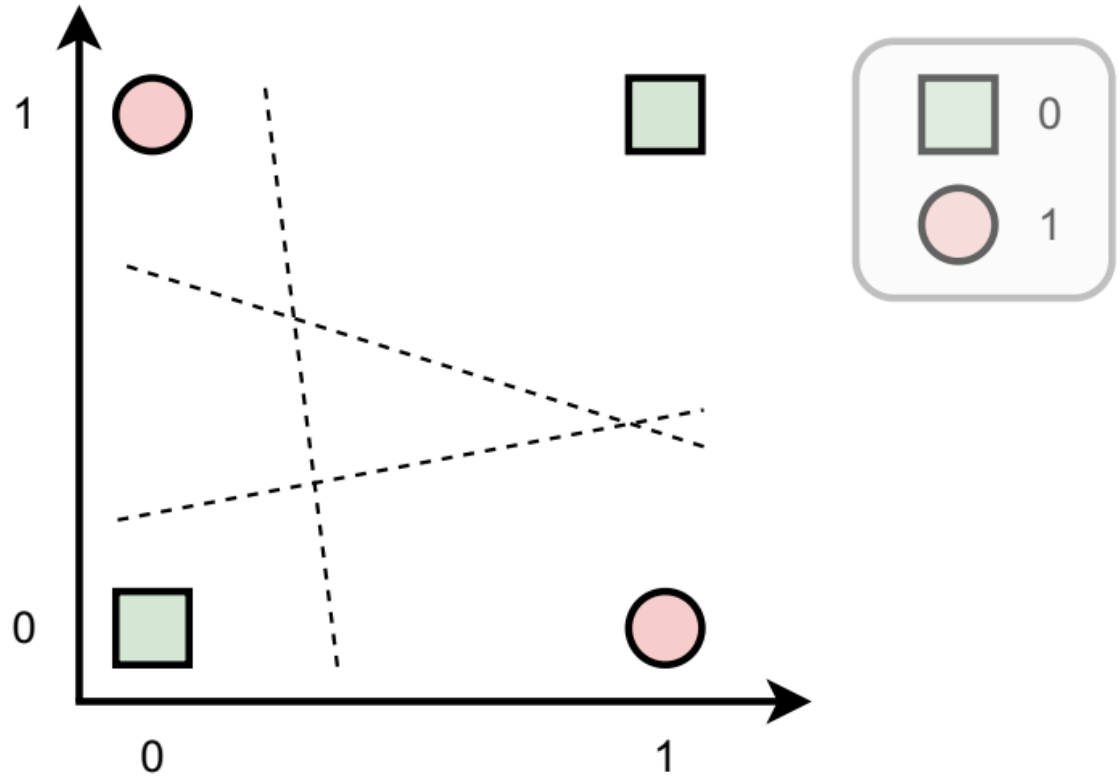


Figure F.5.1: *Illustration of the XOR dataset: green squares are values of one class, red circles are values of another class. No linear model can separate them perfectly (putting all squares on one side and all circles on the other side of the decision boundary). We say that the dataset is not* ***linearly separable****.*

Composition and Hidden Layers

Our differentiable model **f(x)** sometimes can be the composition of many trainable operations:

$$f(\mathbf{x}) = (f_l \circ f_{l-1} \circ \cdots \circ f_2 \circ f_1)(\mathbf{x})$$

For example, in our case the input **x** is a **vector**, hence any vector-to-vector operation (e.g., a matrix multiplication $\mathbf{f_i(x) = Wx}$) can be combined together an endless number of times.

Suppose we chain together **two** different linear projections:

$$\mathbf{h} = f_1(\mathbf{x}) = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$

$$y = f_2(\mathbf{h}) = \mathbf{w}_2^\top \mathbf{h} + b_2$$

Fully-Connected Models

The idea of **fully-connected** (FC) models, also known as **multi-layer perceptrons** (MLPs) for historical reasons, is to insert a simple elementwise non-linearity $\phi : \mathbb{R} \to \mathbb{R}$ in-between projections to avoid the collapse:

Element-wise non-linearity

$$\mathbf{h} = f_1(\mathbf{x}) = \phi\,(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \tag{E.5.3}$$

$$y = f_2(\mathbf{h}) = \mathbf{w}_2^\top \mathbf{h} + b_2 \tag{E.5.4}$$

The second block can be linear, as in (E.5.4), or it can be wrapped into another non-linearity depending on the task (e.g., a softmax function for classification).

Non-linearity ???

The function φ can be any non-linearity, e.g., a polynomial, a square-root, or the sigmoid function.

Choosing non-linearity φ has a strong effect on the gradients of the model and, consequently, on optimization, and the challenge is to select a φ which is to prevent the collapse while staying as close as possible to the identity in its derivative.

Non-Linearity can be called **activation function**. A good choice → **ReLU**

Definition D.5.1 (Rectified linear unit) *The **rectified linear unit** (ReLU) is defined elementwise as:*

$$\mathrm{ReLU}(s) = \max(0, s) \tag{E.5.5}$$

Fully-Connected Models

Definition D.5.2 (Fully-connected layer) *For a batch of n vectors, each of size c, represented as a matrix $\mathbf{X} \sim (n, c)$, a **fully-connected** (FC) layer is defined as:*

$$\mathrm{FC}(\mathbf{X}) = \phi\left(\mathbf{X}\mathbf{W} + \mathbf{b}\right) \tag{E.5.7}$$

The parameters of the layer are the matrix $\mathbf{W} \sim (c', c)$ and the bias vector $\mathbf{b} \sim (c')$, for a total of $(c' + 1)c$ parameters (assuming ϕ does not have parameters). Its hyper-parameters are the width c' and the non-linearity ϕ.

Fully-Connected Models

```
class FullyConnectedLayer(nn.Module):
  def __init__(self, c: int, cprime: int):
    super().__init__()
    # Initialize the parameters
    self.W = nn.Parameter(torch.randn(c, cprime))
    self.b = nn.Parameter(torch.randn(1, cprime))

  def forward(self, x):
    return relu(x @ self.W + self.b)
```

Box C.5.1: *The FC layer in* (E.5.7) *implemented as an object in PyTorch. We require a special syntax to differentiate trainable parameters, such as* **W**, *from other non-trainable tensors: in PyTorch, this is obtained by wrapping the tensors in a* `Parameter` *object. PyTorch also has its collection of layers in* `torch.nn`, *including the FC layer (implemented as* `torch.nn.Linear`*).*

Fully-Connected Models

Then, a model can be defined by chaining together instances of such layers. For example, in PyTorch this can be achieved by the **`Sequential`** object:

```
model = nn.Sequential(
    FullyConnectedLayer(3, 5),
    FullyConnectedLayer(5, 4)
)
```

Fully-Connected Models

Element-wise non-linearity

$$\mathbf{h} = f_1(\mathbf{x}) = \phi\,(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \tag{E.5.3}$$

$$y = f_2(\mathbf{h}) = \mathbf{w}_2^\top \mathbf{h} + b_2 \tag{E.5.4}$$

Theorem 5.1 (Universal approximation of MLPs [Cyb89]) *Given a continuous function $g : \mathbb{R}^d \to \mathbb{R}$, we can always find a model $f(\mathbf{x})$ of the form (E.5.3)-(E.5.4) (an MLP with a single hidden layer) and sigmoid activation functions, such that for any $\varepsilon > 0$:*

$$|f(\mathbf{x}) - g(\mathbf{x})| \leq \varepsilon\ ,\ \forall \mathbf{x}$$

where the result holds over a compact domain. Stated differently, one-hidden-layer MLPs are "dense" in the space of continuous functions.

Batch vs Mini-Batch vs Stochastic Gradient Descent

Batch Gradient Descent biasa memerlukan resource komputasi yang mahal jika ukuran dataset amat sangat besar!

Stochastic Gradient Descent:

```
for t in range(EPOCH):
  for i in range(size(train)):
     x, y_true = train.X[i], train.Y[i]
     θ^(t+1) := θ^t − α.∇L(model(x), y_true)
```

Mini-Batch Gradient Descent:

```
for t in range(EPOCH):
  for X_m, Y_m -> sebuah batch berukuran m di training data:
    θ^(t+1) := θ^t − α.∇L(model(X_m), Y_m)
```

https://colab.research.google.com/drive/1SUy5MKgD51Hv-GBa14pRGiWCoswYGDFc?usp=sharing

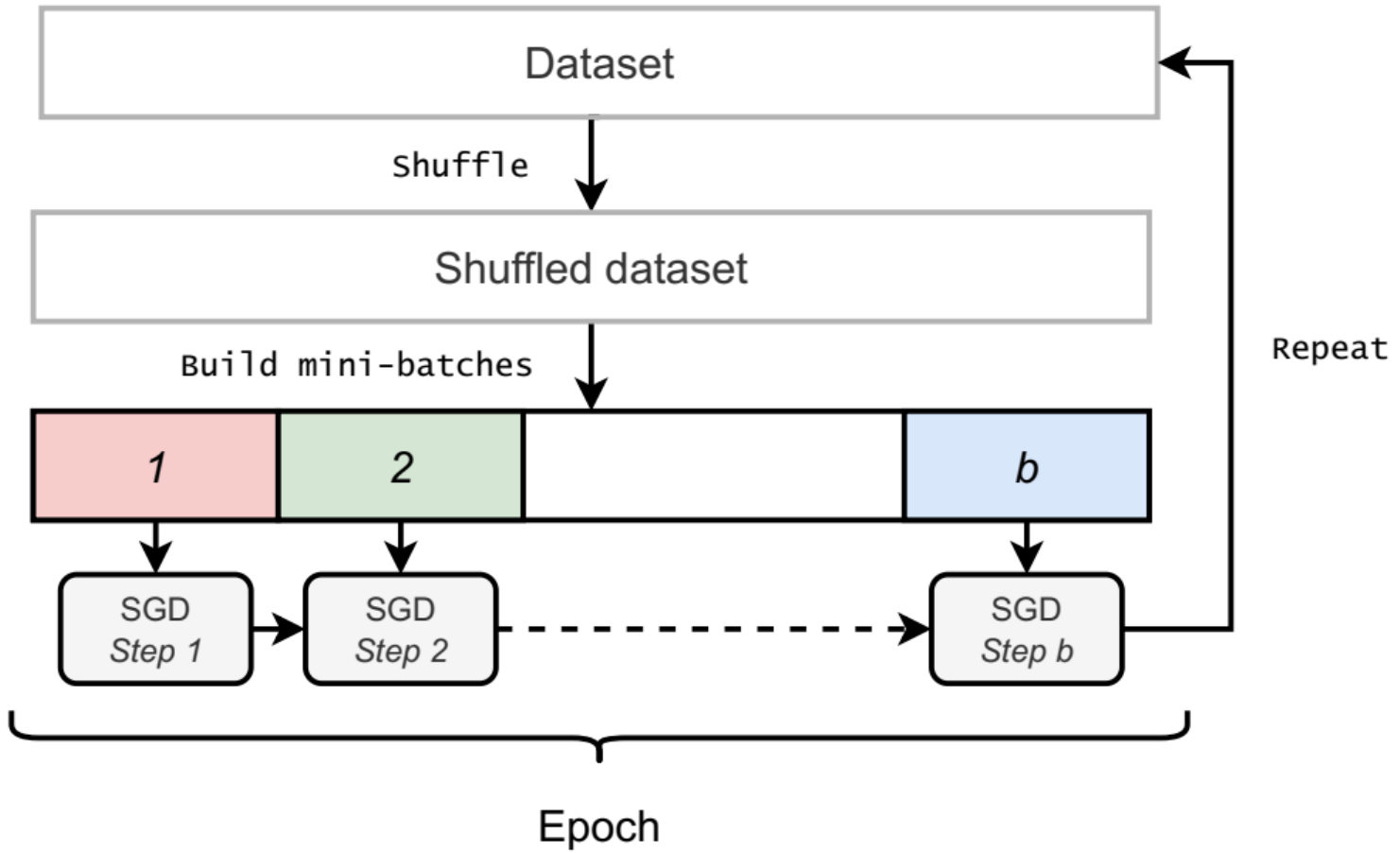
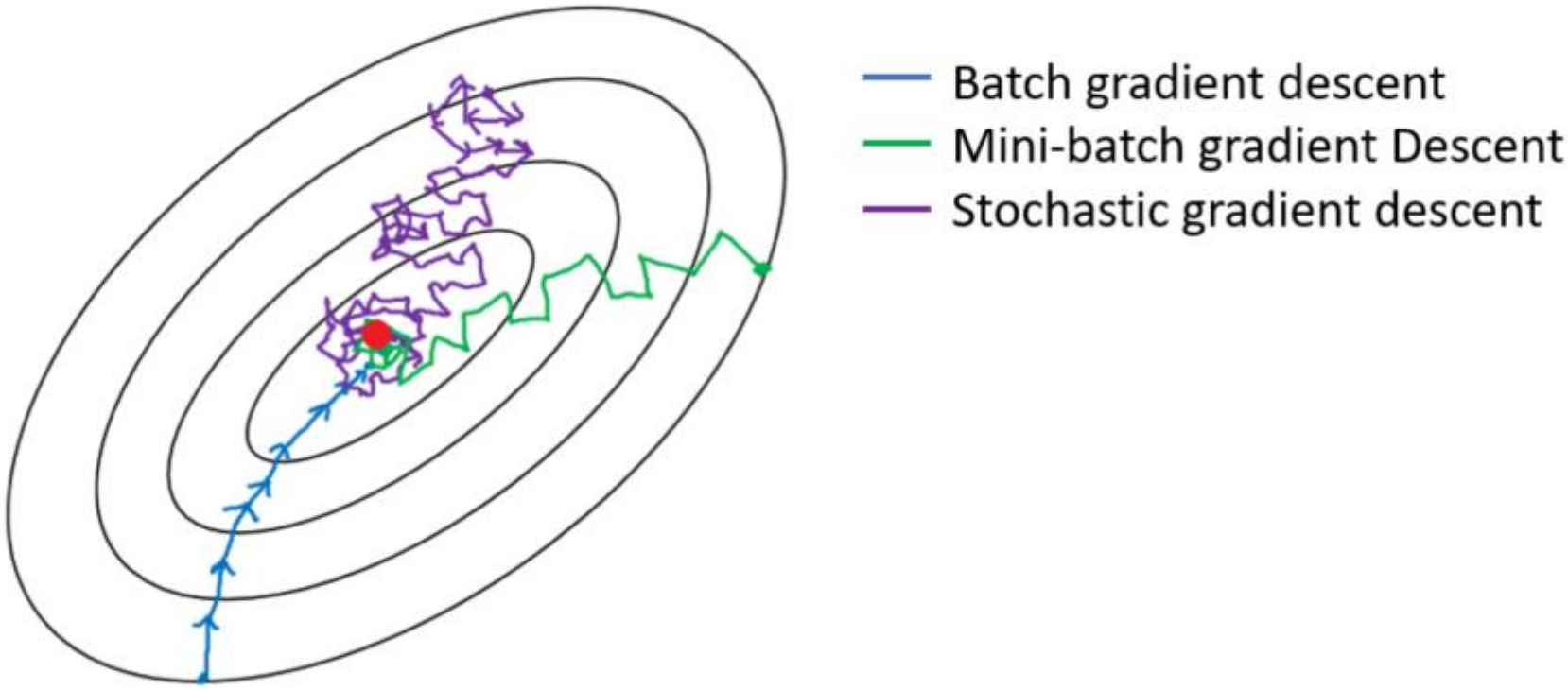


Figure F.5.2: *Building the mini-batch sequence: after shuffling, stochastic optimization starts at mini-batch* 1*, which is composed of the first* r *elements of the dataset. It proceeds in this way to mini-batch* b *(where* $b = \frac{n}{r}$*, assuming the dataset size is perfectly divisible by* r*). After one such epoch, training proceed with mini-batch* $b + 1$*, which is composed of the first* r *elements of the shuffled dataset. The second epoch ends at mini-batch* $2b$*, and so on.*

```
# A dataset composed by two tensors
dataset = torch.utils.data.TensorDataset(
    torch.randn(1000, 3), torch.randn(1000, 1))

# The data loader provides shuffling and mini-batching
dataloader = torch.utils.data.DataLoader(dataset,
                shuffle=True, batch_size=32)

for xb, yb in dataloader:
  # Iterating over the mini-batch sequence (one epoch)
  # xb has shape (32, 3), yb has shape (32, 1)
```

Box C.5.2: *Building the mini-batch sequence with PyTorch's data loader: all frameworks provide similar tools.*

SGD memberikan noise



https://medium.com/analytics-vidhya/gradient-descent-vs-stochastic-gd-vs-mini-batch-sgd-fbd3a2cb4ba4

Solusi 1: Tambahkan "Momentum" / Efek "Heavy Ball"

Update parameter tanpa momentum:

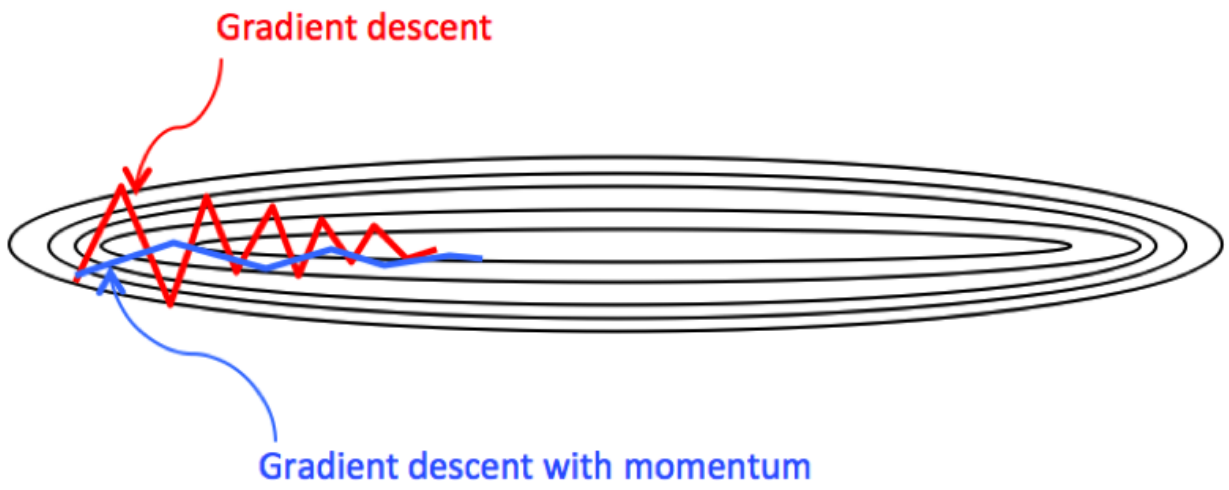
$$\theta_{t+1} \leftarrow \theta_t - \alpha \nabla L(\theta_t)$$

Update parameter **dengan momentum**:

$$m_{t+1} \leftarrow \beta m_t + \alpha \nabla L(\theta_t)$$

$$\theta_{t+1} \leftarrow \theta_t - m_{t+1}$$

Proses update parameter tidak hanya dipengaruhi gradient saat ini, tetapi juga gradient masa lampau.



Seperti memberikan "efek smoothing" karena bola yang berjalan "berat" (heavy ball)

Solusi 2: RMSProp (Geoffrey Hinton)

$$g_t \leftarrow \nabla L(\theta_t)$$

$$v_t \leftarrow \beta . v_{t-1} + (1-\beta) . g_t^2$$

$$\theta_{t+1} \leftarrow \theta_t - \frac{\alpha . g_t}{\sqrt{v_t + \varepsilon}}$$

The core idea behind RMSProp is to keep a **moving average** of the squared gradients to normalize the gradient updates.

Small constant for stability
Ex. 1e-8

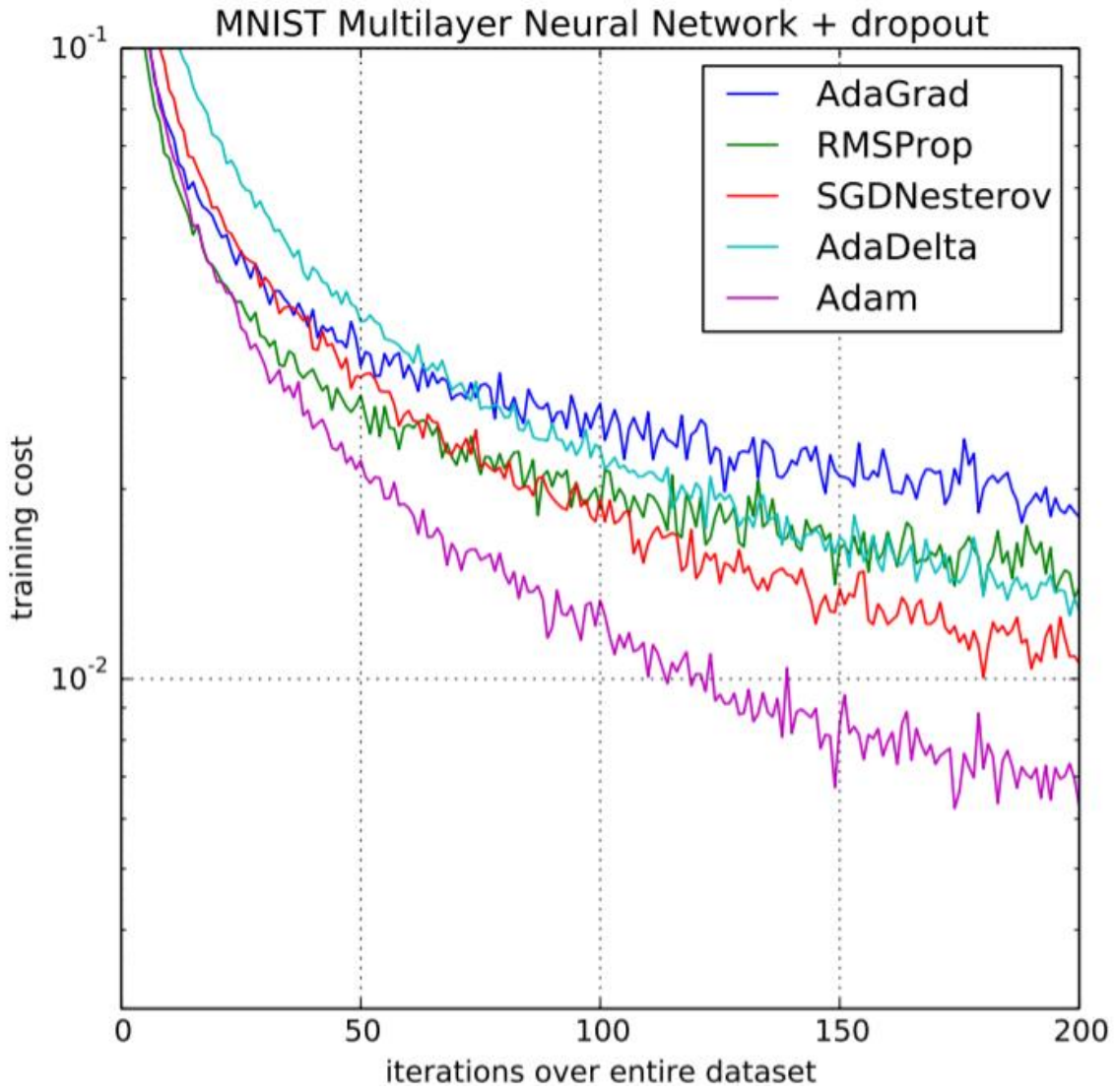
Decay Rate:
Determines how quickly the moving average of squared gradients decays. A common default value is 0.9, which balances the contribution of recent and past gradients.

Adam: RMSProp + Momentum

biasanya $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\alpha = 0.001$

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Adam: RMSProp + Momentum



Adam Optimizer outperforms the rest of the optimizer by a considerable margin in terms of **training cost (low)** and **performance (high)**.