Pertemuan 10 — *Perceptron*, MLP, dan *Backpropagation*

Dari satu neuron ke jaringan berlapis yang belajar sendiri

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Menyambung dari Pertemuan 9
2. Neuron buatan
3. *Perceptron* dan aturan belajarnya
4. Masalah XOR dan *hidden layer*
5. *Multilayer perceptron* (MLP)
6. *Activation function*: sigmoid, tanh, ReLU
7. *Forward pass* dalam bentuk matriks
8. *Backpropagation* sebagai aturan rantai
9. Contoh hitung satu iterasi lengkap
10. *Initialization weight*
11. MLP dari nol dengan NumPy, lalu PyTorch
12. Kesalahan umum dan praktikum

Sasaran pertemuan ini

Anda dapat menjelaskan batas keputusan *perceptron* dan alasan ia gagal pada XOR, menurunkan *backpropagation* dari aturan rantai, menghitung satu langkah maju dan mundur sebuah MLP kecil dengan tangan, lalu menulis MLP sederhana dari nol.

Menyambung dari Pertemuan 9

Pertemuan 9 sudah memberi kita dua alat

Alat 1: *gradient descent*

$$\theta \leftarrow \theta - \eta \, \nabla_\theta L(\theta)$$

Asalkan *gradient loss* terhadap setiap parameter dapat dihitung, model apa pun dapat dilatih dengan cara menuruni permukaan *loss*, satu *mini-batch* demi satu *mini-batch*.

Alat 2: *logistic regression*

$$\hat{p} = \sigma(\mathbf{w}^\top \mathbf{x} + b), \qquad \nabla_{\mathbf{w}} L = (\hat{p} - y)\,\mathbf{x}$$

Pasangan sigmoid dan *binary cross-entropy* memberi *gradient* yang sangat sederhana: *prediction error* dikalikan *input*.

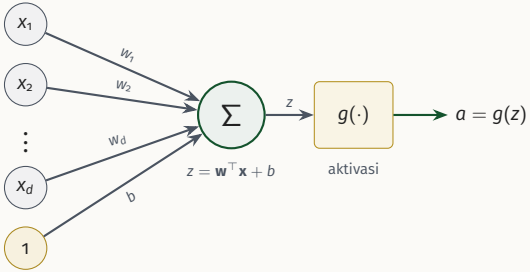
Logistic regression sebenarnya adalah **satu neuron**. Batas keputusannya tetap sebuah garis lurus, seberapa lama pun kita melatihnya.

Pertanyaan hari ini

Apa yang terjadi bila neuron-neuron itu kita tumpuk menjadi beberapa *layer*? Dan bila parameternya menjadi ratusan ribu, bagaimana menghitung *gradient* terhadap **setiap** parameter secara efisien? Jawaban pertanyaan kedua bernama *backpropagation*.

Neuron buatan

Model neuron McCulloch–Pitts



Neuron buatan

Menerima $\mathbf{x}$, menghitung ***pre-activation*** $z = \mathbf{w}^\top \mathbf{x} + b$, lalu mengeluarkan $a = g(z)$. $\mathbf{w}$ adalah ***weight***, b adalah **bias**, dan g adalah ***activation function***. Tanpa bias, batas keputusan selalu melewati titik asal.

Linear regression adalah neuron dengan $g(z) = z$; *logistic regression* adalah neuron dengan $g(z) = \sigma(z)$. Analogi otak cukup sebagai ilham; jaringan modern dirancang menurut apa yang mudah dioptimasi.

Perceptron

Perceptron Rosenblatt (1958)

Model. Neuron dengan aktivasi tangga:

$$\hat{y} = \text{step}(\mathbf{w}^\top \mathbf{x} + b) = \begin{cases} 1 & \mathbf{w}^\top \mathbf{x} + b > 0 \\ 0 & \text{selainnya} \end{cases}$$

Titik dengan $\mathbf{w}^\top \mathbf{x} + b = 0$ membentuk **batas keputusan**: garis di dua dimensi, bidang di tiga dimensi, *hyperplane* di d dimensi. Vektor $\mathbf{w}$ tegak lurus terhadap batas itu dan menunjuk ke sisi kelas 1.

Jadi *perceptron* adalah *classifier* linier. Ia hanya mampu memisahkan kelas yang ***linearly separable***.

Aturan belajar

$$\mathbf{w} \leftarrow \mathbf{w} + \eta\,(y - \hat{y})\,\mathbf{x}, \qquad b \leftarrow b + \eta\,(y - \hat{y})$$

- *Prediction* benar: $y - \hat{y} = 0$, tidak ada perubahan.
- $y = 1$ tetapi $\hat{y} = 0$: z terlalu kecil. Menambah $\eta\mathbf{x}$ menaikkan z untuk *input* itu sebesar $\eta\|\mathbf{x}\|^2$.
- $y = 0$ tetapi $\hat{y} = 1$: *weight* dikurangi $\eta\mathbf{x}$.

Ulangi seluruh data per *epoch*; berhenti bila satu *epoch* penuh berjalan tanpa kesalahan.

Contoh terhitung: melatih *perceptron* untuk AND

Mulai dari $\mathbf{w} = (0, 0)$, $b = 0$, $\eta = 1$; urutan data 00, 01, 10, 11. Kolom $\mathbf{w}, b$ adalah nilai **setelah** langkah itu; tanda $*$ berarti terjadi pembaruan.

Ep.	x	y	z	$\hat{y}$	$\mathbf{w}, b$	Ep.	x	y	z	$\hat{y}$	$\mathbf{w}, b$
1	00	0	0	0	$(0,0), 0$	3	00	0	0	0	$(2,1), 0$
	01	0	0	0	$(0,0), 0$		01	0	1	1	$(2,0), -1^*$
	10	0	0	0	$(0,0), 0$		10	0	1	1	$(1,0), -2^*$
	11	1	0	0	$(1,1), 1^*$		11	1	-1	0	$(2,1), -1^*$
2	00	0	1	1	$(1,1), 0^*$	4	00	0	-1	0	$(2,1), -1$
	01	0	1	1	$(1,0), -1^*$		01	0	0	0	$(2,1), -1$
	10	0	0	0	$(1,0), -1$		10	0	1	1	$(1,1), -2^*$
	11	1	0	0	$(2,1), 0^*$		11	1	0	0	$(2,2), -1^*$
5: hanya 01 salah $\Rightarrow (2,1), -2$						6: tidak ada yang salah, berhenti					

Weight bergoyang beberapa *epoch*, lalu menetap di $\mathbf{w} = (2, 1)$, $b = -2$. Batas keputusannya garis $2x_1 + x_2 = 2$. Periksa sendiri: hanya $(1, 1)$ yang memberi $z = 1 > 0$.

Teorema konvergensi, dan perbandingan dengan *logistic regression*

Novikoff (1962). Bila data *linearly separable*, semua $\|\mathbf{x}_i\| \leq R$, dan ada pemisah dengan *margin* γ, maka jumlah pembaruan paling banyak $(R/\gamma)^2$. Makin lebar *margin*, makin cepat konvergen.

Dua catatan:

- Pemisah yang ditemukan hanyalah salah satu pemisah, dan bergantung pada urutan data. SVM (Pertemuan 7) mencari yang ber-*margin* terbesar.
- Bila data **tidak** *linearly separable*, *perceptron* tidak pernah berhenti; *weight*-nya terus berputar.

	Perceptron	***Logistic regression***
Aktivasi	tangga, 0 atau 1	sigmoid, peluang
Yang diminimalkan	jumlah kesalahan (tak langsung)	*binary cross-entropy*
Pembaruan	hanya saat salah	setiap contoh, sebanding $\hat{p} - y$
Data tak terpisah	tidak konvergen	tetap konvergen
Keyakinan	tidak ada	ada (peluang)
Peran sekarang	sejarah dan intuisi	dasar *output layer*

Ringkasan

Aturan *perceptron* hampir sama dengan *gradient logistic regression* $(\hat{p} - y)\mathbf{x}$. Bedanya, fungsi tangga punya *derivative* nol di mana-mana, sehingga *gradient descent* tidak dapat dipakai. Karena itulah jaringan modern meninggalkan fungsi tangga.

Perceptron dari nol dengan NumPy

```python
import numpy as np

def latih_perceptron(X, y, eta=1.0, max_epoch=20):
    w, b = np.zeros(X.shape[1]), 0.0
    for epoch in range(1, max_epoch + 1):
        salah = 0
        for xi, yi in zip(X, y):
            y_hat = 1 if xi @ w + b > 0 else 0  # tangga
            if y_hat != yi:                  # belajar saat salah
                w += eta * (yi - y_hat) * xi
                b += eta * (yi - y_hat)
                salah += 1
        if salah == 0:
            return w, b, epoch                   # konvergen
    return w, b, None                            # tidak konvergen

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=float)
for nama, y in [("AND", [0, 0, 0, 1]), ("OR", [0, 1, 1, 1]),
                ("XOR", [0, 1, 1, 0])]:
    print(nama, latih_perceptron(X, np.array(y)))
```

***Output* yang Anda peroleh:**

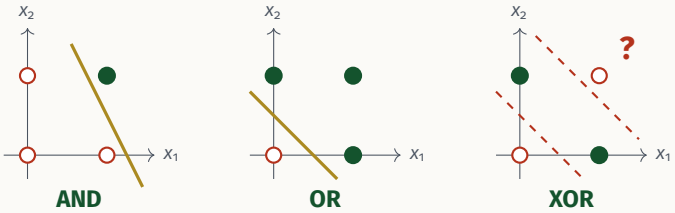
- AND: $\mathbf{w} = (2, 1)$, $b = -2$, berhenti di *epoch* 6, sama dengan tabel tadi.
- OR: $\mathbf{w} = (1, 1)$, $b = 0$, *epoch* 4.
- XOR: `None`. Setelah 20 *epoch* masih ada yang salah, dan akan selalu begitu berapa pun batasnya.

Coba naikkan `max_epoch` menjadi 1000 untuk XOR. Hasilnya tetap `None`.

Sumber: `sample-code-buku, bab12-perceptron-mlp-backpropagation/perceptron_numpy.py`.

Masalah XOR

XOR: tidak ada satu garis pun yang memisahkan



Bukti singkat bahwa XOR tidak *linearly separable*

Andaikan ada w_1, w_2, b dengan aturan "kelas 1 bila $z > 0$". Keempat titik memberi syarat

$$(0,0)\colon b \le 0, \quad (1,1)\colon w_1 + w_2 + b \le 0, \quad (0,1)\colon w_2 + b > 0, \quad (1,0)\colon w_1 + b > 0$$

Jumlah dua syarat terakhir: $w_1 + w_2 + 2b > 0$. Jumlah dua syarat pertama: $w_1 + w_2 + 2b \le 0$. Bertentangan, jadi tidak ada *perceptron* tunggal untuk XOR.

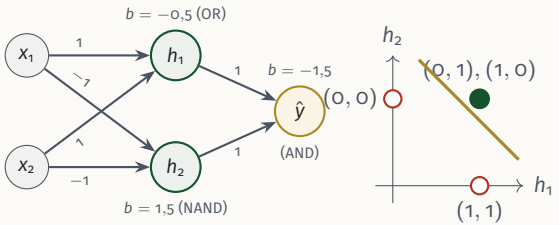
Minsky dan Papert (1969) mempopulerkan keterbatasan ini, dan minat pada *neural network* meredup selama bertahun-tahun.

Jalan keluar: *hidden layer*

XOR $=$ AND$($OR, NAND$)$, dan ketiganya *linearly separable*:

$$h_1 = \text{step}(x_1 + x_2 - 0{,}5) \quad \text{OR}$$
$$h_2 = \text{step}(-x_1 - x_2 + 1{,}5) \quad \text{NAND}$$
$$\hat{y} = \text{step}(h_1 + h_2 - 1{,}5) \quad \text{AND}$$

Periksa $(1,1)$: $h_1 = \text{step}(1{,}5) = 1$, $h_2 = \text{step}(-0{,}5) = 0$, $\hat{y} = \text{step}(-0{,}5) = 0$. Benar.
Periksa $(0,1)$: $h_1 = h_2 = 1$, $\hat{y} = \text{step}(0{,}5) = 1$. Benar.



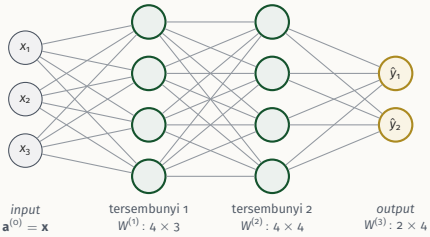
Kanan: posisi keempat *input* di ruang tersembunyi (h_1, h_2). Di sana satu garis sudah cukup memisahkan kedua kelas.

Ringkasan

Hidden layer **memetakan ulang** data ke **representasi** baru, dan masalah yang sulit di ruang asli menjadi mudah di ruang representasi. Pertanyaannya: bagaimana jaringan menemukan *weight* seperti ini sendiri, padahal data tidak memberi "nilai benar" untuk h_1 dan h_2? Inilah *credit assignment problem*.

Multilayer perceptron

Arsitektur MLP dan notasinya



MLP 3–4–4–2. Setiap garis adalah satu *weight*. Jumlah parameternya
$(3 \cdot 4 + 4) + (4 \cdot 4 + 4) + (4 \cdot 2 + 2) = 16 + 20 + 10 = 46$.

Layer $\ell = 0, 1, \ldots, L$; *layer* ℓ berisi n_ℓ neuron.

- $W^{(\ell)}$ berukuran $n_\ell \times n_{\ell-1}$; $w_{jk}^{(\ell)}$ adalah *weight* dari neuron k di *layer* $\ell - 1$ ke neuron j di *layer* ℓ.
- $\mathbf{b}^{(\ell)}$ vektor bias berukuran n_ℓ.
- $\mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}$ dan $\mathbf{a}^{(\ell)} = g^{(\ell)}(\mathbf{z}^{(\ell)})$.

Jumlah parameter *layer* ℓ: $n_\ell n_{\ell-1} + n_\ell$.

MLP 784–256–128–10 untuk MNIST:

$$200.960 + 32.896 + 1.290 = 235.146 \text{ parameter.}$$

Jaringan yang tampak kecil pun sudah punya ratusan ribu *weight*, dan setiap *weight* butuh *gradient*-nya sendiri.

Mengapa aktivasi harus tak linier

Bila semua g adalah identitas, MLP berapa pun dalamnya runtuh menjadi satu *layer* linier:

$$W^{(2)}\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)} = \underbrace{W^{(2)}W^{(1)}}_{W'}\mathbf{x} + \underbrace{W^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)}}_{\mathbf{b}'}$$

Komposisi fungsi linier tetap linier. Jaringan seperti itu sama kuatnya dengan *linear regression* dan tetap gagal pada XOR. Ketaklinieran di setiap *hidden layer* adalah syarat mutlak.

Universal approximation theorem (Cybenko; Hornik dkk., 1989)

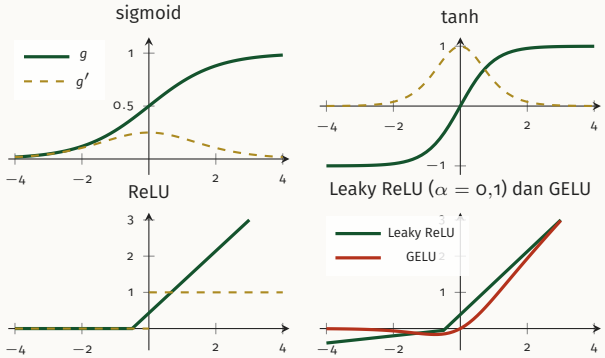
MLP dengan **satu** *hidden layer* dan neuron yang cukup banyak dapat menghampiri fungsi kontinu apa pun pada daerah terbatas, seakurat yang kita mau.

Yang sering disalahpahami

Teorema ini menjamin bahwa *weight* yang baik **ada**. Ia tidak menyebut berapa neuron yang diperlukan (bisa sangat banyak), tidak menjamin *gradient descent* menemukannya, dan tidak menjamin *generalization* ke data baru. Dalam praktik, jaringan yang lebih **dalam** sering jauh lebih hemat daripada jaringan yang hanya lebar. Itu bahan Pertemuan 12.

Activation function

Empat aktivasi dan *derivative*-nya



Garis tebal adalah g, putus-putus adalah g'. Sigmoid dan tanh *saturate*: *derivative*-nya mendekati nol untuk $|z|$ besar. ReLU punya *derivative* tetap 1 di sisi positif. GELU digambar dengan hampiran $z\,\sigma(1{,}702z)$.

Membandingkan aktivasi *hidden layer*

Aktivasi	Rumus	*Derivative* maks.	Catatan dan kapan dipakai
Sigmoid	$1/(1+e^{-z})$	0,25	Jenuh di kedua sisi, tidak berpusat nol; kini di *output* biner dan *gate* RNN
Tanh	$2\sigma(2z)-1$	1	Berpusat nol, lebih cepat dilatih daripada sigmoid, tetap jenuh
ReLU	$\max(0,z)$	1	Murah, tidak jenuh di sisi positif, jarang (*sparse*); bawaan MLP dan CNN sejak 2012
Leaky ReLU	$\max(\alpha z, z)$	1	Kemiringan kecil di sisi negatif cegah neuron mati
GELU	$z\,\Phi(z)$	$\approx 1{,}13$	*Smooth*; bawaan pada *transformer* (Pertemuan 14)

Derivative yang perlu Anda hafal untuk hitungan tangan: $\sigma'(z)=\sigma(z)\,(1-\sigma(z))$ dan $\tanh'(z)=1-\tanh^2 z$. Φ (*phi* kapital) pada GELU adalah CDF normal baku. Keduanya dapat dihitung dari ***output*** neuron, sehingga nilai a yang disimpan saat maju cukup untuk langkah mundur.

Dying ReLU

Bila *pre-activation* neuron ReLU negatif untuk **semua** data (misalnya setelah langkah dengan *learning rate* terlalu besar), *output* dan *gradient*-nya selalu nol, dan neuron itu tidak belajar lagi. Turunkan *learning rate*, pakai *initialization* yang tepat (Pertemuan 11), atau ganti dengan Leaky ReLU atau GELU.

Output layer dan pasangan *loss*-nya

Aktivasi *output* ditentukan oleh jenis tugas. Pasangan yang tepat membuat sinyal *error* di *layer* terakhir berbentuk sederhana $\hat{\mathbf{y}} - \mathbf{y}$, seperti pada *logistic regression*.

$$\text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \qquad L_{\text{CE}} = -\sum_k y_k \log \hat{y}_k \quad (\mathbf{y} \textit{ one-hot})$$

Tugas	**Aktivasi**	***Loss***	**PyTorch**
Regression	identitas	MSE atau Huber	`nn.MSELoss`, `nn.HuberLoss`
Binary classification	sigmoid	*binary cross-entropy*	`nn.BCEWithLogitsLoss`
Multikelas	*softmax*	*categorical cross-entropy*	`nn.CrossEntropyLoss`
Multilabel	sigmoid per label	BCE per label	`nn.BCEWithLogitsLoss`

Biarkan *library* menggabungkan aktivasi dan *loss*

`nn.CrossEntropyLoss` dan `nn.BCEWithLogitsLoss` menerima *logit*. Keduanya menghitung *softmax* atau sigmoid beserta logaritmanya sekaligus dengan trik *log-sum-exp* yang stabil. Menambahkan `nn.Softmax` di akhir model lalu memakai `nn.CrossEntropyLoss` membuat *softmax* terhitung dua kali, dan *training* menjadi lambat tanpa pesan *error* apa pun.

Forward pass

Forward pass: *layer* demi *layer*, dari kiri ke kanan

Algoritma *forward pass*

1. $\mathbf{a}^{(0)} \leftarrow \mathbf{x}$
2. Untuk $\ell = 1, \ldots, L$:
   - $\mathbf{z}^{(\ell)} \leftarrow W^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}$
   - $\mathbf{a}^{(\ell)} \leftarrow g^{(\ell)}(\mathbf{z}^{(\ell)})$
3. Kembalikan $\hat{\mathbf{y}} = \mathbf{a}^{(L)}$ beserta semua $\mathbf{z}^{(\ell)}, \mathbf{a}^{(\ell)}$

Nilai antara **disimpan**, karena akan dibutuhkan lagi saat *backward pass*.

Bentuk *mini-batch*. Susun B contoh sebagai baris $X \in \mathbb{R}^{B \times d}$:

$$Z^{(\ell)} = A^{(\ell-1)}\, W^{(\ell)\top} + \mathbf{1}\,\mathbf{b}^{(\ell)\top}, \qquad A^{(\ell)} = g\big(Z^{(\ell)}\big)$$

$A^{(\ell)}$ berukuran $B \times n_\ell$, dan bias ditambahkan ke setiap baris lewat *broadcasting*. Satu *batch* diproses dengan satu perkalian matriks per *layer*, operasi yang sangat efisien di GPU. Itu sebabnya `nn.Linear` menyimpan *weight* berukuran (*output* $\times$ *input*) dan menghitung $XW^\top + \mathbf{b}$.

Periksa dimensi lebih dahulu

Sebagian besar *error* dalam kode jaringan buatan sendiri adalah *error* dimensi. Tulislah ukuran setiap matriks di samping rumusnya: $\underbrace{Z}_{B\times n_\ell} = \underbrace{A}_{B\times n_{\ell-1}}\ \underbrace{W^\top}_{n_{\ell-1}\times n_\ell}$. Bila ukurannya tidak cocok di atas kertas, kodenya pasti salah.

Backpropagation

Masalahnya: *gradient* terhadap *semua weight*

Cara naif: selisih hingga. Geser satu *weight* sedikit, jalankan *forward pass*, lihat perubahan *loss*. Untuk P parameter diperlukan P kali *forward pass* per langkah. Dengan $P = 235.146$, satu langkah saja sudah ratusan ribu *forward pass*.

Backpropagation menghitung **semua** *gradient* sekaligus dengan *cost* sekitar dua sampai tiga kali satu *forward pass*. Kuncinya:

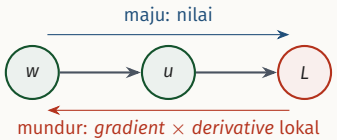
1. *gradient* dapat diuraikan dengan **aturan rantai**;
2. banyak suku dalam uraian itu dipakai bersama oleh banyak *weight*, sehingga cukup dihitung sekali lalu dipakai ulang.

Aturan rantai

Bila L bergantung pada u dan u bergantung pada w:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial u} \cdot \frac{\partial u}{\partial w}$$

Bila w memengaruhi L lewat beberapa jalur, sumbangan semua jalur **dijumlahkan**.



Pada **graf komputasi**, nilai mengalir ke kanan; *gradient* mengalir ke kiri dan di setiap *node* dikalikan dengan *derivative* lokal *node* itu.

Empat persamaan *backpropagation*

Definisikan **sinyal *error*** *layer* ℓ: $\boldsymbol{\delta}^{(\ell)} = \partial L / \partial \mathbf{z}^{(\ell)}$.

Inti *backpropagation*

$$\text{(BP1)} \quad \boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{a}^{(L)}} L \odot g'^{(L)}\left(\mathbf{z}^{(L)}\right)$$

$$\text{(BP2)} \quad \boldsymbol{\delta}^{(\ell)} = \left(W^{(\ell+1)\top} \boldsymbol{\delta}^{(\ell+1)}\right) \odot g'^{(\ell)}\left(\mathbf{z}^{(\ell)}\right)$$

$$\text{(BP3)} \quad \frac{\partial L}{\partial W^{(\ell)}} = \boldsymbol{\delta}^{(\ell)} \, \mathbf{a}^{(\ell-1)\top}$$

$$\text{(BP4)} \quad \frac{\partial L}{\partial \mathbf{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}$$

- **BP1** memulai dari *output*; $\odot$ adalah perkalian elemen demi elemen.
- **BP2** mengirim *error* mundur satu *layer*: disebar balik lewat *weight* yang sama (ditransposkan), lalu dikalikan kemiringan aktivasi.
- **BP3**: *gradient* sebuah *weight* adalah "*error* di ujung tujuan kali aktivasi di ujung asal".
- Untuk sigmoid atau *softmax* dengan *cross-entropy*, BP1 menyusut menjadi $\boldsymbol{\delta}^{(L)} = \hat{\mathbf{y}} - \mathbf{y}$.

Mengapa namanya *backpropagation*

Error dihitung di *output* lalu **dirambatkan mundur** dengan BP2. Setiap $\boldsymbol{\delta}^{(\ell)}$ dihitung sekali, lalu dipakai untuk semua *weight* pada *layer* ℓ dan untuk $\boldsymbol{\delta}^{(\ell-1)}$. Pemakaian ulang itu yang membuatnya murah. Nama umumnya *reverse-mode automatic differentiation*.

Satu iterasi *training*, dalam urutan yang benar

1. **Maju.** Jalankan *forward pass*; simpan semua $\mathbf{z}^{(\ell)}$ dan $\mathbf{a}^{(\ell)}$.
2. ***Error output.*** $\delta^{(L)} \leftarrow \nabla_{\mathbf{a}^{(L)}} L \odot g'^{(L)}(\mathbf{z}^{(L)})$.
3. **Mundur**, untuk $\ell = L, L-1, \ldots, 1$:
   - $\nabla W^{(\ell)} \leftarrow \boldsymbol{\delta}^{(\ell)} \mathbf{a}^{(\ell-1)\top}$ dan $\nabla \mathbf{b}^{(\ell)} \leftarrow \boldsymbol{\delta}^{(\ell)}$
   - bila $\ell > 1$: $\boldsymbol{\delta}^{(\ell-1)} \leftarrow \left(W^{(\ell)\top} \boldsymbol{\delta}^{(\ell)}\right) \odot g'^{(\ell-1)}(\mathbf{z}^{(\ell-1)})$, memakai W **lama**
4. **Perbarui** semua parameter sekaligus: $W^{(\ell)} \leftarrow W^{(\ell)} - \eta \nabla W^{(\ell)}$, $\mathbf{b}^{(\ell)} \leftarrow \mathbf{b}^{(\ell)} - \eta \nabla \mathbf{b}^{(\ell)}$.

Ringkasan

Langkah 4 adalah *gradient descent* Pertemuan 9 apa adanya. Yang baru hari ini hanyalah langkah 2 dan 3: cara memperoleh $\nabla W^{(\ell)}$ untuk *layer* yang tidak terhubung langsung ke *loss*.

Contoh hitung: satu iterasi lengkap

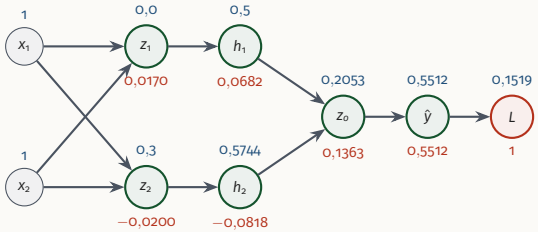
Soal: jaringan 2–2–1 pada satu titik XOR

Sigmoid di *hidden layer* dan *output*. *Training example* $\mathbf{x} = (1, 1)$, target $y = 0$. *Loss* kuadrat $L = \frac{1}{2}(\hat{y} - y)^2$ agar seluruh rantai *derivative*-nya terlihat.

$$W = \begin{bmatrix} 0{,}2 & -0{,}3 \\ 0{,}4 & 0{,}1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 0{,}1 \\ -0{,}2 \end{bmatrix}$$

$$\mathbf{v} = \begin{bmatrix} 0{,}5 \\ -0{,}6 \end{bmatrix}, \quad c = 0{,}3, \quad \eta = 0{,}5$$

W, $\mathbf{b}$ milik *hidden layer*; $\mathbf{v}$, c milik neuron *output*. Baris ke-j dari W adalah *weight* yang masuk ke h_j.



Graf komputasinya. Angka biru di atas *node* adalah nilai maju; angka merah di bawah *node* adalah $\partial L / \partial (node)$ yang dihitung mundur dari L. Empat slide berikut menurunkan setiap angka itu.

Kerjakan bersama dengan kertas dan kalkulator. Inilah latihan terpenting hari ini.

Langkah 1: *forward pass*

Hidden layer

$$z_1 = 0{,}2(1) + (-0{,}3)(1) + 0{,}1 = 0{,}0, \qquad h_1 = \sigma(0) = 0{,}5$$
$$z_2 = 0{,}4(1) + 0{,}1(1) - 0{,}2 = 0{,}3, \qquad h_2 = \sigma(0{,}3) = 0{,}5744$$

Output layer

$$z_o = 0{,}5(0{,}5) + (-0{,}6)(0{,}5744) + 0{,}3 = 0{,}25 - 0{,}3447 + 0{,}3 = 0{,}2053$$
$$\hat{y} = \sigma(0{,}2053) = 0{,}5512$$

Loss

$L = \frac{1}{2}(0{,}5512 - 0)^2 = 0{,}1519$. Jaringan menjawab 0,55 padahal seharusnya 0.

Simpan $z_1, z_2, h_1, h_2, \hat{y}$: semuanya dipakai lagi pada langkah mundur.

Langkah 2: *backward pass* di *output layer*

Sinyal *error output* (BP1)

Mulai dari ujung: $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y = 0{,}5512$.

Derivative sigmoid di *output*:
$\hat{y}(1-\hat{y}) = 0{,}5512 \times 0{,}4488 = 0{,}2474$.

$$\delta_o = \frac{\partial L}{\partial z_o} = 0{,}5512 \times 0{,}2474 = 0{,}1363$$

Gradient parameter *output* (BP3, BP4)

$$\frac{\partial L}{\partial v_1} = \delta_o h_1 = 0{,}1363 \times 0{,}5 = 0{,}0682$$
$$\frac{\partial L}{\partial v_2} = \delta_o h_2 = 0{,}1363 \times 0{,}5744 = 0{,}0783$$
$$\frac{\partial L}{\partial c} = \delta_o = 0{,}1363$$

Baca rantainya dari kanan ke kiri:

$$\frac{\partial L}{\partial v_1} = \underbrace{\frac{\partial L}{\partial \hat{y}}}_{0{,}5512} \cdot \underbrace{\frac{\partial \hat{y}}{\partial z_o}}_{0{,}2474} \cdot \underbrace{\frac{\partial z_o}{\partial v_1}}_{h_1 = 0{,}5} = 0{,}0682$$

Dua faktor pertama adalah δ_o. Faktor itu dipakai ulang untuk v_1, v_2, c, dan untuk *layer* di belakangnya.

Langkah 3: *backward pass* di *hidden layer*

Rambatkan *error* mundur lewat v (BP2)

Gradient terhadap aktivasi tersembunyi:

$$\frac{\partial L}{\partial h_1} = v_1 \delta_o = 0{,}5 \times 0{,}1363 = 0{,}0682, \qquad \frac{\partial L}{\partial h_2} = v_2 \delta_o = -0{,}6 \times 0{,}1363 = -0{,}0818$$

Kalikan dengan *derivative* sigmoid masing-masing, $h_1(1-h_1) = 0{,}25$ dan $h_2(1-h_2) = 0{,}2445$:

$$\delta_1 = 0{,}0682 \times 0{,}25 = 0{,}0170, \qquad \delta_2 = -0{,}0818 \times 0{,}2445 = -0{,}0200$$

Gradient weight tersembunyi (BP3, BP4)

Gradient-nya $\delta_j x_k$. Karena $x_1 = x_2 = 1$:

$$\frac{\partial L}{\partial W} = \begin{bmatrix} 0{,}0170 & 0{,}0170 \\ -0{,}0200 & -0{,}0200 \end{bmatrix}, \qquad \frac{\partial L}{\partial \mathbf{b}} = \begin{bmatrix} 0{,}0170 \\ -0{,}0200 \end{bmatrix}$$

Perhatikan: δ_1, δ_2 tujuh sampai delapan kali lebih kecil daripada δ_o, karena setiap *layer* sigmoid mengalikan *error* dengan paling banyak 0,25. Gejala ini muncul lagi sebagai *vanishing gradient* di Pertemuan 12.

Langkah 4: pembaruan, lalu periksa hasilnya

Setiap parameter dikurangi $0{,}5\times$ *gradient*-nya

$$v_1 = 0{,}5 - 0{,}0341 = 0{,}4659 \qquad v_2 = -0{,}6 - 0{,}0392 = -0{,}6392$$

$$c = 0{,}3 - 0{,}0682 = 0{,}2318$$

$$w_{11} = 0{,}2 - 0{,}0085 = 0{,}1915 \qquad w_{12} = -0{,}3 - 0{,}0085 = -0{,}3085$$

$$w_{21} = 0{,}4 + 0{,}0100 = 0{,}4100 \qquad w_{22} = 0{,}1 + 0{,}0100 = 0{,}1100$$

$$b_1 = 0{,}1 - 0{,}0085 = 0{,}0915 \qquad b_2 = -0{,}2 + 0{,}0100 = -0{,}1900$$

Forward pass ulang

$z_1 = -0{,}0256$, $z_2 = 0{,}3300$

$h_1 = 0{,}4936$, $h_2 = 0{,}5818$

$z_o = 0{,}0900$, $\hat{y} = 0{,}5225$

$L = 0{,}1365$, turun dari $0{,}1519$.

Arah perubahannya masuk akal

Output terlalu tinggi, maka semua yang menaikkannya dikurangi: c turun, v_1 turun, v_2 makin negatif. Neuron h_2 punya *weight* negatif ke *output*, sehingga menaikkan h_2 justru menurunkan $\hat{y}$; karena itu *weight* ke h_2 (w_{21}, w_{22}, b_2) **naik**.

Memeriksa hitungan: kode NumPy dan *gradient checking*

```python
import numpy as np

def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

x = np.array([1.0, 1.0]); y = 0.0
W = np.array([[0.2, -0.3], [0.4, 0.1]])
b = np.array([0.1, -0.2])
v = np.array([0.5, -0.6]); c = 0.3

# forward pass
z = W @ x + b;  h = sigmoid(z)
zo = v @ h + c; y_hat = sigmoid(zo)
L = 0.5 * (y_hat - y) ** 2                    # 0.1519

# backward pass
delta_o = (y_hat - y) * y_hat * (1 - y_hat)  # 0.1363
dv, dc = delta_o * h, delta_o                 # [0.0682 0.0783]
delta_h = delta_o * v * h * (1 - h)           # [0.0170 -0.0200]
dW, db = np.outer(delta_h, x), delta_h
```

Gradient checking dari Pertemuan 9: geser w_{11} sebesar $\pm 10^{-6}$, hitung *loss* dua kali, bagi selisihnya dengan 2×10^{-6}:

$$\frac{L(w_{11} + \epsilon) - L(w_{11} - \epsilon)}{2\epsilon} = 0{,}017043$$

Sama dengan *gradient* analitik $\delta_1 x_1 = 0{,}017043$ sampai enam angka desimal.

Pakai cara ini setiap kali Anda menulis *backward pass* sendiri. Ia lambat, jadi cukup dijalankan pada jaringan kecil sebagai uji kebenaran.

Sumber: `sample-code-buku`, `bab12-perceptron-mlp-backpropagation/backprop_manual.py`.

Autograd: *backpropagation* otomatis di PyTorch

```python
import torch

x = torch.tensor([1.0, 1.0]); y = torch.tensor(0.0)
W = torch.tensor([[0.2, -0.3], [0.4, 0.1]],
                 requires_grad=True)
b = torch.tensor([0.1, -0.2], requires_grad=True)
v = torch.tensor([0.5, -0.6], requires_grad=True)
c = torch.tensor(0.3, requires_grad=True)

h = torch.sigmoid(W @ x + b)      # forward: graf dicatat
y_hat = torch.sigmoid(v @ h + c)
L = 0.5 * (y_hat - y) ** 2
L.backward()                      # aturan rantai otomatis

print(W.grad)  # [[0.0170, 0.0170], [-0.0200, -0.0200]]
print(v.grad)  # [0.0682, 0.0783]
```

PyTorch mencatat graf komputasi selama *forward pass*, lalu menjalankan BP1–BP4 ketika Anda memanggil `.backward()`. Hasilnya sama persis dengan hitungan tangan kita.

Memory cost

Backward pass membutuhkan semua aktivasi dari *forward pass*, sehingga memori saat *training* jauh lebih besar daripada saat inferensi dan tumbuh bersama ukuran *batch* serta kedalaman. Saat inferensi, bungkus kode dengan `torch.no_grad()` agar graf tidak dicatat.

Initialization weight

Mengapa *weight* tidak boleh diinisialisasi nol

Pada *linear regression* dan logistik, $\mathbf{w} = \mathbf{0}$ aman karena *loss*-nya konveks. Pada MLP, itu kesalahan fatal:

1. Semua *weight* tersembunyi nol, maka semua neuron tersembunyi memberi aktivasi yang sama.
2. Menurut BP2, sinyal *error*-nya juga sama, sehingga *gradient* dan pembaruannya sama.
3. Setelah seribu iterasi pun semua neuron tetap kembar. Seratus neuron berperilaku seperti satu neuron.

Masalah ini disebut ***symmetry breaking***: kesamaan harus dipecah sejak awal dengan *weight* **acak**. Bias boleh nol, karena kesimetrian sudah dipecah oleh *weight*.

Skala acaknya penting

- **Terlalu kecil** (simpangan baku 10^{-4}): sinyal mengecil di setiap *layer*, aktivasi *layer* dalam nyaris nol, *gradient* ikut lenyap.
- **Terlalu besar** (simpangan baku 1 dengan 1000 *input*): z sangat besar, sigmoid dan tanh jenuh, *gradient* mendekati nol; dengan ReLU aktivasinya meledak.

Aturan praktis: *variance weight* berbanding terbalik dengan jumlah *input* neuron (*fan-in*), agar *variance* z kira-kira tetap dari *layer* ke *layer*. Resep Xavier dan He diturunkan pada Pertemuan 11. PyTorch sudah menginisialisasi `nn.Linear` menurut *fan-in*.

Implementasi MLP

MLP 2–4–1 dari nol dengan NumPy: menyelesaikan XOR

```
rng = np.random.default_rng(0)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=float)
y = np.array([[0], [1], [1], [0]], dtype=float)
W1 = rng.normal(0, np.sqrt(1 / 2), size=(2, 4))
W2 = rng.normal(0, np.sqrt(1 / 4), size=(4, 1))
b1, b2, eta = np.zeros(4), np.zeros(1), 1.0

for epoch in range(5001):
    # forward pass (baris = contoh)
    A1 = np.tanh(X @ W1 + b1)
    P = sigmoid(A1 @ W2 + b2)
    # backward pass
    dZ2 = (P - y) / len(X)                # sigmoid + BCE
    dW2 = A1.T @ dZ2; db2 = dZ2.sum(0)
    dZ1 = (dZ2 @ W2.T) * (1 - A1 ** 2)    # turunan tanh
    dW1 = X.T @ dZ1;  db1 = dZ1.sum(0)
    # pembaruan
    W1 -= eta * dW1; b1 -= eta * db1
    W2 -= eta * dW2; b2 -= eta * db2

print(P.ravel().round(3))                 # [0. 1. 1. 0.]
```

- Bentuk *mini-batch*: setiap baris X satu contoh, sehingga *weight* ditulis $d \times n$ dan maju menjadi `X @ W1`.
- *Output* sigmoid dengan BCE, sehingga sinyal *error output* cukup $P - Y$ (BP1 yang menyusut). Dibagi B karena *loss* dirata-rata.
- `dZ1` adalah BP2; `A1.T @ dZ2` adalah BP3 untuk satu *batch* sekaligus.
- *Initialization* acak berskala $\sqrt{1/fan\text{-}in}$.

Setelah 5000 *epoch* *loss* sekitar 0,0003 dan keempat *prediction* tepat.

Sumber: `sample-code-buku`, `bab12-perceptron-mlp-backpropagation/mlp_xor_numpy.py`.

Dengan PyTorch: MLP 784–256–128–10 untuk MNIST

```
tf = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])
latih = datasets.MNIST("data", train=True, download=True,
                       transform=tf)
dl_latih = DataLoader(latih, batch_size=128, shuffle=True)
model = nn.Sequential(
    nn.Flatten(),                       # 28x28 -> 784
    nn.Linear(784, 256), nn.ReLU(),
    nn.Linear(256, 128), nn.ReLU(),
    nn.Linear(128, 10),                 # logit, tanpa softmax
)
loss_fn = nn.CrossEntropyLoss()         # softmax + CE
opt = torch.optim.SGD(model.parameters(), lr=0.1)

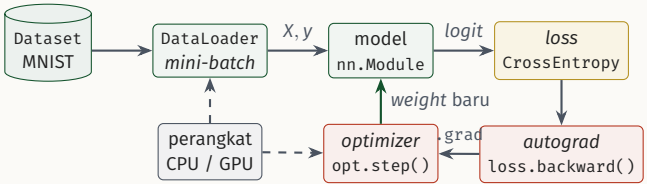
for epoch in range(5):
    for xb, yb in dl_latih:
        loss = loss_fn(model(xb), yb)  # forward
        opt.zero_grad()                # nolkan gradien lama
        loss.backward()                # backpropagation
        opt.step()                     # gradient descent
```

MNIST: 70.000 citra angka 28×28 (60.000 latih, 10.000 uji), sepuluh kelas; setiap citra diratakan menjadi 784 nilai.
Pola empat baris di dalam *loop* akan Anda tulis berulang kali sampai akhir semester: ***forward*, `zero_grad`, `backward`, `step`**.
Test accuracy biasanya 97–98% dalam beberapa *epoch*.

Lupa `zero_grad()`

PyTorch **menjumlahkan** *gradient* baru ke `.grad` yang lama. Tanpa `zero_grad()` *gradient* menumpuk dan *training* kacau.

Komponen *training* dan alat yang lazim



Komponen	Pilihan utama	Alasan dan alternatif
Framework	PyTorch 2.x	*autograd* dinamis, mudah di-*debug*; alternatif JAX/Flax, Keras 3
Data	`DataLoader`, `torchvision`	*mini-batch*, pengacakan, pemuatan paralel
Pemahaman konsep	NumPy dari nol	setiap *gradient* terlihat; cocok untuk praktikum hari ini
Data tabular kecil	`MLPClassifier` scikit-learn	cepat dicoba tanpa GPU

Kesalahan umum

Kesalahan yang paling sering muncul

Saat menghitung dengan tangan

1. Memperbarui **v** lebih dahulu, lalu memakai **v** yang **baru** untuk $\partial L/\partial h$. Semua *gradient* dihitung dengan parameter lama; pembaruan dilakukan setelah *backward pass* selesai.
2. Lupa mengalikan dengan *derivative* aktivasi, atau menulis $\sigma'(z) = \sigma(z)$.
3. Memakai z alih-alih a sebagai "aktivasi di ujung asal" pada BP3.
4. Lupa bahwa bias juga punya *gradient* (BP4).

Saat menulis kode

1. Menginisialisasi semua *weight* nol, atau dengan nilai yang sama.
2. Tidak ada aktivasi tak linier di antara dua `nn.Linear`: modelnya tetap linier.
3. `nn.Softmax` sebelum `nn.CrossEntropyLoss`.
4. Lupa `opt.zero_grad()`.
5. W dan $W^\top$ tertukar tetapi lolos karena *broadcasting*; periksa `.shape` di setiap *layer*.
6. Menyimpulkan model gagal karena satu *seed*; XOR dengan dua neuron tersembunyi kadang macet di minimum lokal.

Ringkasan

Hampir semua kesalahan *backpropagation* buatan sendiri tertangkap oleh dua kebiasaan: menulis ukuran matriks di samping rumus, dan *gradient checking* pada jaringan kecil.

Praktikum

Praktikum Pertemuan 10

Bagian A — satu iterasi *backpropagation* dengan tangan

1. Pakai jaringan 2–2–1 dan parameter awal yang sama seperti contoh hari ini, tetapi dengan *input* $\mathbf{x} = (1, 0)$ dan target $y = 1$.
2. Tuliskan *forward pass*, δ_o, δ_1, δ_2, seluruh *gradient*, dan parameter baru dengan empat angka desimal.
3. *Weight* mana yang *gradient*-nya nol? Jelaskan mengapa.
4. Ulangi dengan *binary cross-entropy*; tunjukkan $\delta_o = \hat{y} - y$ dan bandingkan besarnya dengan *loss* kuadrat.
5. Periksa semua angka Anda dengan kode NumPy dan `.backward()` PyTorch.

Bagian B — MLP dari nol

1. Tulis kelas `MLP` NumPy dengan metode `forward`, `backward`, dan `step`, untuk jumlah *layer* sembarang.
2. Uji pada XOR dengan 4 neuron tersembunyi, lalu dengan 2 neuron untuk sepuluh *seed*. Berapa kali berhasil?
3. Tambahkan *gradient checking* untuk semua parameter; selisih relatif harus di bawah 10^{-6}.
4. Latih pada `make_moons` scikit-learn dan gambarkan batas keputusannya.
5. Bandingkan hasilnya dengan MLP PyTorch berarsitektur sama.

Dikumpulkan: foto atau PDF hitungan tangan dan satu *notebook*. Penilaian: ketepatan perhitungan dan kebenaran implementasi (tugas individu).

Rekap

Rekap pertemuan ini

- Neuron buatan menghitung $g(\mathbf{w}^\top \mathbf{x} + b)$. *Linear regression* dan logistik adalah neuron tunggal dengan aktivasi identitas dan sigmoid.
- *Perceptron*: aktivasi tangga dan aturan $\mathbf{w} \leftarrow \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$. Pasti konvergen bila data *linearly separable*; tidak pernah konvergen bila tidak.
- XOR tidak *linearly separable*. *Hidden layer* memetakan data ke representasi yang dapat dipisahkan secara linier.
- Tanpa aktivasi tak linier, MLP runtuh menjadi model linier. ReLU adalah bawaan *layer* tersembunyi; sigmoid dan *softmax* di *output*, berpasangan dengan *cross-entropy*.
- *Backpropagation* adalah aturan rantai dari *output* ke *input*: $\boldsymbol{\delta}^{(\ell)} = (W^{(\ell+1)\top}\boldsymbol{\delta}^{(\ell+1)}) \odot g'(\mathbf{z}^{(\ell)})$, $\partial L/\partial W^{(\ell)} = \boldsymbol{\delta}^{(\ell)}\mathbf{a}^{(\ell-1)\top}$, dengan *cost* sebanding satu *forward pass*.
- Contoh hari ini: satu iterasi menurunkan *loss* dari 0,1519 ke 0,1365.
- *Weight* diinisialisasi acak dengan skala menurut *fan-in*. PyTorch menjalankan *backpropagation* lewat *autograd* dengan pola *forward*, `zero_grad`, `backward`, `step`.

Yang perlu Anda siapkan

1. Baca buku ajar Bab 12, Mitchell Bab 4, dan Prince (*Understanding Deep Learning*) Bab 3, 4, dan 7.
2. Latihan tangan: latih *perceptron* untuk OR mulai dari $\mathbf{w} = (0, 0)$, $b = 0$, $\eta = 1$, dan tuliskan jejaknya seperti tabel AND.
3. Hitung jumlah parameter MLP 3072–512–256–10 untuk CIFAR-10. Berapa persen di *layer* pertama?
4. Jalankan kode MNIST dengan aktivasi sigmoid, tanh, dan ReLU; catat *test accuracy* setelah lima *epoch* dalam satu tabel.

Pertemuan depan

Kita sudah bisa melatih MLP. Pertemuan 11 membahas cara melatihnya **dengan stabil**: *initialization* Xavier dan He, *optimizer* momentum, RMSProp, dan Adam, *batch normalization*, *dropout*, *early stopping*, serta cara membaca kurva *loss* untuk mendiagnosis *training* yang bermasalah.

Terima kasih

Pertanyaan?