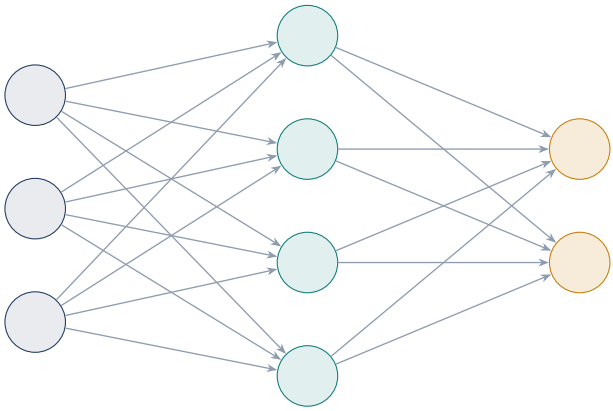
JARINGAN SYARAF TIRUAN

Feedforward Satu *Hidden Layer*

Satu Model, Tiga Wajah

Matematika, NumPy, dan TensorFlow



Dr. Mohammad Jamhuri

Penerbit Ediide Infografika

2026

JARINGAN SYARAF TIRUAN

Feedforward Satu Hidden Layer Satu Model, Tiga Wajah:
Matematika, NumPy, dan TensorFlow
Copyright © Mohammad Jamhuri, 2026
xi + 109 hlm, 16 x 24 cm

Penulis : Dr. Mohammad Jamhuri
Editor : Juhari, M.Si
Layout Isi/Cover : @aba_tara

Cetakan Pertama : Juli 2026
ISBN : 978-623-5934-98-3

Diterbitkan pertama kali oleh:
Penerbit Ediide Infografika
Jl. Joyo Agung III, Blok J-21 Villa Tlogomas, Malang
Email: penerbit.ediide@gmail.com
Website: penerbitan.ediide.com

Anggota IKAPI Jawa Timur
No. 242/JTI/2020

All Rights Reserved © 2026

Hak Cipta Dilindungi Oleh Undang-undang.
Buku ini dilisensikan di bawah **Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0)**.

Anda bebas menyalin, mendistribusikan ulang, dan mengadaptasi materi ini dalam media atau format apa pun untuk tujuan *nonkomersial*, selama mencantumkan atribusi yang sesuai kepada penulis asli. Penggunaan komersial memerlukan izin tertulis dari pihak penerbit maupun penulis.

Informasi lisensi: https://creativecommons.org/licenses/by-nc/4.0/

Kata Pengantar

Ada banyak sekali buku tentang *neural network*. Sebagian besar memilih salah satu dari dua jalan. Jalan pertama: bercerita panjang lebar tentang ide besar dan analogi otak, lalu menutupnya dengan satu-dua rumus yang muncul begitu saja tanpa pernah benar-benar diturunkan. Jalan kedua: melompat langsung ke `model.fit()` sehingga pembaca bisa melatih sebuah jaringan dalam lima menit, tetapi tidak pernah tahu apa yang sebenarnya terjadi di balik satu baris itu.

Buku ini sengaja mengambil jalan yang lebih sempit, dan menurut saya lebih memuaskan: kita akan membuka kap mesinnya. Kita akan menurunkan *backpropagation* dari nol, suku demi suku, sampai tidak ada lagi rumus yang turun dari langit. Setelah memahaminya, kita membangun jaringannya sendiri dengan NumPy—tanpa *framework*, tanpa sihir—supaya setiap perkalian matriks dan setiap pembaruan bobot terlihat di depan mata. Barulah setelah itu kita membandingkannya dengan versi TensorFlow, dan Anda akan tersenyum karena tahu persis apa yang dikerjakan oleh kode yang ringkas itu.

Agar pembahasannya tetap dalam tanpa membuat kita tersesat, ruang lingkupnya sengaja dipersempit ke satu arsitektur saja: **jaringan feedforward dengan satu *hidden layer***. Pilihan ini bukan karena arsitektur yang lebih besar tidak penting, melainkan karena satu *hidden layer* sudah cukup untuk memuat hampir seluruh ide inti—*forward pass*, fungsi aktivasi, fungsi *loss*, gradien, *backpropagation*, dan optimisasi—tanpa beban notasi berlapis-lapis. Begitu Anda menguasainya untuk satu lapisan, menambah lapisan berikutnya hanyalah pengulangan pola yang sama.

Benang merah yang akan kita ikuti dari awal sampai akhir sederhana namun jarang ditonjolkan: *ini model yang sama, hanya ujungnya yang berganti*. Jaringan yang persis sama kita pakai untuk tiga tugas yang kelihatannya berbeda— klasifikasi, regresi, dan *time series forecasting*—dan satu-satunya yang betul-betul berubah adalah lapisan keluaran beserta fungsi *loss*-nya. Untuk setiap tugas kita memakai data publik dari Kaggle yang berukuran kecil, supaya Anda bisa mengunduh, menjalankan, dan memverifikasi hasilnya sendiri di laptop.

Buku ini ditujukan untuk pembaca yang penasaran dan tidak takut

pada rumus, tetapi bukan untuk ahli matematika. Saya berasumsi Anda pernah berkenalan dengan kalkulus dan aljabar linear di tahun-tahun awal kuliah, meski mungkin sudah lama tidak menyentuhnya. Karena itu, Bab 2 menyediakan penyegaran secukupnya: hanya bekal yang benar-benar kita pakai, tidak lebih.

Penulis menyampaikan terima kasih kepada LP2M dan LITAPDIMAS Tahun 2026 atas dukungan pendanaan melalui skema Hibah Penelitian Interdisipliner. Dukungan ini memberikan ruang yang sangat berarti bagi penulis untuk menyusun, merapikan, dan mengembangkan bahan ajar ini agar dapat digunakan secara lebih luas dalam pembelajaran dan pengembangan keilmuan. Semoga buku ini menjadi salah satu luaran yang bermanfaat dari kegiatan penelitian tersebut.

Selamat membaca, dan selamat membongkar mesinnya.

Malang, Juni 2026

Penulis

Daftar Isi

Daftar Gambar

Daftar Tabel

1

Pengantar

Bayangkan Anda menunjukkan foto seekor burung kepada seorang anak kecil yang belum pernah melihat burung. Anda tidak memberinya daftar aturan—"kalau ada paruh dan sayap dan dua kaki, itu burung". Anda cukup menunjukkan banyak contoh, dan lama-kelamaan ia bisa menebak sendiri, bahkan untuk burung yang belum pernah ia lihat. Tidak ada aturan eksplisit yang ditulis; yang ada hanyalah pola yang perlahan terbentuk dari contoh-contoh.

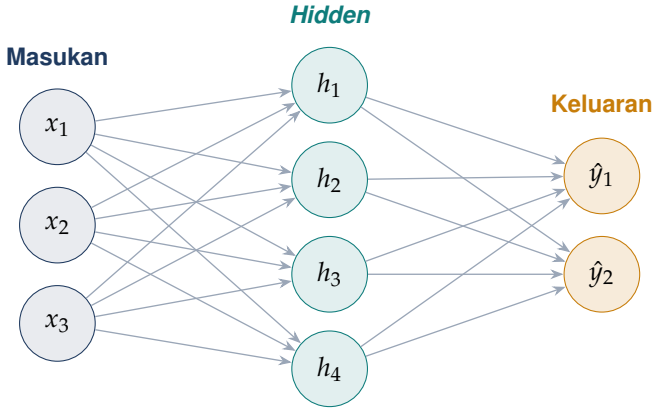
Neural network bekerja dengan semangat yang sama. Kita tidak memberitahunya aturan. Kita memberinya contoh—pasangan masukan dan jawaban yang benar—lalu membiarkannya menyetel ribuan angka kecil di dalam dirinya sampai tebakannya cocok dengan jawaban yang kita inginkan. Angka-angka kecil itu disebut *bobot* (*weights*), dan seluruh isi buku ini, pada hakikatnya, adalah cerita tentang bagaimana angka-angka itu disetel.

1.1 Apa itu *neural network*?

Secara teknis, sebuah *neural network* hanyalah sebuah fungsi matematika. Ia menerima sekumpulan angka sebagai masukan, melakukan serangkaian perkalian dan penjumlahan, menyelipkan beberapa fungsi tak-linear di antaranya, lalu mengeluarkan sekumpulan angka sebagai keluaran. Itu saja. Tidak ada otak, tidak ada kesadaran—yang ada hanya aritmetika yang disusun dengan rapi.

Yang membuatnya menarik adalah *susunan*-nya. Angka-angka masukan ditempatkan pada sekelompok titik yang disebut *neuron* pada lapisan masukan (*input layer*). Nilai-nilai itu kemudian mengalir melalui sambungan-sambungan berbobot menuju lapisan tersembunyi (*hidden layer*), diolah, lalu diteruskan ke lapisan keluaran (*output layer*) yang memberikan jawaban

akhir. Gambar 1.1 memperlihatkan susunan yang akan menemani kita sepanjang buku ini.



Gambar 1.1: Jaringan feedforward dengan satu *hidden layer*: tiga neuron masukan, empat neuron tersembunyi, dua neuron keluaran. Setiap panah membawa sebuah bobot. Informasi mengalir satu arah, dari kiri ke kanan—itulah arti *feedforward*.

Kata "feedforward" menegaskan bahwa informasi mengalir lurus ke depan: dari masukan, ke *hidden*, ke keluaran, tanpa pernah berputar balik. Tidak ada gelung (*loop*), tidak ada ingatan internal. Inilah bentuk jaringan yang paling sederhana sekaligus paling mendasar; arsitektur-arsitektur termasyhur seperti *convolutional* dan *recurrent network* pada dasarnya adalah variasi dan perluasan dari ide ini.

⇝ Intuisi

Anggaplah tiap neuron *hidden* sebagai seorang detektor pola sederhana yang hanya bisa berkata "ya, polaku terlihat" atau "tidak". Sendirian ia tidak banyak berguna. Tetapi gabungan banyak detektor sederhana, yang masing-masing menyalakan diri pada situasi berbeda, ternyata cukup untuk menyusun keputusan yang rumit—persis seperti banyak saksi sederhana yang, jika digabungkan, bisa melukiskan satu kejadian secara utuh.

1.2 Mengapa cukup satu *hidden layer*?

Mungkin Anda bertanya: kalau jaringan modern punya puluhan bahkan ratusan lapisan, mengapa buku ini berkutat pada satu lapisan saja? Ada dua

alasan, dan keduanya penting.

Alasan pertama bersifat *pedagogis*. Hampir seluruh gagasan inti *neural network*—bagaimana sinyal mengalir maju, bagaimana kesalahan diukur, bagaimana gradien mengalir mundur lewat *backpropagation*, dan bagaimana bobot diperbarui—semuanya sudah hadir utuh pada satu *hidden layer*. Menambah lapisan kedua, ketiga, dan seterusnya tidak memunculkan ide baru; ia hanya mengulang pola yang sama berlapis-lapis. Dengan menahan diri pada satu lapisan, kita bisa menurunkan setiap rumus secara lengkap tanpa tenggelam dalam indeks dan notasi.

Alasan kedua bersifat *teoretis*, dan cukup mengejutkan. Sebuah teorema yang dikenal sebagai *universal approximation theorem* [2, 3] menyatakan, secara longgar, bahwa jaringan dengan **satu** *hidden layer* saja—asalkan neuronnya cukup banyak—sudah mampu mendekati hampir semua fungsi kontinu sebaik yang kita mau. Jadi satu *hidden layer* bukan sekadar mainan; secara prinsip ia sudah sangat ekspresif.

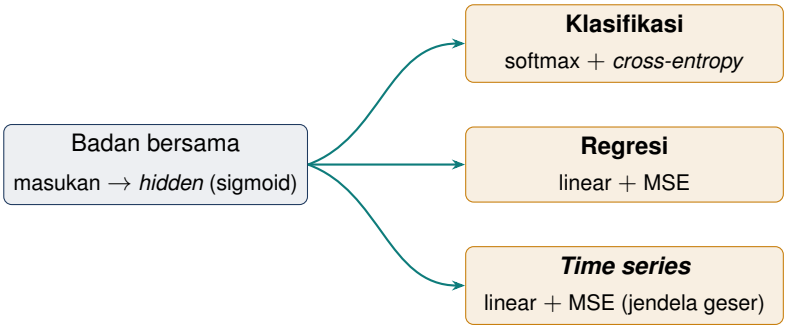
Catatan

"Mampu mendekati" tidak sama dengan "mudah dilatih" atau "efisien". Dalam praktik, jaringan dalam (*deep*) sering kali jauh lebih hemat dan lebih mudah dilatih untuk masalah rumit. Universal approximation menjamin keberadaannya, bukan kemudahannya. Kita kembali ke soal ini di bab penutup.

1.3 Satu model, tiga wajah

Inilah gagasan yang menjadi tulang punggung buku ini. Tiga tugas yang akan kita kerjakan—*klasifikasi*, *regresi*, dan *time series forecasting*—sekilas terdengar seperti tiga masalah berbeda yang menuntut tiga model berbeda. Kenyataannya tidak demikian. Kita memakai *badan* jaringan yang sama persis: masukan $\rightarrow$ satu *hidden layer* bersigmoid. Yang berganti hanyalah *kepala*-nya, yaitu lapisan keluaran beserta fungsi *loss* yang mengukur kesalahannya. Gambar 1.2 merangkum gagasan ini.

Pada **klasifikasi** kita ingin menebak *kategori*—misalnya spesies seekor penguin dari ukuran tubuhnya. Kepalanya memakai fungsi *softmax* yang mengubah keluaran menjadi peluang antar-kelas, dan kesalahannya diukur dengan *cross-entropy*. Pada **regresi** kita menebak *angka kontinu*— misalnya kekuatan tekan beton dari komposisinya. Kepalanya cukup berupa satu neuron linear, dan kesalahannya diukur dengan *mean squared error* (MSE). Pada **time series forecasting** kita menebak nilai berikutnya dari sebuah



Gambar 1.2: Benang merah buku ini. Badan jaringan tidak berubah; hanya kepala (lapisan keluaran dan fungsi *loss*) yang diganti sesuai tugas.

deret waktu—misalnya suhu besok dari suhu beberapa hari terakhir. Begitu kita menyusun datanya menjadi "jendela geser" (*sliding window*), tugas ini *berubah menjadi regresi biasa*, dan kepalanya sama persis dengan kasus regresi.

⇝ Intuisi

Pikirkan badan jaringan sebagai juru masak yang selalu menyiapkan bumbu dasar yang sama. Yang menentukan apakah hidangannya menjadi sup, tumisan, atau saus hanyalah langkah terakhir di penyajian. Mempelajari satu juru masak yang baik jauh lebih hemat daripada mempelajari tiga juru masak berbeda.

1.4 Peta jalan buku

Perjalanan kita tersusun dalam empat bagian.

Bagian I — Bekal. Bab ini dan Bab 2. Setelah pengantar ini, Bab 2 menyegarkan kembali bekal matematika yang benar-benar kita pakai: turunan, aturan rantai, gradien, serta perkalian matriks dan vektor. Tidak lebih dari yang diperlukan.

Bagian II — Membangun otaknya. Bab 3 membedah anatomi *forward pass* sebuah jaringan satu *hidden layer*. Bab 4—jantung buku ini—menurunkan *backpropagation* dari nol, mula-mula dalam notasi skalar lalu dirapikan ke notasi matriks. Bab 5 membahas optimisasi, dari *gradient descent* polos, naik ke *stochastic gradient descent*, hingga sentuhan *momentum* dan Adam. Bab 6 merakit semuanya menjadi kelas `NeuralNetwork` dari nol dengan NumPy, dan Bab 7 mempertajam pelatihannya dengan mengimplementasikan mini-batch, momentum, dan Adam dari nol lalu mengadu

kecepatannya.

Bagian III — Tiga wajah, satu model. Bab 8 sampai Bab 10 menerapkan jaringan yang sama untuk klasifikasi (Bab 8), regresi (Bab 9), dan *time series* (Bab 10). Tiap bab memakai data publik, dimulai dari implementasi NumPy lalu dibandingkan dengan TensorFlow.

Bagian IV — Penutup. Bab 12 merangkum kiat praktis—normalisasi, *overfitting*, dan alasan orang akhirnya beralih ke jaringan yang lebih dalam—serta menunjukkan ke mana melangkah selanjutnya. Apendiks memuat kode lengkap, panduan menyiapkan lingkungan kerja, dan sumber setiap dataset.

1.5 Cara membaca buku ini

Sepanjang buku, Anda akan menjumpai empat jenis kotak berwarna yang masing-masing punya peran tetap:

- **Intuisi** (amber) memberi gambaran kasar sebelum kita masuk ke rumus—tempat membangun "rasa" sebelum berhitung.
- **Definisi** (biru tua) menetapkan istilah atau notasi secara presisi.
- **Ingat Kembali** (teal) menyegarkan konsep matematika yang mungkin sudah lama tak Anda sentuh, tepat saat ia dibutuhkan.
- **Catatan** (merah) memperingatkan jebakan umum atau salah paham yang sering terjadi.

Mengenai notasi, kita berusaha konsisten sejak awal: huruf tebal kecil seperti $\boldsymbol{x}$ menandai *vektor*, huruf tebal kapital seperti $\boldsymbol{W}$ menandai *matriks*, dan huruf biasa seperti w_{ij} menandai sebuah *skalar* (satu angka tunggal). Kotak berikut merangkumnya sebagai rujukan cepat.

Konvensi notasi

$x,\ w$	skalar (satu angka)
$\boldsymbol{x},\ \boldsymbol{b}$	vektor (huruf tebal kecil)
$\boldsymbol{W}$	matriks (huruf tebal kapital)
$\hat{y}$	nilai *tebakan* jaringan (beda dari target y)
$\sigma(\cdot)$	fungsi aktivasi sigmoid
$\mathcal{L}$	fungsi *loss* (ukuran kesalahan)

Anda tidak perlu menghafal apa pun sekarang. Setiap simbol akan kita perkenalkan ulang saat pertama kali dipakai. Yang penting Anda bawa

dari bab ini hanyalah satu kalimat: *neural network adalah fungsi yang belajar dari contoh, dan satu hidden layer sudah cukup untuk memahami nyaris seluruh mekanismenya*. Mari kita siapkan bekal matematikanya.

Soal Latihan

1.1 [Konsep] Jelaskan dengan kata-kata Anda sendiri mengapa sebuah *neural network* pada hakikatnya "hanyalah sebuah fungsi". Apa masukan dan keluaran fungsi itu, dan apa yang "dipelajari"?

1.2 [Konsep] Benang merah buku ini adalah "satu model, ujungnya yang ganti". Untuk ketiga tugas (klasifikasi, regresi, *time series*), sebutkan bagian mana dari jaringan yang *tetap* dan bagian mana yang *berganti*.

1.3 [Konsep] Teorema *universal approximation* menjamin *keberadaan*, bukan *kemudahan*. Jelaskan perbedaan kedua hal itu dan mengapa ia penting dalam praktik.

1.4 [Konsep] Kata "feedforward" berarti informasi mengalir satu arah tanpa gelung. Kemampuan apa yang ditiadakan oleh ketiadaan gelung ini, dan pada tugas seperti apa hal itu mungkin terasa membatasi?

1.5 [Hitung] Tinjau jaringan dengan 3 neuron masukan, 5 neuron tersembunyi, dan 2 neuron keluaran. Berapa banyak panah (sambungan berbobot) di antara lapisan masukan dan tersembunyi? Antara tersembunyi dan keluaran? Berapa totalnya?

2

Bekal Matematika Seperlunya

Sebelum membongkar mesin sebuah *neural network*, kita perlu beberapa alat. Kabar baiknya, alatnya tidak banyak. Seluruh isi buku ini hanya bersandar pada empat gagasan: *turunan*, *aturan rantai*, *gradien*, dan *perkalian matriks*. Itu saja. Bab ini menyegarkannya kembali—bukan sebagai kuliah penuh kalkulus atau aljabar linear, melainkan sebagai bekal secukupnya yang langsung kita pakai nanti.

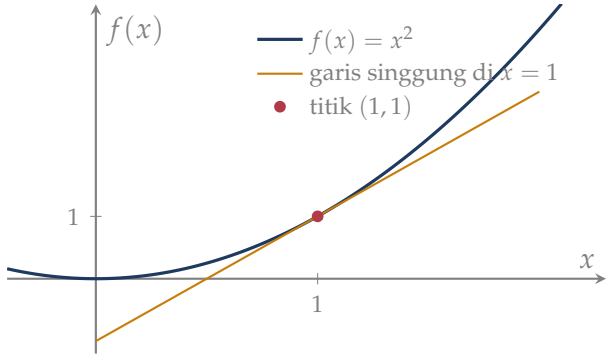
Kalau sebagian di antaranya terasa seperti teman lama yang sudah lama tak Anda jumpai, itu wajar. Bacalah santai. Anda tidak perlu mengerjakan soal latihan seabrek; Anda hanya perlu cukup nyaman dengan keempat gagasan ini sehingga, saat *backpropagation* kita turunkan di Bab 4, tidak ada langkah yang terasa ajaib.

2.1 Fungsi, turunan, dan kemiringan

Sebuah *fungsi* adalah mesin yang menerima angka dan mengembalikan angka. Tulis $f(x) = x^2$, maka memasukkan 3 menghasilkan 9. Pertanyaan paling penting yang akan kita ajukan berkali-kali sepanjang buku ini sederhana: *kalau masukannya digeser sedikit, seberapa besar keluarannya berubah?* Jawaban atas pertanyaan itulah yang disebut **turunan** (*derivative*).

Secara geometris, turunan di sebuah titik adalah *kemiringan* (*slope*) garis singgung kurva di titik itu. Kemiringan curam berarti keluaran sangat peka terhadap perubahan masukan; kemiringan landai berarti keluaran nyaris tak bergeming. Gambar 2.1 memperlihatkannya untuk $f(x) = x^2$ di $x = 1$.

Kita menuliskan turunan f terhadap x sebagai $f'(x)$ atau $\frac{df}{dx}$. Untuk $f(x) = x^2$, turunannya adalah $f'(x) = 2x$, sehingga di $x = 1$ kemiringannya $f'(1) = 2$—persis sesuai garis oranye pada gambar.



Gambar 2.1: Turunan sebagai kemiringan garis singgung. Di $x = 1$, kurva $f(x) = x^2$ memiliki kemiringan $f'(1) = 2$: setiap pergeseran kecil x ke kanan membuat f naik kira-kira dua kali lipat besar pergeserannya.

Turunan = laju perubahan

Turunan menjawab: "jika x bertambah sebesar Δx yang sangat kecil, berapa pertambahan f?" Jawabannya kira-kira $\Delta f \approx f'(x)\,\Delta x$. Inilah satu-satunya makna turunan yang perlu Anda pegang. Semua rumus di bawah hanyalah cara cepat menghitungnya.

2.2 Aturan turunan yang kita pakai

Kita tidak perlu seluruh tabel turunan. Hanya beberapa yang muncul berulang:

- **Pangkat:** jika $f(x) = x^n$ maka $f'(x) = n\,x^{n-1}$. Contoh: turunan x^2 adalah $2x$.
- **Konstanta dikali:** jika $f(x) = c\,g(x)$ maka $f'(x) = c\,g'(x)$. Konstanta ikut terbawa apa adanya.
- **Penjumlahan:** jika $f(x) = g(x) + h(x)$ maka $f'(x) = g'(x) + h'(x)$. Turunan dari jumlah adalah jumlah turunan.
- **Eksponensial:** turunan e^x adalah e^x itu sendiri—ia unik karena laju perubahannya sama dengan nilainya. Kita memerlukannya karena fungsi sigmoid mengandung e^{-x}.

Itu sudah cukup untuk hampir seluruh perhitungan kita. Yang menyatukan semuanya—dan yang akan menjadi tulang punggung *backpropaga-*

tion—adalah satu aturan terakhir yang pantas mendapat seksinya sendiri.

2.3 Aturan rantai

Bayangkan dua mesin yang dirangkai: keluaran mesin pertama menjadi masukan mesin kedua. Anda memutar tuas di mesin pertama; pertanyaannya, seberapa cepat jarum di mesin kedua bergerak? Akal sehat menjawab: kalikan kedua laju. Kalau mesin pertama membuat keluarannya bergerak tiga kali lebih cepat dari tuasnya, dan mesin kedua membuat jarumnya bergerak dua kali lebih cepat dari masukannya, maka jarum akhir bergerak $3 \times 2 = 6$ kali lebih cepat dari tuas. Itulah seluruh isi **aturan rantai** (*chain rule*).

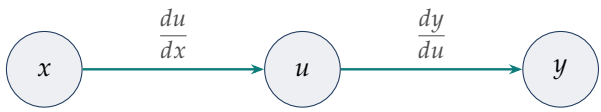
Aturan rantai

Jika y bergantung pada u, dan u bergantung pada x (sehingga $y = f(g(x))$ dengan $u = g(x)$), maka

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}.$$

Laju perubahan total adalah hasil kali laju perubahan di setiap tahap.

Gambar 2.2 menggambarkan rangkaian ini. Kelak, sebuah *neural network* tidak lain adalah rangkaian panjang seperti ini—masukan melewati bobot, lalu sigmoid, lalu bobot lagi, lalu *loss*—dan *backpropagation* hanyalah penerapan aturan rantai di sepanjang rangkaian itu, dari belakang ke depan.



Gambar 2.2: Aturan rantai sebagai rangkaian. Untuk mengetahui pengaruh x terhadap y, kalikan laju perubahan di sepanjang jalur: $\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$.

Contoh terhitung: turunan sigmoid

Mari kita pakai aturan rantai untuk satu hasil yang akan kita andalkan terus-menerus: turunan fungsi sigmoid

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

Tulis $u = 1 + e^{-x}$, sehingga $\sigma = u^{-1}$. Dengan aturan pangkat, $\frac{d\sigma}{du} = -u^{-2}$. Dengan aturan rantai pada bagian dalam, $\frac{du}{dx} = -e^{-x}$. Kalikan keduanya:

$$\frac{d\sigma}{dx} \;=\; \frac{d\sigma}{du}\cdot\frac{du}{dx} \;=\; \left(-u^{-2}\right)\left(-e^{-x}\right) \;=\; \frac{e^{-x}}{\left(1+e^{-x}\right)^{2}}.$$

Bentuk ini bisa dirapikan menjadi sesuatu yang sangat elegan. Perhatikan bahwa $\sigma(x) = \frac{1}{1+e^{-x}}$ dan $1-\sigma(x) = \frac{e^{-x}}{1+e^{-x}}$, sehingga

$$\boxed{\sigma'(x) \;=\; \sigma(x)\,\big(1-\sigma(x)\big).} \tag{2.1}$$

⇝ Intuisi

Persamaan (2.1) adalah hadiah yang membuat sigmoid digemari dalam pengajaran: untuk menghitung turunannya, kita *tidak perlu menghitung ulang apa pun*. Kalau nilai $\sigma(x)$ sudah kita punya dari *forward pass*, turunannya tinggal $\sigma(1-\sigma)$. Di Bab 4 sifat ini membuat *backward pass* menjadi murah dan rapi.

2.4 Lebih dari satu variabel: turunan parsial dan gradien

Fungsi sebuah *neural network* tidak bergantung pada satu angka, melainkan pada ribuan bobot sekaligus. Kita perlu bahasa untuk membicarakan "seberapa peka keluaran terhadap *masing-masing* masukan".

Caranya: ambil satu variabel, anggap semua variabel lain sebagai konstanta, lalu turunkan seperti biasa. Hasilnya disebut **turunan parsial** dan ditulis dengan lambang ∂ (dibaca "del"). Untuk $f(x,y) = x^2 + 3xy$, misalnya,

$$\frac{\partial f}{\partial x} = 2x + 3y, \qquad \frac{\partial f}{\partial y} = 3x.$$

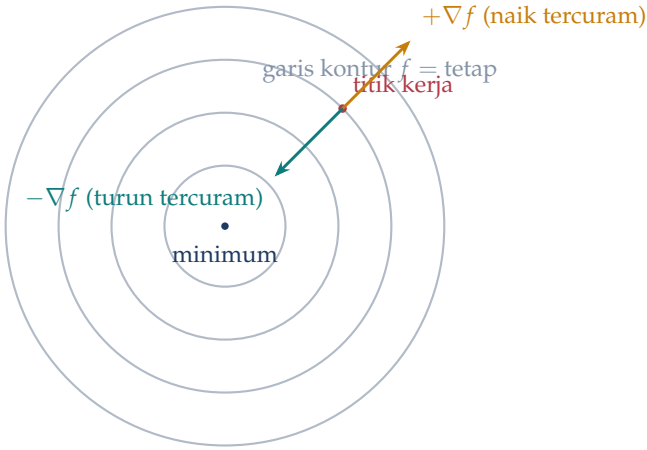
Saat menurunkan terhadap x, suku $3xy$ memperlakukan y sebagai konstanta (jadi turunannya $3y$); saat menurunkan terhadap y, giliran x yang dianggap konstanta.

Kumpulan semua turunan parsial, disusun menjadi sebuah vektor, disebut **gradien** dan ditulis ∇f:

$$\nabla f \;=\; \left(\frac{\partial f}{\partial x},\, \frac{\partial f}{\partial y}\right).$$

Gradien punya tafsir geometris yang indah dan sangat penting bagi kita: *ia menunjuk ke arah kenaikan tercuram*, dan panjangnya menyatakan seberapa curam kenaikan itu. Akibatnya, arah *kebalikannya*, $-\nabla f$, menunjuk ke

arah *penurunan* tercuram. Gambar 2.3 menggambarkannya pada sebuah "mangkuk" $f(x, y) = x^2 + y^2$.



Gambar 2.3: Gradien pada mangkuk $f(x, y) = x^2 + y^2$. Lingkaran-lingkaran adalah garis kontur (tempat f bernilai sama). Gradien ∇f tegak lurus kontur dan menunjuk keluar (mendaki); kebalikannya, $-\nabla f$, menunjuk ke pusat tempat f paling kecil. Inilah arah yang akan kita ikuti saat melatih jaringan.

⇝ Intuisi

Inilah inti dari seluruh proses pelatihan, dibocorkan lebih awal. Anggap f sebagai *kesalahan* jaringan dan sumbu-sumbunya sebagai bobot. Melatih jaringan berarti turun menuruni mangkuk menuju kesalahan terkecil. Di setiap langkah kita hitung $-\nabla f$ (arah turun tercuram) lalu melangkah sedikit ke arah itu. Itulah *gradient descent*, yang kita bahas tuntas di Bab 5.

2.5 Aturan rantai bercabang: mesin sesungguhnya di balik backprop

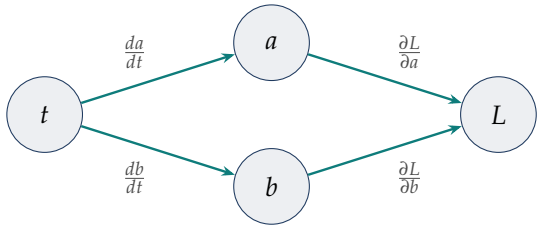
Ada satu kerumitan yang harus kita siapkan sejak sekarang, karena ia adalah *justru* mekanisme yang membuat *backpropagation* bekerja. Pada rangkaian sederhana di Gambar 2.2, sebuah variabel hanya punya satu jalur menuju keluaran. Tetapi di sebuah jaringan, satu bobot bisa memengaruhi *loss* melalui *banyak jalur* sekaligus—misalnya karena satu neuron tersembunyi terhubung ke beberapa neuron keluaran.

Aturannya begini: kalau sebuah variabel t memengaruhi keluaran L melalui beberapa perantara a dan b, maka kontribusi dari setiap jalur *dijum-*

lahkan:

$$\frac{dL}{dt} = \frac{\partial L}{\partial a}\frac{da}{dt} + \frac{\partial L}{\partial b}\frac{db}{dt}. \tag{2.2}$$

Gambar 2.4 memvisualkannya. Bacalah persamaan ini sebagai: "telusuri setiap jalur dari t ke L, kalikan laju perubahan di sepanjang jalur itu, lalu jumlahkan semua jalur."



Gambar 2.4: Aturan rantai bercabang. Variabel t memengaruhi L lewat dua jalur (melalui a dan melalui b). Pengaruh totalnya adalah *jumlah* dari kedua jalur, sebagaimana Persamaan (2.2).

Catatan

Penjumlahan di Persamaan (2.2) mudah terlupa, padahal di sinilah letak kesalahan paling umum saat menurunkan *backpropagation* sendiri. Ingat baik-baik: *satu sumber, banyak jalur, semua dijumlahkan*. Kita akan melihat penjumlahan ini muncul persis saat menghitung gradien sebuah neuron tersembunyi di Bab 4.

2.6 Vektor dan matriks secukupnya

Kita beralih sebentar dari kalkulus ke aljabar linear—tetapi hanya secukupnya. Sebuah **vektor** adalah daftar angka yang disusun tegak, misalnya

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix},$$

yang bisa kita pakai untuk menampung ketiga fitur masukan sebuah contoh data. Sebuah **matriks** adalah tabel angka berbaris dan berkolom, dan kita akan memakainya untuk menampung seluruh bobot satu lapisan sekaligus. Sepanjang buku ini, huruf tebal kecil seperti $\boldsymbol{x}$ berarti vektor dan huruf tebal kapital seperti $\boldsymbol{W}$ berarti matriks (lihat kembali konvensi notasi di Bab 1).

Dua operasi yang kita butuhkan:

Hasil kali titik (dot product)

Hasil kali titik dua vektor sepanjang sama adalah jumlah hasil kali pasangan elemennya, menghasilkan *satu* angka:

$$\boldsymbol{x}^\top \boldsymbol{w} \;=\; x_1 w_1 + x_2 w_2 + \cdots + x_n w_n \;=\; \sum_{i=1}^{n} x_i w_i.$$

Inilah cara sebuah neuron memadukan banyak masukan menjadi satu nilai.

Hasil kali matriks-vektor

Mengalikan matriks $\boldsymbol{W}$ (berukuran $m \times n$) dengan vektor $\boldsymbol{x}$ (berpanjang n) menghasilkan vektor berpanjang m. Elemen ke-i hasilnya adalah hasil kali titik antara *baris ke-i* matriks dengan $\boldsymbol{x}$:

$$(\boldsymbol{W}\boldsymbol{x})_i \;=\; \sum_{j=1}^{n} W_{ij}\, x_j.$$

2.7 Perkalian matriks: inti komputasi neural network

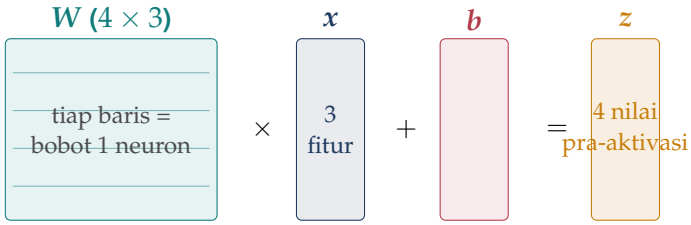
Sekarang dua dunia itu—neuron dan matriks—bertemu. Tinjau lapisan tersembunyi pada Gambar 1.1. Setiap neuron tersembunyi mengambil semua masukan, mengalikannya dengan bobot masing-masing, lalu menjumlahkannya—persis sebuah hasil kali titik. Kalau ada empat neuron tersembunyi, kita melakukan empat hasil kali titik.

Daripada menuliskannya satu per satu, kita susun keempat baris bobot itu menjadi satu matriks $\boldsymbol{W}$. Maka keempat hasil sekaligus dapat ditulis sebagai *satu* hasil kali matriks-vektor, lalu ditambah vektor bias $\boldsymbol{b}$:

$$\boldsymbol{z} \;=\; \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}. \tag{2.3}$$

Gambar 2.5 memperlihatkan susunannya: tiap baris $\boldsymbol{W}$ adalah "resep bobot" satu neuron tersembunyi, dan hasil perkaliannya adalah nilai *pra-aktivasi* $\boldsymbol{z}$ untuk seluruh neuron pada lapisan itu, sekaligus.

Inilah alasan kita repot-repot menyegarkan aljabar linear: menuliskan jaringan dalam bahasa matriks bukan sekadar mempercantik notasi. Ia membuat rumusnya ringkas, membuat penurunan *backpropagation* jauh lebih rapi, dan—secara praktis—membuat NumPy maupun TensorFlow bisa mengerjakannya secepat kilat, karena pustaka-pustaka itu memang dioptimalkan



Gambar 2.5: Seluruh lapisan tersembunyi dalam satu operasi. Matriks bobot $\boldsymbol{W}$ dikalikan vektor masukan $\boldsymbol{x}$ lalu ditambah bias $\boldsymbol{b}$, menghasilkan vektor pra-aktivasi $\boldsymbol{z}$ untuk semua neuron sekaligus (Persamaan (2.3)).

untuk perkalian matriks.

Perkalian elemen (Hadamard)

Satu operasi terakhir yang akan kita temui di Bab 4. Berbeda dari perkalian matriks, *perkalian elemen* mengalikan dua vektor sepanjang sama posisi demi posisi, menghasilkan vektor sepanjang sama. Kita lambangkan dengan $\odot$:

$$\boldsymbol{a} \odot \boldsymbol{b} = \begin{pmatrix} a_1 b_1 \\ a_2 b_2 \\ \vdots \\ a_n b_n \end{pmatrix}.$$

Operasi ini muncul saat kita mengalikan gradien dengan turunan sigmoid $\sigma'(z)$ secara elemen demi elemen.

2.8 Bekal sudah lengkap

Mari kita rangkum apa yang baru saja kita kumpulkan, karena keempat alat inilah yang akan kita pakai berulang-ulang:

- **Turunan** mengukur kepekaan keluaran terhadap masukan—laju perubahan, atau kemiringan.
- **Aturan rantai** mengalikan laju perubahan di sepanjang rangkaian; versi bercabangnya *menjumlahkan* kontribusi semua jalur. Inilah mesin *backpropagation*.
- **Gradien** mengumpulkan semua turunan parsial; $-\nabla f$ menunjuk arah

penurunan tercuram, yang akan kita susuri saat melatih jaringan.

- **Perkalian matriks** memadatkan seluruh perhitungan satu lapisan menjadi satu operasi $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$.

Tidak ada lagi alat baru yang perlu Anda pelajari setelah ini; sisanya hanyalah merangkai keempatnya. Dengan bekal ini di tangan, kita siap membuka anatomi jaringan satu *hidden layer* dan menjalankan *forward pass*-nya yang pertama, di Bab 3.

Soal Latihan

2.1 [Hitung] Hitung turunan tiap fungsi berikut: (a) $f(x) = 4x^3 - 2x + 5$; (b) $f(x) = e^{3x}$ (pakai aturan rantai); (c) $f(x) = (2x+1)^4$.

2.2 [Derivasi] Dengan aturan rantai, turunkan $\dfrac{d}{dx}\tanh(x)$, dengan $\tanh(x) = \dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$. Tunjukkan hasilnya dapat ditulis $1 - \tanh^2(x)$.

2.3 [Hitung] Untuk $f(x, y) = x^2 y + \sin x$, hitung $\frac{\partial f}{\partial x}$ dan $\frac{\partial f}{\partial y}$, lalu tuliskan gradien ∇f.

2.4 [Derivasi] Misalkan $\mathcal{L} = a^2 + b^2$ dengan $a = 2t$ dan $b = t + 1$. Hitung $\dfrac{d\mathcal{L}}{dt}$ dengan dua cara: (a) substitusikan dulu lalu turunkan; (b) pakai aturan rantai bercabang Persamaan (2.2). Pastikan keduanya sama.

2.5 [Hitung] Diberi $\boldsymbol{W} = \begin{pmatrix} 1 & 2 \\ 0 & -1 \\ 3 & 1 \end{pmatrix}$, $\boldsymbol{x} = (2, 1)^\top$, dan $\boldsymbol{b} = (1, 0, -2)^\top$, hitung $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$.

2.6 [Hitung] Hitung hasil kali elemen (Hadamard) $(2, -1, 3) \odot (0{,}5,\ 4,\ -2)$.

3

Anatomi Feedforward Satu Hidden Layer

Di Bab 1 kita melihat jaringan dari kejauhan: kotak-kotak bulat yang dihubungkan panah. Di Bab 2 kita menyiapkan alatnya. Sekarang kita masuk ke dalam dan membongkar mesinnya bagian demi bagian—mulai dari komponen terkecil, satu neuron, sampai seluruh jaringan menyala dan menghasilkan tebakan pertamanya. Proses "menyalakan" jaringan untuk mengubah masukan menjadi keluaran ini disebut *forward pass*, dan ia adalah tujuan akhir bab ini.

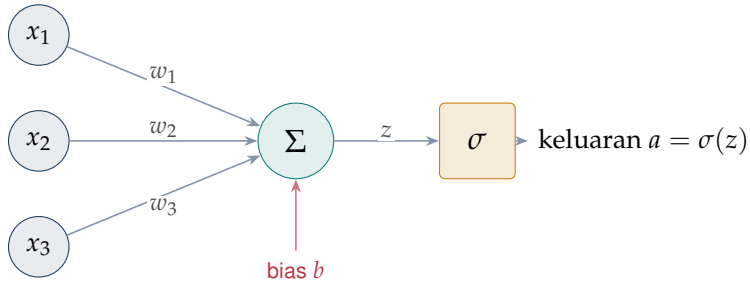
Kita masih belum akan melatih apa pun di sini—tidak ada bobot yang berubah, tidak ada yang "belajar". Bobotnya kita anggap sudah ada nilainya (entah dari mana), dan kita hanya ingin tahu: *diberi masukan, bagaimana persisnya jaringan menghitung keluaran?* Begitu mekanisme maju ini jelas, Bab 4 baru akan membahas bagaimana bobot itu disetel.

3.1 Satu neuron: blok penyusun

Seluruh jaringan tersusun dari satu komponen yang diulang-ulang: **neuron**. Sebuah neuron melakukan dua langkah sederhana. Pertama, ia mengambil semua masukannya, mengalikan tiap masukan dengan sebuah bobot, lalu menjumlahkan semuanya dan menambahkan sebuah *bias*. Kedua, hasil penjumlahan itu dilewatkan melalui sebuah *fungsi aktivasi*. Gambar 3.1 memperlihatkan keduanya.

Dalam rumus, langkah pertama (penjumlahan berbobot) menghasilkan angka yang kita sebut *pra-aktivasi* dan tulis z:

$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 + b = \sum_i w_i x_i + b. \tag{3.1}$$



Gambar 3.1: Anatomi satu neuron. Masukan x_1, x_2, x_3 dikalikan bobotnya masing-masing, dijumlahkan bersama bias b menjadi pra-aktivasi z, lalu dilewatkan fungsi aktivasi σ menghasilkan keluaran a.

Anda mungkin mengenali penjumlahan ini: ia persis sebuah *hasil kali titik* antara vektor bobot $\boldsymbol{w}$ dan vektor masukan $\boldsymbol{x}$, ditambah bias. Langkah kedua melewatkannya melalui fungsi aktivasi:

$$a = \sigma(z). \tag{3.2}$$

Neuron

Sebuah neuron adalah fungsi yang memetakan vektor masukan $\boldsymbol{x}$ menjadi satu angka keluaran lewat dua tahap: pra-aktivasi $z = \boldsymbol{w}^\top \boldsymbol{x} + b$ (linear), lalu aktivasi $a = \sigma(z)$ (tak-linear). Bobot $\boldsymbol{w}$ dan bias b adalah angka-angka yang nanti akan dipelajari.

⇝ Intuisi

Bias b berperan seperti ambang. Tanpa bias, neuron dipaksa "melewati titik asal"; dengan bias, ia bisa menggeser ambang aktivasinya ke kiri atau ke kanan, sehingga lebih mudah "menyala" atau lebih sulit "menyala" sesuai kebutuhan. Itulah sebabnya bias hampir selalu disertakan.

3.2 Mengapa harus tak-linear? Fungsi aktivasi sigmoid

Langkah kedua—melewatkan z melalui fungsi aktivasi—tampak sepele, tetapi justru di situlah seluruh kekuatan jaringan berada. Untuk melihat alasannya, mari kita bayangkan sejenak jika fungsi aktivasi itu *tidak ada*, atau setara dengan itu, jika $\sigma(z) = z$ (fungsi identitas, yang linear).

Kalau setiap neuron hanya linear, maka menumpuk satu lapisan di atas lapisan lain tidak menghasilkan apa-apa yang baru. Keluaran lapisan kedua adalah kombinasi linear dari keluaran lapisan pertama, yang sudah merupakan kombinasi linear dari masukan—dan kombinasi linear dari kombinasi linear tetaplah satu kombinasi linear. Seluruh jaringan, betapapun lebarnya, runtuh menjadi setara dengan *satu* transformasi linear tunggal. Ia tak akan pernah bisa menangkap pola yang melengkung.

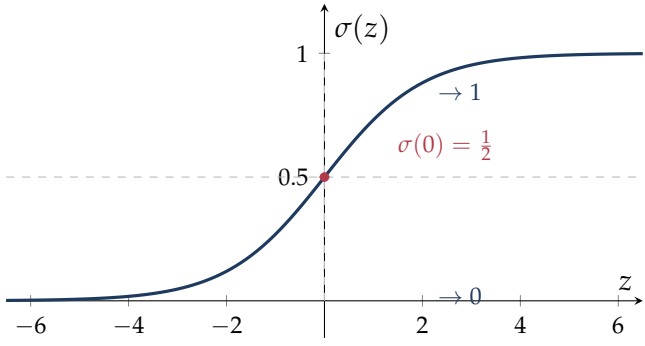
⇝ Intuisi

Tanpa ketaklinearan, jaringan hanya bisa menggambar garis lurus (atau bidang datar). Fungsi aktivasilah yang memberinya kemampuan menekuk dan melengkung, sehingga ia bisa memisahkan data yang tidak bisa dipisahkan oleh satu garis. Inilah yang "mengaktifkan" janji *universal approximation* dari Bab 1.

Fungsi aktivasi pilihan kita sepanjang buku ini adalah **sigmoid**,

$$\sigma(z) \;=\; \frac{1}{1+e^{-z}}, \tag{3.3}$$

yang grafiknya tampak pada Gambar 3.2. Bentuknya seperti huruf S yang dijulurkan: ia menekan bilangan berapa pun—dari $-\infty$ sampai $+\infty$—menjadi sebuah angka di antara 0 dan 1.



Gambar 3.2: Fungsi sigmoid. Ia memetakan seluruh garis bilangan ke selang $(0,1)$, bernilai tepat $\frac{1}{2}$ di $z=0$, naik secara monoton, dan mendatar ("jenuh") untuk z yang sangat besar atau sangat kecil.

Beberapa sifat sigmoid yang akan kita pakai: ia selalu menghasilkan nilai antara 0 dan 1; ia bernilai tepat $\frac{1}{2}$ saat $z=0$; ia naik secara monoton (makin besar z, makin besar keluarannya); dan—yang paling penting bagi Bab 4—turunannya sudah kita hitung di Persamaan (2.1): $\sigma'(z) = \sigma(z)\big(1-\sigma(z)\big)$.

Catatan

Perhatikan ekor grafik yang mendatar di kiri dan kanan. Di daerah "jenuh" (*saturated*) itu, kemiringan sigmoid nyaris nol, sehingga $\sigma'(z) \approx 0$. Nanti kita akan lihat bahwa kemiringan kecil ini membuat sinyal pembelajaran mengecil—masalah yang dikenal sebagai *vanishing gradient*. Inilah salah satu alasan praktik modern sering beralih ke aktivasi lain seperti ReLU; kita bahas singkat di Bab 12. Untuk memahami mekanismenya, sigmoid tetap pilihan terbaik.

3.3 Dari neuron ke lapisan: bentuk matriks

Satu *hidden layer* hanyalah sekumpulan neuron yang bekerja berdampingan, masing-masing menerima *seluruh* masukan yang sama tetapi dengan bobotnya sendiri-sendiri. Misalkan masukan kita berdimensi d (yakni $\boldsymbol{x} \in \mathbb{R}^d$) dan kita memasang m neuron tersembunyi. Maka neuron tersembunyi ke-j menghitung pra-aktivasinya sendiri,

$$z_j \;=\; \sum_{i=1}^{d} W_{ji}\, x_i + b_j,$$

lalu mengaktifkannya menjadi $h_j = \sigma(z_j)$.

Menuliskan m persamaan seperti itu satu per satu melelahkan. Untungnya, seperti sudah kita siapkan di Bagian 2.7, kita bisa menyusun semua bobot menjadi satu matriks $\boldsymbol{W}$ berukuran $m \times d$ (satu baris per neuron tersembunyi) dan semua bias menjadi satu vektor $\boldsymbol{b} \in \mathbb{R}^m$. Seluruh lapisan tersembunyi lalu muat dalam dua baris ringkas:

$$\boldsymbol{z} \;=\; \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}, \qquad \boldsymbol{h} \;=\; \sigma(\boldsymbol{z}). \tag{3.4}$$

Di sini $\sigma(\boldsymbol{z})$ berarti "terapkan sigmoid ke setiap elemen $\boldsymbol{z}$ secara sendiri-sendiri"—sebuah operasi per-elemen yang menghasilkan vektor $\boldsymbol{h}$ sepanjang sama. Vektor $\boldsymbol{h} \in \mathbb{R}^m$ inilah *representasi tersembunyi*: ringkasan masukan yang sudah ditekuk dan diolah, siap dipakai lapisan berikutnya.

3.4 Lapisan keluaran: kepala yang bisa diganti

Sekarang vektor tersembunyi $\boldsymbol{h}$ menjadi masukan bagi lapisan terakhir, yaitu *lapisan keluaran*. Mekanismenya sama persis: penjumlahan berbobot lalu sebuah fungsi. Untuk membedakannya dari lapisan tersembunyi, kita pakai

matriks bobot $\boldsymbol{V}$ (berukuran $k \times m$, dengan k banyaknya keluaran) dan bias $\boldsymbol{c} \in \mathbb{R}^k$:

$$\boldsymbol{u} \;=\; \boldsymbol{V}\boldsymbol{h} + \boldsymbol{c}, \qquad \hat{\boldsymbol{y}} \;=\; g(\boldsymbol{u}). \tag{3.5}$$

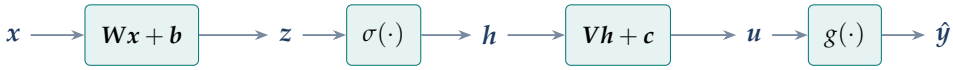
Di sini $\boldsymbol{u}$ adalah pra-aktivasi keluaran, dan g adalah *fungsi keluaran* yang kita pilih sesuai tugas. Inilah "kepala" yang dijanjikan di Bab 1—bagian yang berganti sementara badan tetap sama:

- Untuk **regresi**, g cukup fungsi identitas: $\hat{\boldsymbol{y}} = \boldsymbol{u}$. Keluaran dibiarkan berupa angka mentah, bebas bernilai berapa saja.
- Untuk **klasifikasi**, g adalah *softmax*, yang mengubah $\boldsymbol{u}$ menjadi peluang antar-kelas. Kita bahas tuntas di Bab 8.
- Untuk **time series**, kita perlakukan sebagai regresi, jadi g kembali identitas (Bab 10).

Sepanjang bab ini, untuk membuat contoh tetap konkret, kita pakai kasus paling sederhana: regresi dengan satu keluaran ($k = 1$), sehingga g identitas dan $\hat{\boldsymbol{y}} = \boldsymbol{u}$ hanyalah satu angka.

3.5 Forward pass lengkap

Mari satukan semuanya. Menggabungkan Persamaan (3.4) dan (3.5), seluruh perjalanan dari masukan ke tebakan terangkum dalam empat langkah berurutan. Gambar 3.3 menggambarkan alirannya, dan kotak definisi di bawahnya merangkum rumusnya sebagai rujukan.



Gambar 3.3: *Forward pass* sebagai jalur ban berjalan. Masukan $\boldsymbol{x}$ diubah menjadi pra-aktivasi $\boldsymbol{z}$, lalu representasi tersembunyi $\boldsymbol{h}$, lalu pra-aktivasi keluaran $\boldsymbol{u}$, dan akhirnya tebakan $\hat{\boldsymbol{y}}$. Kotak teal adalah operasi; simbol biru adalah nilai yang mengalir di antaranya.

Forward pass jaringan satu hidden layer

Diberi masukan $\boldsymbol{x}$ dan parameter $(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{V}, \boldsymbol{c})$:

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}, \quad \boldsymbol{h} = \sigma(\boldsymbol{z}), \quad \boldsymbol{u} = \boldsymbol{V}\boldsymbol{h} + \boldsymbol{c}, \quad \hat{\boldsymbol{y}} = g(\boldsymbol{u}).$$

Untuk regresi, g adalah identitas sehingga $\hat{\boldsymbol{y}} = \boldsymbol{u}$.

Contoh terhitung: menjalankan jaringan dengan tangan

Tidak ada yang lebih meyakinkan daripada menghitungnya sendiri. Ambil jaringan mungil dengan $d = 2$ masukan, $m = 2$ neuron tersembunyi, dan $k = 1$ keluaran (regresi). Misalkan parameternya sudah bernilai

$$\boldsymbol{W} = \begin{pmatrix} 0{,}5 & -0{,}3 \\ 0{,}8 & 0{,}2 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 0{,}1 \\ -0{,}4 \end{pmatrix}, \quad \boldsymbol{V} = \begin{pmatrix} 0{,}7 & -0{,}5 \end{pmatrix}, \quad \boldsymbol{c} = \begin{pmatrix} 0{,}2 \end{pmatrix},$$

dan kita beri masukan $\boldsymbol{x} = (1{,}0,\ 2{,}0)$.

Langkah 1 — pra-aktivasi tersembunyi. Hitung $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$:

$$\begin{aligned} z_1 &= 0{,}5(1{,}0) + (-0{,}3)(2{,}0) + 0{,}1 = 0{,}0, \\ z_2 &= 0{,}8(1{,}0) + 0{,}2(2{,}0) - 0{,}4 = 0{,}8. \end{aligned}$$

Langkah 2 — aktivasi. Terapkan sigmoid ke tiap elemen:

$$h_1 = \sigma(0{,}0) = 0{,}500, \qquad h_2 = \sigma(0{,}8) = \frac{1}{1 + e^{-0{,}8}} \approx 0{,}690.$$

Langkah 3 — pra-aktivasi keluaran. Hitung $\boldsymbol{u} = \boldsymbol{V}\boldsymbol{h} + \boldsymbol{c}$:

$$u = 0{,}7(0{,}500) + (-0{,}5)(0{,}690) + 0{,}2 = 0{,}350 - 0{,}345 + 0{,}200 = 0{,}205.$$

Langkah 4 — keluaran. Karena ini regresi, g identitas, sehingga

$$\hat{\boldsymbol{y}} = u = 0{,}205.$$

Itulah tebakan jaringan untuk masukan $(1{,}0,\ 2{,}0)$. Seluruh "kecerdasan" jaringan, pada tahap ini, hanyalah rangkaian sembilan perkalian-penjumlahan dan dua evaluasi sigmoid. Tidak ada yang ajaib—hanya aritmetika yang tertata.

Catatan

Apakah 0,205 tebakan yang *bagus*? Kita belum tahu, karena kita belum menetapkan jawaban yang benar maupun mengukur kesalahannya. Lagipula bobot di atas kita pilih sembarang. Mengukur kesalahan dan menyetel bobot agar tebakan membaik adalah pekerjaan Bab 4 dan Bab 5.

3.6 Memproses banyak contoh sekaligus

Dalam praktik kita jarang menjalankan jaringan untuk satu contoh saja; kita biasanya punya ratusan atau ribuan contoh data. Kabar baiknya, bentuk matriks membuat ini nyaris gratis. Susun n contoh sebagai baris-baris sebuah matriks masukan $\boldsymbol{X}$ berukuran $n \times d$, dan seluruh *forward pass* bisa dikerjakan sekali jalan dengan operasi matriks yang sama—inilah yang membuat NumPy dan TensorFlow sanggup mengolah satu *batch* data dalam sekejap. Kita akan memanfaatkannya secara serius di Bab 6 saat menulis kodenya; untuk sekarang, cukup ketahui bahwa rumus yang sama berlaku, hanya vektor berganti menjadi matriks.

3.7 Berapa banyak yang harus dipelajari?

Sebelum menutup, ada baiknya menghitung berapa banyak angka yang nanti harus disetel oleh proses pelatihan—yakni jumlah *parameter*. Pada jaringan satu *hidden layer* dengan d masukan, m neuron tersembunyi, dan k keluaran:

$$\underbrace{m\,d}_{\boldsymbol{W}} + \underbrace{m}_{\boldsymbol{b}} + \underbrace{k\,m}_{\boldsymbol{V}} + \underbrace{k}_{\boldsymbol{c}} \;=\; m(d+1) + k(m+1) \quad \text{parameter.}$$

Untuk jaringan mungil pada contoh tadi ($d = 2,\ m = 2,\ k = 1$), itu berarti $2(3) + 1(3) = 9$ parameter—persis sembilan angka yang kita tuliskan di $\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{V}, \boldsymbol{c}$. Sebuah jaringan nyata dengan, katakanlah, $d = 8$ masukan, $m = 16$ neuron tersembunyi, dan $k = 3$ keluaran sudah memiliki $16(9) + 3(17) = 195$ parameter. Setiap satunya akan disetel oleh *backpropagation*.

3.8 Mesin maju sudah jalan

Kita kini punya jaringan yang utuh dan bisa menghasilkan tebakan: masukan masuk di satu ujung, mengalir melalui $\boldsymbol{z}$, $\boldsymbol{h}$, dan $\boldsymbol{u}$, lalu keluar sebagai $\hat{\boldsymbol{y}}$ di ujung lain. Yang masih hilang hanyalah cara membuat tebakan itu *benar*.

Untuk itu kita perlu dua hal di bab berikutnya: sebuah cara *mengukur* seberapa salah tebakan jaringan (fungsi *loss*), dan sebuah cara *memperbaiki* setiap dari sekian banyak parameter agar kesalahan itu mengecil. Cara kedua itulah *backpropagation*—penerapan aturan rantai yang sudah kita siapkan di Bab 2—dan ia menanti di Bab 4.

Soal Latihan

3.1 [Hitung] Jalankan *forward pass* dengan tangan untuk jaringan $d = 2$, $m = 2$, $k = 1$ (regresi) dengan $\boldsymbol{W} = \left(\begin{smallmatrix} 1 & 0 \\ -1 & 2 \end{smallmatrix}\right)$, $\boldsymbol{b} = (0, 1)^\top$, $\boldsymbol{V} = (1, -1)$, $\boldsymbol{c} = (0)$, dan masukan $\boldsymbol{x} = (1, 1)^\top$. Hitung $\boldsymbol{z}$, $\boldsymbol{h}$, $\boldsymbol{u}$, dan $\hat{\boldsymbol{y}}$.

3.2 [Derivasi] Tunjukkan bahwa jika fungsi aktivasi adalah identitas ($\sigma(z) = z$), maka seluruh jaringan $\boldsymbol{u} = \boldsymbol{V}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + \boldsymbol{c}$ setara dengan satu transformasi linear tunggal. Tuliskan matriks dan vektor bias efektifnya.

3.3 [Hitung] Berapa banyak parameter pada jaringan dengan $d = 10$ masukan, $m = 20$ neuron tersembunyi, dan $k = 4$ keluaran? Pakai rumus $m(d + 1) + k(m + 1)$.

3.4 [Derivasi] Buktikan sifat simetri sigmoid: $\sigma(-x) = 1 - \sigma(x)$.

3.5 [Kode] Bagaimana `forward` berubah jika aktivasi tersembunyi diganti dari sigmoid ke tanh? Tuliskan barisnya, dan jelaskan bagian mana dari `backward` (Bab 4) yang ikut berubah.

4

Backpropagation dari Nol

Inilah jantung buku ini. Sampai sekarang jaringan kita bisa menebak, tetapi tebakannya sembarang—bobotnya kita pilih asal. Bab ini menjawab pertanyaan yang sesungguhnya: bagaimana cara mengetahui ke arah mana setiap bobot harus digeser agar tebakan membaik?

Jawabannya adalah *backpropagation* [1], dan meski namanya terdengar canggih, ia sebenarnya tidak lebih dari penerapan rapi aturan rantai yang sudah Anda kuasai di Bab 2—khususnya versi bercabang di Bagian 2.5. Kita akan menurunkannya dari nol, suku demi suku. Pertama dalam notasi skalar, supaya setiap langkah aturan rantai terlihat telanjang; lalu kita rapikan ke notasi matriks yang ringkas dan siap dikodekan di Bab 6. Tidak ada langkah yang akan diturunkan dari langit.

4.1 Mengukur kesalahan: fungsi loss

Sebelum bisa memperbaiki tebakan, kita perlu *angka* yang menyatakan seberapa salah tebakan itu. Angka itu disebut **fungsi loss**. Untuk regresi, pilihan paling alami adalah selisih kuadrat antara tebakan $\hat{\boldsymbol{y}}$ dan jawaban benar $\boldsymbol{y}$. Untuk satu contoh data, kita pakai

$$\mathcal{L} \;=\; \tfrac{1}{2}\sum_{j=1}^{k}\left(\hat{y}_j - y_j\right)^2 \;=\; \tfrac{1}{2}\,\|\hat{\boldsymbol{y}} - \boldsymbol{y}\|^2. \tag{4.1}$$

Ini adalah *mean squared error* (MSE) dalam bentuk jumlah kuadrat. Faktor $\frac{1}{2}$ di depan murni demi kerapian: nanti saat diturunkan, ia akan membatalkan angka 2 yang muncul dari pangkat dua, dan karena ia hanya konstanta pengali, ia tidak menggeser letak titik minimum sama sekali.

⇝ Intuisi

Bayangkan $\mathcal{L}$ sebagai "jarak" antara tebakan dan kebenaran. Kalau tebakan tepat, $\mathcal{L} = 0$. Makin jauh meleset, makin besar $\mathcal{L}$, dan karena dikuadratkan, kesalahan besar dihukum jauh lebih berat daripada kesalahan kecil. Seluruh tujuan pelatihan adalah membuat angka ini sekecil mungkin.

Untuk seluruh data berisi n contoh, loss totalnya adalah rata-rata dari loss tiap contoh. Karena turunan dari sebuah rata-rata adalah rata-rata dari turunan-turunannya, kita cukup menurunkan gradien untuk *satu* contoh lebih dahulu, lalu merata-ratakannya nanti (Bagian 4.7). Jadi sepanjang penurunan di bawah, anggaplah kita sedang menangani satu contoh $(\boldsymbol{x}, \boldsymbol{y})$ saja.

4.2 Ide besar backpropagation

Yang ingin kita hitung adalah empat gradien: $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}$, $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}}$, $\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}}$, dan $\frac{\partial \mathcal{L}}{\partial \boldsymbol{c}}$. Masing-masing memberi tahu kita ke arah mana menggeser bobot atau bias bersangkutan untuk menaikkan $\mathcal{L}$—sehingga arah *kebalikannya* menurunkannya.

Mencoba menghitung tiap turunan parsial secara langsung akan sangat berulang dan boros, sebab banyak perhitungan yang sama dipakai lagi dan lagi. Gagasan kunci *backpropagation* adalah menghitung dari *belakang ke depan* dan menyimpan hasil antara yang bisa dipakai ulang. Hasil antara itu adalah dua besaran istimewa, yang kita namai δ:

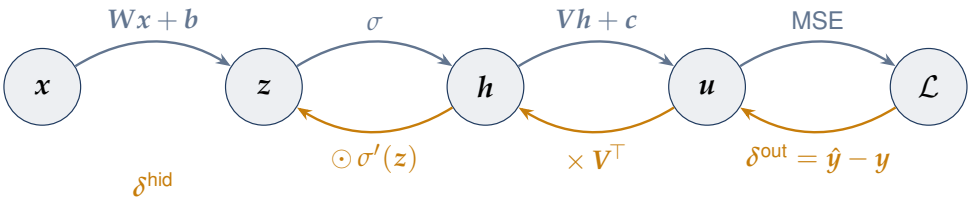
- $\boldsymbol{\delta}^{\text{out}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{u}}$, kepekaan loss terhadap pra-aktivasi keluaran;
- $\boldsymbol{\delta}^{\text{hid}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$, kepekaan loss terhadap pra-aktivasi tersembunyi.

Begitu kedua δ ini diketahui, keempat gradien yang kita incar jatuh dengan mudah. Dan $\boldsymbol{\delta}^{\text{hid}}$ sendiri dihitung dari $\boldsymbol{\delta}^{\text{out}}$ dengan "mengalirkannya mundur" melewati lapisan keluaran. Inilah arti kata *back-propagation*: kesalahan merambat mundur melalui jaringan. Gambar 4.1 merangkum dua arah aliran itu.

Kita kerjakan dari belakang: lapisan keluaran dulu, baru lapisan tersembunyi.

4.3 Gradien lapisan keluaran

Ingat forward pass-nya: $\boldsymbol{u} = \boldsymbol{V}\boldsymbol{h} + \boldsymbol{c}$ dan $\hat{\boldsymbol{y}} = g(\boldsymbol{u})$. Sepanjang bab ini kita pakai regresi dengan keluaran linear, jadi g adalah identitas dan $\hat{\boldsymbol{y}} = \boldsymbol{u}$.



Gambar 4.1: Dua arah aliran. Di atas (biru), *forward pass* membawa nilai dari x ke $\mathcal{L}$. Di bawah (amber), *backward pass* membawa gradien mundur dari $\mathcal{L}$: mulai dari δ^{out} di u, dikalikan $V^\top$ untuk pindah ke ruang h, lalu dikali per-elemen $\sigma'(z)$ menjadi δ^{hid} di z.

Langkah skalar: satu komponen

Mulai dari yang termudah, δ^{out}. Karena $\hat{y}_j = u_j$, loss pada Persamaan (4.1) menjadi $\mathcal{L} = \frac{1}{2}\sum_l (u_l - y_l)^2$. Turunkan terhadap satu u_j: hanya suku ke-j yang mengandung u_j, sehingga

$$\delta_j^{\text{out}} \;=\; \frac{\partial \mathcal{L}}{\partial u_j} \;=\; \tfrac{1}{2}\cdot 2\,(u_j - y_j) \;=\; u_j - y_j \;=\; \hat{y}_j - y_j. \tag{4.2}$$

Faktor $\frac{1}{2}$ tadi tepat membatalkan angka 2, persis seperti dijanjikan. Hasilnya sangat lugas: kepekaan loss terhadap keluaran ke-j hanyalah *selisih tebakan dengan kebenaran*.

Sekarang gradien bobot keluaran. Bobot V_{jp} menghubungkan h_p ke u_j lewat $u_j = \sum_p V_{jp} h_p + c_j$, jadi $\frac{\partial u_j}{\partial V_{jp}} = h_p$. Dengan aturan rantai,

$$\frac{\partial \mathcal{L}}{\partial V_{jp}} \;=\; \frac{\partial \mathcal{L}}{\partial u_j}\,\frac{\partial u_j}{\partial V_{jp}} \;=\; \delta_j^{\text{out}}\, h_p. \tag{4.3}$$

Serupa, karena $\frac{\partial u_j}{\partial c_j} = 1$,

$$\frac{\partial \mathcal{L}}{\partial c_j} \;=\; \delta_j^{\text{out}}. \tag{4.4}$$

Dirapikan ke matriks

Persamaan (4.3) berlaku untuk setiap pasangan (j, p). Susun semuanya dalam satu matriks, dan pola "elemen (j, p) = δ_j^{out} kali h_p" itu persis sebuah

hasil kali luar (*outer product*) antara vektor $\boldsymbol{\delta}^{\text{out}}$ dan vektor $\boldsymbol{h}$:

$$\boxed{\begin{aligned} &\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}} = \boldsymbol{\delta}^{\text{out}} \boldsymbol{h}^{\top}, \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}} = \boldsymbol{\delta}^{\text{out}}, \\ &\text{dengan} \quad \boldsymbol{\delta}^{\text{out}} = \hat{\boldsymbol{y}} - \boldsymbol{y}. \end{aligned}} \tag{4.5}$$

Itu menuntaskan lapisan keluaran. Perhatikan betapa $\boldsymbol{h}$ dari *forward pass* dipakai kembali di sini—tidak ada yang dihitung dua kali.

Catatan

Bentuk $\boldsymbol{\delta}^{\text{out}} = \hat{\boldsymbol{y}} - \boldsymbol{y}$ di sini khusus untuk pasangan keluaran-linear dengan loss MSE. Yang menarik—dan akan kita buktikan di Bab 8—pasangan *softmax* dengan *cross-entropy* untuk klasifikasi menghasilkan bentuk yang *sama persis*: $\boldsymbol{\delta}^{\text{out}} = \hat{\boldsymbol{y}} - \boldsymbol{y}$. Inilah keindahan tersembunyi dari benang merah "satu model, ujungnya yang ganti": kedua kepala berbeda itu menyodorkan sinyal mundur yang identik ke badan jaringan, sehingga seluruh penurunan di lapisan tersembunyi berikut berlaku apa adanya untuk keduanya.

4.4 Gradien lapisan tersembunyi

Sekarang bagian yang paling berisi—dan tempat aturan rantai *bercabang* dari Bab 2 akhirnya terpakai. Kita ingin $\boldsymbol{\delta}^{\text{hid}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$.

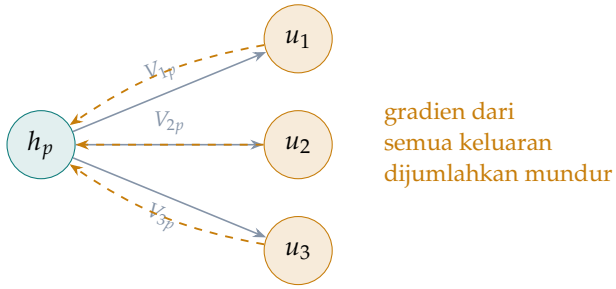
Langkah skalar: kepekaan terhadap satu neuron tersembunyi

Tinjau satu neuron tersembunyi, katakanlah yang ke-p, dengan pra-aktivasi z_p dan aktivasi $h_p = \sigma(z_p)$. Bagaimana z_p memengaruhi loss? Lewat h_p tentunya—tetapi h_p disuap ke *setiap* neuron keluaran $u_1, \ldots, u_k$ sekaligus (lihat Gambar 4.2). Jadi z_p punya banyak jalur menuju $\mathcal{L}$, dan menurut aturan rantai bercabang, kontribusi semua jalur itu harus *dijumlahkan*.

Pertama, kepekaan loss terhadap aktivasi h_p. Karena h_p memengaruhi $\mathcal{L}$ melalui setiap u_j, dan $\frac{\partial u_j}{\partial h_p} = V_{jp}$, kita jumlahkan atas semua j:

$$\frac{\partial \mathcal{L}}{\partial h_p} = \sum_{j=1}^{k} \frac{\partial \mathcal{L}}{\partial u_j} \frac{\partial u_j}{\partial h_p} = \sum_{j=1}^{k} \delta_j^{\text{out}} V_{jp}. \tag{4.6}$$

Kedua, kita teruskan mundur melewati sigmoid. Karena $h_p = \sigma(z_p)$, kita



Gambar 4.2: Satu neuron tersembunyi h_p menyuplai *semua* neuron keluaran. Maju (biru), ia menyebar ke $u_1, \dots, u_k$ lewat bobot V_{jp}. Mundur (amber putus-putus), gradien dari semua keluaran itu menyatu kembali ke h_p dengan *dijumlahkan*—inilah aturan rantai bercabang dalam wujud jaringan.

punya $\frac{\partial h_p}{\partial z_p} = \sigma'(z_p)$, sehingga

$$\delta_p^{\text{hid}} = \frac{\partial \mathcal{L}}{\partial z_p} = \frac{\partial \mathcal{L}}{\partial h_p}\frac{\partial h_p}{\partial z_p} = \left(\sum_{j=1}^{k} \delta_j^{\text{out}} V_{jp}\right)\sigma'(z_p). \tag{4.7}$$

Di sinilah turunan sigmoid manis dari Bab 2 terpakai: $\sigma'(z_p) = \sigma(z_p)\big(1 - \sigma(z_p)\big) = h_p(1 - h_p)$, yang lagi-lagi memakai ulang h_p dari forward pass tanpa hitungan baru.

Setelah $\boldsymbol{\delta}^{\text{hid}}$ di tangan, gradien bobot tersembunyi mengikuti pola yang sama persis seperti lapisan keluaran. Bobot W_{pi} menghubungkan x_i ke z_p lewat $z_p = \sum_i W_{pi} x_i + b_p$, jadi $\frac{\partial z_p}{\partial W_{pi}} = x_i$, dan

$$\frac{\partial \mathcal{L}}{\partial W_{pi}} = \frac{\partial \mathcal{L}}{\partial z_p}\frac{\partial z_p}{\partial W_{pi}} = \delta_p^{\text{hid}}\, x_i, \qquad \frac{\partial \mathcal{L}}{\partial b_p} = \delta_p^{\text{hid}}. \tag{4.8}$$

Dirapikan ke matriks

Penjumlahan $\sum_j \delta_j^{\text{out}} V_{jp}$ pada Persamaan (4.6) adalah komponen ke-p dari perkalian matriks-vektor $\boldsymbol{V}^\top \boldsymbol{\delta}^{\text{out}}$ (perhatikan: matriksnya *ditranspos*). Perkalian dengan $\sigma'(z_p)$ untuk setiap p adalah operasi per-elemen. Maka seluruhnya menjadi

$$\boxed{\begin{aligned} \boldsymbol{\delta}^{\text{hid}} &= \left(\boldsymbol{V}^\top \boldsymbol{\delta}^{\text{out}}\right) \odot \sigma'(\boldsymbol{z}), \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} &= \boldsymbol{\delta}^{\text{hid}}\, \boldsymbol{x}^\top, \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} = \boldsymbol{\delta}^{\text{hid}}. \end{aligned}} \tag{4.9}$$

dengan $\sigma'(\boldsymbol{z}) = \boldsymbol{h} \odot (\boldsymbol{1} - \boldsymbol{h})$ dihitung per-elemen.

Rincian: menelusuri satu bobot tersembunyi

Agar tak ada langkah yang tersembunyi, mari telusuri *satu* bobot tersembunyi W_{pi} dari ujung ke ujung, untuk kasus dua keluaran ($k = 2$). Bobot ini memengaruhi loss melalui rantai bertingkat:

$$W_{pi} \longrightarrow z_p \longrightarrow h_p \longrightarrow \{u_1, u_2\} \longrightarrow \mathcal{L}.$$

Perhatikan percabangannya: h_p menyuplai *kedua* neuron keluaran, jadi ada dua jalur menuju $\mathcal{L}$. Kita bongkar aturan rantai lapis demi lapis. Lapisan terluar:

$$\frac{\partial \mathcal{L}}{\partial W_{pi}} = \frac{\partial \mathcal{L}}{\partial z_p}\,\frac{\partial z_p}{\partial W_{pi}} = \frac{\partial \mathcal{L}}{\partial z_p} \cdot x_i,$$

karena $z_p = \sum_i W_{pi} x_i + b_p$ memberi $\frac{\partial z_p}{\partial W_{pi}} = x_i$. Lapisan berikutnya, melewati sigmoid:

$$\frac{\partial \mathcal{L}}{\partial z_p} = \frac{\partial \mathcal{L}}{\partial h_p}\,\frac{\partial h_p}{\partial z_p} = \frac{\partial \mathcal{L}}{\partial h_p} \cdot \sigma'(z_p).$$

Dan kini lapisan yang bercabang—di sinilah penjumlahan muncul, karena h_p memengaruhi $\mathcal{L}$ lewat u_1 *dan* u_2:

$$\frac{\partial \mathcal{L}}{\partial h_p} = \frac{\partial \mathcal{L}}{\partial u_1}\,\frac{\partial u_1}{\partial h_p} + \frac{\partial \mathcal{L}}{\partial u_2}\,\frac{\partial u_2}{\partial h_p} = \delta_1^{\text{out}} V_{1p} + \delta_2^{\text{out}} V_{2p},$$

sebab $u_j = \sum_p V_{jp} h_p + c_j$ memberi $\frac{\partial u_j}{\partial h_p} = V_{jp}$. Rangkai ketiganya menjadi satu:

$$\frac{\partial \mathcal{L}}{\partial W_{pi}} = \underbrace{\left(\delta_1^{\text{out}} V_{1p} + \delta_2^{\text{out}} V_{2p}\right)}_{= (\boldsymbol{V}^\top \boldsymbol{\delta}^{\text{out}})_p} \underbrace{\sigma'(z_p)}_{\text{lewat sigmoid}} \underbrace{x_i}_{\text{lewat } z_p}. \tag{4.10}$$

Bandingkan dengan bentuk matriksnya: faktor pertama persis komponen ke-p dari $\boldsymbol{V}^\top \boldsymbol{\delta}^{\text{out}}$, sehingga dua faktor pertama bersama adalah δ_p^{hid}, dan seluruhnya menjadi $\delta_p^{\text{hid}}\, x_i$— yakni elemen (p, i) dari $\boldsymbol{\delta}^{\text{hid}} \boldsymbol{x}^\top$. Setiap potong rumus matriks yang ringkas itu, ternyata, hanyalah pembukuan rapi dari aturan rantai skalar yang baru kita telusuri satu per satu.

4.5 Empat gradien, dirangkum

Itulah seluruh *backpropagation* untuk jaringan satu *hidden layer*. Mengejutkan betapa sedikit yang sebenarnya terjadi. Kotak berikut adalah resep lengkapnya—satu-satunya hal yang perlu diketahui kode kita di Bab 6.

Algoritma backpropagation (satu contoh)

Forward: $z = Wx + b,\ \ h = \sigma(z),\ \ u = Vh + c,\ \ \hat{y} = g(u).$

Backward:

$$\delta^{\text{out}} = \hat{y} - y,$$

$$\delta^{\text{hid}} = \left(V^{\top}\delta^{\text{out}}\right) \odot h \odot (\mathbf{1} - h).$$

Gradien:

$$\frac{\partial \mathcal{L}}{\partial V} = \delta^{\text{out}} h^{\top}, \qquad \frac{\partial \mathcal{L}}{\partial c} = \delta^{\text{out}},$$

$$\frac{\partial \mathcal{L}}{\partial W} = \delta^{\text{hid}} x^{\top}, \qquad \frac{\partial \mathcal{L}}{\partial b} = \delta^{\text{hid}}.$$

Perhatikan keindahan simetrinya. *Forward pass* mengalikan dengan W lalu V untuk mendorong nilai ke depan; *backward pass* mengalikan dengan $V^{\top}$ lalu $W^{\top}$ (dan menyelipkan σ') untuk menarik gradien ke belakang. Maju dan mundur adalah dua sisi dari koin yang sama, dengan bobot yang sama hanya ditranspos. Begitu Anda melihat pola ini, menambah lapisan ketiga, keempat, dan seterusnya hanyalah mengulang langkah yang sama—itulah mengapa menguasai satu *hidden layer* sudah membuka pintu ke jaringan sedalam apa pun.

4.6 Contoh terhitung: backprop dengan tangan

Mari lanjutkan jaringan mungil dari Bab 3 ($d = 2,\ m = 2,\ k = 1$). Di sana, untuk masukan $x = (1{,}0,\ 2{,}0)$ kita sudah menghitung forward pass-nya:

$$z = (0{,}0,\ 0{,}8), \qquad h = (0{,}500,\ 0{,}690), \qquad \hat{y} = u = 0{,}205.$$

Misalkan jawaban yang benar ternyata $y = 1{,}0$. Kita hitung keempat gradiennya.

Loss. $\mathcal{L} = \frac{1}{2}(0{,}205 - 1{,}0)^2 = \frac{1}{2}(0{,}632) = 0{,}316.$

Delta keluaran. $\delta^{\text{out}} = \hat{y} - y = 0{,}205 - 1{,}0 = -0{,}795$. Tandanya negatif, yang menandakan tebakan kita terlalu *rendah* dibanding target—petunjuk yang masuk akal.

Gradien lapisan keluaran. Dengan $\boldsymbol{h} = (0{,}500,\ 0{,}690)$,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}} = \delta^{\text{out}} \boldsymbol{h}^{\top} = -0{,}795\,(0{,}500,\ 0{,}690) = (-0{,}398,\ -0{,}549),$$
$$\frac{\partial \mathcal{L}}{\partial c} = \delta^{\text{out}} = -0{,}795.$$

Delta tersembunyi. Dengan $\boldsymbol{V} = (0{,}7,\ -0{,}5)$, mula-mula $\boldsymbol{V}^{\top}\delta^{\text{out}} = (0{,}7,\ -0{,}5) \cdot (-0{,}795) = (-0{,}557,\ 0{,}398)$. Lalu $\sigma'(z_p) = h_p(1 - h_p)$ memberi $\sigma'(z_1) = 0{,}5 \cdot 0{,}5 = 0{,}250$ dan $\sigma'(z_2) = 0{,}690 \cdot 0{,}310 = 0{,}214$. Kalikan per-elemen:

$$\boldsymbol{\delta}^{\text{hid}} = (-0{,}557,\ 0{,}398) \odot (0{,}250,\ 0{,}214) = (-0{,}139,\ 0{,}085).$$

Gradien lapisan tersembunyi. Dengan $\boldsymbol{x} = (1{,}0,\ 2{,}0)$,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} = \boldsymbol{\delta}^{\text{hid}} \boldsymbol{x}^{\top} = \begin{pmatrix} -0{,}139 \\ 0{,}085 \end{pmatrix} (1{,}0,\ 2{,}0) = \begin{pmatrix} -0{,}139 & -0{,}278 \\ 0{,}085 & 0{,}170 \end{pmatrix},$$
$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} = \boldsymbol{\delta}^{\text{hid}} = (-0{,}139,\ 0{,}085).$$

Dan selesai. Kita kini punya kemiringan loss terhadap kesembilan parameter—arah yang harus kita lawan untuk memperbaiki tebakan. Yang tersisa hanyalah: seberapa jauh melangkah, dan bagaimana mengulanginya ribuan kali. Itu pekerjaan Bab 5.

⇝ Intuisi

Anda baru saja menjalankan *backpropagation* secara penuh dengan tangan. Apa yang dilakukan TensorFlow di balik layar, untuk jutaan parameter sekaligus, *persis* ini—hanya jauh lebih cepat. Tidak ada keajaiban di dalam kotak hitam itu; hanya aturan rantai yang diterapkan rapi dan berulang.

4.7 Dari satu contoh ke seluruh data

Penurunan di atas memakai satu contoh $(\boldsymbol{x}, \boldsymbol{y})$. Untuk seluruh data berisi n contoh, loss totalnya adalah rata-rata loss tiap contoh (Bagian 4.1), dan karena turunan bersifat linear, gradien totalnya pun sekadar *rata-rata* gradien tiap contoh:

$$\frac{\partial \mathcal{L}_{\text{total}}}{\partial \boldsymbol{W}} = \frac{1}{n} \sum_{s=1}^{n} \frac{\partial \mathcal{L}^{(s)}}{\partial \boldsymbol{W}}, \tag{4.11}$$

dan serupa untuk $\boldsymbol{b}$, $\boldsymbol{V}$, $\boldsymbol{c}$. Secara praktis, kita tidak benar-benar menjalankan lingkaran satu per satu; di Bab 6 kita akan menyusun semua contoh sebagai baris-baris sebuah matriks dan menghitung seluruh gradien rata-rata sekaligus dengan beberapa perkalian matriks—ringkas dan cepat. Tetapi matematikanya tidak berubah sedikit pun dari yang baru kita turunkan.

4.8 Gradien sudah di tangan

Kita telah menempuh bagian tersulit sekaligus terpenting dari seluruh buku. Diberi sebuah contoh, kita kini tahu cara menghitung persis seberapa peka loss terhadap setiap bobot dan bias—lewat satu *forward pass* untuk mendapat nilai, lalu satu *backward pass* untuk mengalirkan gradien kembali.

Namun mengetahui arah belum berarti sampai di tujuan. Gradien hanya memberi tahu ke mana harus melangkah; ia tidak memberi tahu seberapa besar langkahnya, atau bagaimana mengulang langkah itu ribuan kali tanpa tersesat. Menerjemahkan gradien menjadi proses pelatihan yang benar-benar menurunkan loss—dari *gradient descent* polos sampai sentuhan *momentum* dan Adam—adalah isi Bab 5.

Soal Latihan

4.1 [Derivasi] Turunkan $\frac{\partial \mathcal{L}}{\partial \boldsymbol{c}} = \boldsymbol{\delta}^{\text{out}}$ secara lengkap, dimulai dari $u_j = \sum_p V_{jp} h_p + c_j$ dan aturan rantai.

4.2 [Hitung] Ulangi contoh terhitung di Bagian 4.6, tetapi dengan target $y = 0{,}0$ (bukan 1,0). Hitung $\mathcal{L}$, $\boldsymbol{\delta}^{\text{out}}$, $\boldsymbol{\delta}^{\text{hid}}$, dan keempat gradiennya.

4.3 [Derivasi] Untuk kasus dua keluaran ($k = 2$), tuliskan $\frac{\partial \mathcal{L}}{\partial h_p}$ secara eksplisit sebagai penjumlahan atas $j = 1, 2$, lalu nyatakan dalam bentuk matriks $\boldsymbol{V}^{\top} \boldsymbol{\delta}^{\text{out}}$.

4.4 [Konsep] Mengapa perhitungan $\frac{\partial \mathcal{L}}{\partial h_p}$ memerlukan *penjumlahan* atas semua neuron keluaran, bukan satu suku saja? Kaitkan dengan aturan rantai bercabang Bab 2.

4.5 [Derivasi] Periksa kecocokan dimensi setiap gradien di kotak algoritma backprop: pastikan $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}$ berukuran $m \times d$, $\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}}$ berukuran $k \times m$, dan seterusnya.

4.6 [Kode] *Gradient checking*: untuk satu bobot W_{pi}, taksir gradiennya secara

numerik dengan $\frac{\mathcal{L}(W_{pi}+\varepsilon)-\mathcal{L}(W_{pi}-\varepsilon)}{2\varepsilon}$ (mis. $\varepsilon = 10^{-5}$), lalu bandingkan dengan hasil `backward`. Keduanya harus nyaris sama—ini cara baku memverifikasi implementasi backprop.

5

Optimisasi: Melatih Jaringan

Di Bab 4 kita memperoleh hadiah besar: diberi sebuah contoh, kita bisa menghitung gradien loss terhadap setiap parameter. Gradien itu menunjuk arah—tetapi arah saja belum membawa kita ke tujuan. Bab ini menjawab dua pertanyaan praktis yang tersisa: *seberapa jauh* kita melangkah ke arah itu, dan bagaimana *mengulang* langkah itu ratusan atau ribuan kali sampai loss benar-benar mengecil. Proses berulang inilah yang kita sebut *pelatihan* (*training*), dan mesin di baliknya adalah *gradient descent*.

5.1 Dari gradien ke langkah: gradient descent

Ingat tafsir gradien dari Bab 2: $\nabla \mathcal{L}$ menunjuk arah *kenaikan* tercuram, sehingga $-\nabla \mathcal{L}$ menunjuk arah *penurunan* tercuram. Kalau kita ingin loss mengecil, kita cukup melangkah sedikit ke arah negatif gradien. Itulah seluruh isi *gradient descent*. Untuk parameter apa pun—kita pakai θ sebagai lambang umum yang mewakili $\boldsymbol{W}$, $\boldsymbol{b}$, $\boldsymbol{V}$, atau $\boldsymbol{c}$— aturan pembaruannya adalah

$$\theta \;\leftarrow\; \theta - \eta \,\frac{\partial \mathcal{L}}{\partial \theta}, \tag{5.1}$$

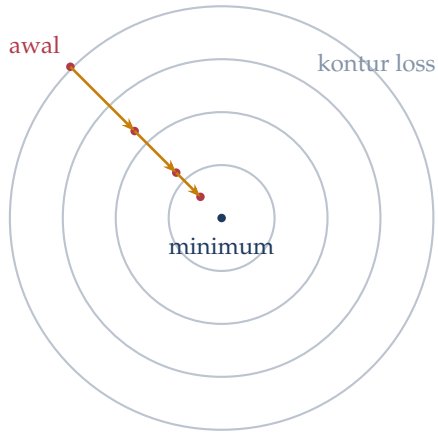
dengan $\eta > 0$ sebuah angka kecil yang disebut **learning rate** (laju belajar)—besar langkah kita. Diterapkan ke keempat parameter kita sekaligus:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \,\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}, \quad \boldsymbol{b} \leftarrow \boldsymbol{b} - \eta \,\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}}, \quad \boldsymbol{V} \leftarrow \boldsymbol{V} - \eta \,\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}}, \quad \boldsymbol{c} \leftarrow \boldsymbol{c} - \eta \,\frac{\partial \mathcal{L}}{\partial \boldsymbol{c}}.$$

⇝ Intuisi

Bayangkan seorang pendaki yang tersesat di kabut tebal dan ingin turun ke lembah. Ia tak bisa melihat jauh, tetapi bisa merasakan

kemiringan tanah di bawah kakinya. Strategi paling masuk akal: melangkah ke arah yang paling menurun, lalu meraba lagi, lalu melangkah lagi. Gradien adalah indra kemiringan itu; learning rate adalah panjang langkahnya. Gambar 5.1 menggambarkan turunannya pada permukaan loss berbentuk mangkuk.



Gambar 5.1: *Gradient descent* menuruni permukaan loss. Dari titik awal, tiap langkah bergerak melawan gradien (ke arah penurunan tercuram), makin lama makin pendek seiring mendekati dasar—karena gradien mengecil di sekitar minimum.

Contoh terhitung: satu langkah gradient descent

Tidak ada yang lebih meyakinkan daripada melihat loss benar-benar turun. Mari pakai gradien yang sudah kita hitung dengan tangan di Bab 4 untuk jaringan mungil kita. Di sana, untuk $\boldsymbol{x} = (1{,}0,\ 2{,}0)$ dengan target $y = 1{,}0$, tebakannya $\hat{y} = 0{,}205$ dan loss-nya $\mathcal{L} = 0{,}316$. Gradiennya:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} = \begin{pmatrix} -0{,}139 & -0{,}278 \\ 0{,}085 & 0{,}170 \end{pmatrix}, \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} = (-0{,}139,\ 0{,}085),$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}} = (-0{,}398,\ -0{,}549), \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}} = -0{,}795.$$

Ambil learning rate $\eta = 0{,}5$ dan terapkan Persamaan (5.1). Parameter lama bergeser menjadi

$$\boldsymbol{W} \leftarrow \begin{pmatrix} 0{,}570 & -0{,}161 \\ 0{,}758 & 0{,}115 \end{pmatrix}, \qquad \boldsymbol{b} \leftarrow (0{,}170,\ -0{,}443),$$

$$\boldsymbol{V} \leftarrow (0{,}899,\ -0{,}226), \qquad c \leftarrow 0{,}598.$$

Sekarang jalankan ulang *forward pass* dengan parameter baru ini (masukan sama, $\boldsymbol{x} = (1{,}0,\ 2{,}0)$):

$$\boldsymbol{z} = (0{,}417,\ 0{,}545), \quad \boldsymbol{h} = (0{,}603,\ 0{,}633), \quad \hat{y} = 0{,}997.$$

Tebakan melompat dari 0,205 menjadi 0,997—nyaris tepat sasaran 1,0! Loss-nya pun anjlok dari 0,316 menjadi sekitar 0,000005, praktis nol.

Catatan

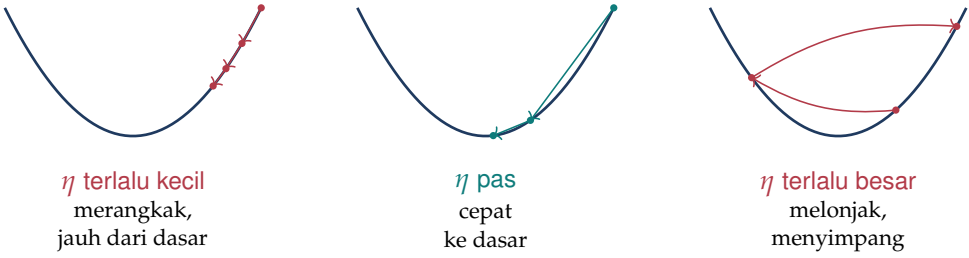
Penurunan sedramatis ini terjadi karena masalah mainan kita sangat kecil dan satu contoh saja. Pada data nyata dengan ribuan contoh dan parameter, satu langkah hanya menggeser loss sedikit; kemajuan datang dari *mengulang* langkah ribuan kali. Lagipula learning rate 0,5 kebetulan pas di sini—angka yang keliru bisa membuat segalanya kacau, seperti kita lihat berikut.

5.2 Learning rate: seberapa besar melangkah

Learning rate η adalah satu angka, tetapi memilihnya secara keliru bisa menggagalkan seluruh pelatihan. Ada tiga skenario, dirangkum pada Gambar 5.2. Jika η *terlalu kecil*, tiap langkah hanya beranjak sedikit, sehingga pelatihan merangkak lambat dan butuh sangat banyak iterasi. Jika η *pas*, kita meluncur ke dasar dalam beberapa langkah saja. Jika η *terlalu besar*, kita melompati dasar dan mendarat di seberang yang lebih tinggi—dan kalau terlalu parah, lompatan demi lompatan justru makin membesar sehingga loss *menyimpang* (*diverge*) menuju tak hingga, bukannya turun.

Catatan

Tidak ada nilai η ajaib yang berlaku untuk semua masalah. Memilihnya adalah seni praktis: orang biasanya mencoba beberapa nilai (misalnya 0,1; 0,01; 0,001) dan mengamati apakah loss turun mulus. Metode adaptif seperti Adam, yang kita bahas di akhir bab, sebagian



Gambar 5.2: Tiga nasib gradient descent menurut besar learning rate. Kiri: langkah terlalu pendek, merangkak. Tengah: pas, meluncur ke dasar. Kanan: langkah terlalu panjang, melompati dasar dan justru makin menjauh.

lahir untuk mengurangi kerepotan ini.

5.3 Batch, stochastic, dan mini-batch

Pertanyaan berikutnya: gradien dihitung dari *berapa banyak* contoh sebelum setiap pembaruan? Ada tiga pilihan, dan perbedaannya nyata dalam praktik.

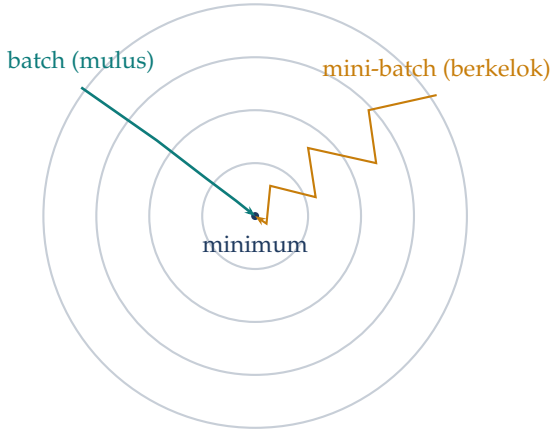
Batch gradient descent memakai *seluruh* n contoh untuk menghitung satu gradien rata-rata (Persamaan (4.11)), lalu melakukan satu pembaruan. Arahnya akurat dan mulus, tetapi tiap langkah mahal: pada data besar, kita harus mengolah jutaan contoh hanya untuk satu langkah kecil.

Stochastic gradient descent (SGD) [5] berada di ujung lain: ia memperbarui parameter setelah *setiap satu* contoh. Tiap langkah sangat murah dan kita melakukan n pembaruan dalam sekali lintasan data. Harganya, arah tiap langkah "berisik" (*noisy*) karena hanya didasarkan pada satu contoh, sehingga lintasannya berkelok-kelok.

Mini-batch gradient descent mengambil jalan tengah yang paling lazim dipakai: hitung gradien rata-rata dari sekelompok kecil contoh—sebuah *mini-batch* berukuran, katakanlah, 32 atau 64—lalu perbarui. Ia cukup murah, cukup stabil, dan cocok dengan cara perangkat keras modern bekerja. Gambar 5.3 membandingkan lintasan batch yang mulus dengan mini-batch yang berkelok namun tetap sampai ke tujuan.

⇝ Intuisi

Keberisikan SGD ternyata tidak selalu buruk. Sedikit keacakan justru bisa membantu jaringan "melompat keluar" dari cekungan dangkal



Gambar 5.3: Batch versus mini-batch. Batch (teal) menghitung arah dari seluruh data sehingga lintasannya mulus dan langsung. Mini-batch/SGD (amber) memakai sedikit contoh per langkah, jadi arahnya berisik dan berkelok—tetapi tiap langkah jauh lebih murah dan akhirnya tetap sampai ke minimum.

yang bukan minimum terbaik, sesuatu yang tak bisa dilakukan batch yang serba mulus. Itulah salah satu alasan mini-batch menjadi pilihan baku dalam praktik.

Satu istilah penting: satu kali lintasan penuh melalui seluruh data disebut satu **epoch**. Pelatihan biasanya berlangsung selama puluhan hingga ratusan epoch, dan di tiap epoch data diaduk ulang (*shuffle*) lalu dibagi menjadi mini-batch.

5.4 Lingkaran pelatihan lengkap

Sekarang semua kepingan bisa kita rangkai menjadi satu prosedur utuh. Kotak berikut adalah kerangka pelatihan yang akan kita kodekan langsung di Bab 6.

Lingkaran pelatihan (mini-batch gradient descent)

1. Inisialisasi $\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{V}, \boldsymbol{c}$ dengan angka acak kecil.
2. Ulangi selama sejumlah *epoch*:

(a) Aduk ulang urutan data.

(b) Untuk setiap mini-batch:

i. *Forward pass*: hitung $\hat{\boldsymbol{y}}$ untuk batch (Bab 3).
ii. Hitung loss (Bab 4).
iii. *Backward pass*: hitung gradien rata-rata (Bab 4).
iv. Perbarui tiap parameter: $\theta \leftarrow \theta - \eta \, \frac{\partial \mathcal{L}}{\partial \theta}$.

Itulah keseluruhan pelatihan sebuah *neural network*—sungguh tidak lebih rumit dari itu. Sisa buku ini hanyalah mewujudkan kerangka ini dalam kode (Bab 6) dan menerapkannya pada data nyata (Bab 8–10). Tetapi sebelum itu, ada dua perbaikan pada langkah pembaruan yang membuat pelatihan jauh lebih cepat dan mulus; keduanya pantas dikenal.

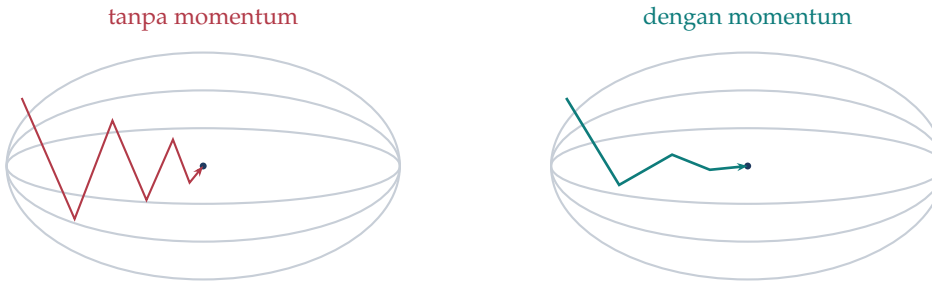
5.5 Momentum: meredam ayunan

Gradient descent polos punya satu kelemahan yang menjengkelkan. Bayangkan permukaan loss berbentuk lembah sempit memanjang—curam di arah melintang, landai di arah memanjang. Gradien di sana hampir selalu menunjuk melintang, ke dinding seberang, bukan ke arah dasar lembah yang sebenarnya kita tuju. Akibatnya lintasan memantul bolak-balik antar dinding sambil merangkak pelan sekali di arah memanjang. Gambar 5.4 (kiri) memperlihatkan pemborosan ini.

Momentum menyembuhkannya dengan satu gagasan sederhana: alih-alih melangkah murni menurut gradien saat ini, kita biarkan langkah-langkah sebelumnya menumpuk menjadi semacam "kecepatan" [4]. Kita simpan vektor kecepatan $\boldsymbol{v}$ (satu untuk tiap parameter) dan memperbaruinya sebagai

$$\boldsymbol{v} \;\leftarrow\; \beta\,\boldsymbol{v} + \frac{\partial \mathcal{L}}{\partial \theta}, \qquad \theta \;\leftarrow\; \theta - \eta\,\boldsymbol{v}, \tag{5.2}$$

dengan β (lazimnya 0,9) menentukan seberapa banyak kecepatan lama dipertahankan. Arah yang konsisten dari langkah ke langkah akan saling *menguatkan* dan mempercepat; arah yang bolak-balik akan saling *membatalkan* dan teredam. Hasilnya seperti bola berat yang menggelinding menuruni lembah: ia mengabaikan riak-riak kecil dan meluncur mantap ke dasar (Gambar 5.4, kanan).



Gambar 5.4: Momentum pada lembah sempit. Tanpa momentum (kiri, merah), lintasan memantul antar dinding curam sambil merangkak pelan ke dasar. Dengan momentum (kanan, teal), ayunan melintang teredam dan langkah di arah memanjang menumpuk, sehingga jaringan meluncur ke minimum jauh lebih cepat.

5.6 Adam, sekilas

Momentum mempercepat lewat satu kecepatan yang sama untuk semua. Pertanyaan berikutnya yang wajar: bisakah *setiap* parameter punya learning rate-nya sendiri, yang menyesuaikan diri secara otomatis? Itulah gagasan **Adam** [6] (*Adaptive Moment Estimation*), pengoptimal yang kini menjadi pilihan baku di banyak pustaka, termasuk yang akan kita pakai lewat TensorFlow.

Adam memadukan dua ingatan berjalan untuk tiap parameter: rata-rata gradien (seperti momentum) dan rata-rata gradien-kuadrat. Dengan menulis $g = \frac{\partial \mathcal{L}}{\partial \theta}$,

$$\begin{aligned} \boldsymbol{m} &\leftarrow \beta_1 \, \boldsymbol{m} + (1-\beta_1)\, g, && \text{(rata-rata gradien)} \\ \boldsymbol{s} &\leftarrow \beta_2 \, \boldsymbol{s} + (1-\beta_2)\, g^2, && \text{(rata-rata gradien kuadrat)} \\ \theta &\leftarrow \theta - \eta \, \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{s}}} + \epsilon}, && \end{aligned} \tag{5.3}$$

dengan $\hat{\boldsymbol{m}}$ dan $\hat{\boldsymbol{s}}$ versi $\boldsymbol{m}, \boldsymbol{s}$ yang dikoreksi-bias pada langkah-langkah awal, dan ϵ angka mungil penjaga pembagian. Nilai lazimnya $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$, $\epsilon = 10^{-8}$.

⇝ Intuisi

Bagian $\sqrt{\hat{\boldsymbol{s}}}$ di penyebut adalah inti kecerdikan Adam: parameter yang gradiennya sering besar dan bergejolak akan dibagi dengan angka besar, sehingga langkahnya diperkecil dan tidak liar; parameter yang

gradiennya kecil dan tenang dibiarkan melangkah lebih lebar. Dengan kata lain, tiap parameter mendapat learning rate efektifnya sendiri, disetel otomatis dari riwayat gradiennya.

Catatan

Di bab ini kita baru mengenal bentuk dan gagasan Adam. Mengimplementasikannya dari nol—bersama mini-batch dan momentum—kita kerjakan di Bab 7, lengkap dengan eksperimen yang mengadu kecepatan keempat pengoptimal. Dan saat beralih ke TensorFlow di Bab 8, kita tinggal memanggil Adam—kini tahu persis apa yang dikerjakannya.

5.7 Siap dikodekan

Lingkaran besarnya kini lengkap. Kita bisa menghasilkan tebakan (Bab 3), mengukur kesalahannya dan menghitung gradiennya (Bab 4), lalu memakai gradien itu untuk menggeser tiap parameter sedikit demi sedikit hingga loss mengecil (bab ini). Tiga roda gigi—*forward*, *backward*, dan *update*—yang berputar bersama itulah keseluruhan pembelajaran sebuah *neural network*.

Yang tersisa hanyalah menerjemahkan ketiganya ke dalam kode yang benar-benar berjalan. Di Bab 6 kita akan menulis seluruh jaringan dari nol dengan NumPy: sebuah kelas `NeuralNetwork` yang merangkum ketiga roda gigi tadi, lalu menyaksikannya belajar untuk pertama kali.

Soal Latihan

5.1 [Hitung] Ulangi contoh "satu langkah gradient descent" (Bagian 5.1) dengan learning rate $\eta = 0{,}1$ alih-alih 0,5. Berapa nilai $\hat{y}$ yang baru? Apakah loss turun lebih sedikit?

5.2 [Derivasi] Untuk $f(w) = w^2$, aturan GD memberi $w \leftarrow w - \eta \cdot 2w = (1 - 2\eta)w$. Untuk nilai η berapa proses ini *menyimpang* (nilai $|w|$ membesar)? Untuk berapa ia konvergen?

5.3 [Konsep] Bandingkan *batch*, *stochastic*, dan *mini-batch gradient descent* dari segi biaya per langkah, kemulusan arah, dan jumlah pembaruan per epoch.

5.4 [Derivasi] Bentangkan rumus momentum Persamaan (5.2) untuk tiga langkah pertama (mulai dari $\boldsymbol{v} = \mathbf{0}$), dan tunjukkan bahwa $\boldsymbol{v}$ adalah

jumlah berbobot dari gradien-gradien sebelumnya dengan bobot yang meluruh menurut β.

5.5 [Kode] Modifikasi lingkaran pelatihan agar memakai mini-batch: aduk data, lalu lakukan satu pembaruan per potongan berukuran B contoh.

5.6 [Konsep] Pada Adam, penyebut $\sqrt{\hat{s}} + \epsilon$ membagi langkah tiap parameter. Mengapa parameter dengan gradien yang besar dan bergejolak justru mendapat langkah yang lebih kecil?

6

Implementasi dari Nol dengan NumPy

Lima bab terakhir membangun seluruh teori: *forward pass* (Bab 3), *backpropagation* (Bab 4), dan *gradient descent* (Bab 5). Kini saatnya mengubah rumus menjadi kode yang benar-benar berjalan. Kita akan menulis sebuah jaringan satu *hidden layer* dari nol—hanya dengan NumPy [18], tanpa *framework*, tanpa autodiff—lalu menyaksikannya belajar untuk pertama kali.

Yang membuat bab ini memuaskan: hampir setiap baris kode di bawah punya pasangan langsung dengan persamaan yang sudah Anda turunkan sendiri. Tidak ada kotak hitam. Begitu jaringan ini bekerja, Anda akan tahu persis *mengapa* ia bekerja.

6.1 Satu penyesuaian: contoh sebagai baris

Sepanjang penurunan matematis, kita memakai vektor kolom untuk satu contoh: $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$. Dalam kode, jauh lebih praktis mengolah *seluruh batch* sekaligus. Maka kita susun n contoh sebagai *baris-baris* sebuah matriks $\boldsymbol{X}$ berukuran $n \times d$, dan keluaran pun menjadi matriks.

Akibatnya rumus per-contoh berubah sedikit bentuknya (bukan isinya). Karena masukan kini baris, perkalian $\boldsymbol{W}\boldsymbol{x}$ menjadi $\boldsymbol{X}\boldsymbol{W}^\top$ agar dimensinya cocok:

$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{W}^\top + \boldsymbol{b}, \qquad \boldsymbol{H} = \sigma(\boldsymbol{Z}), \qquad \boldsymbol{U} = \boldsymbol{H}\boldsymbol{V}^\top + \boldsymbol{c}.$$

Matriks bobot tetap kita simpan dalam orientasi yang sama seperti di buku ($\boldsymbol{W}$ berukuran $m \times d$, $\boldsymbol{V}$ berukuran $k \times m$), sehingga simbol di kode dan di rumus benar-benar sejajar. Satu-satunya tambahan adalah transpos $\boldsymbol{W}^\top$ dan $\boldsymbol{V}^\top$ saat mengalikan dengan batch berbentuk baris.

6.2 Blok demi blok

Sigmoid dan inisialisasi

Mulai dari fungsi aktivasi—terjemahan harfiah Persamaan (3.3)—dan konstruktor jaringan yang menyiapkan keempat parameter.

```
import numpy as np

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

class NeuralNetwork:
    def __init__(self, d, m, k, seed=0):
        r = np.random.default_rng(seed)
        self.W = r.normal(0, 0.5, size=(m, d))    # W : m x d
        self.b = np.zeros(m)
        self.V = r.normal(0, 0.5, size=(k, m))    # V : k x m
        self.c = np.zeros(k)
```

Perhatikan bobot diisi angka acak kecil, sedangkan bias cukup nol.

Catatan

Mengapa bobot tidak diinisialisasi nol saja? Karena kalau semua bobot sama, semua neuron tersembunyi akan menghitung hal yang persis sama dan menerima gradien yang persis sama pula—mereka tak akan pernah berdiferensiasi, dan jaringan lebar berperilaku seolah hanya punya satu neuron. Sedikit keacakan "mematahkan simetri" itu, memberi tiap neuron titik awal berbeda agar bisa mempelajari pola yang berbeda.

Forward pass

Berikutnya, terjemahan langsung dari kotak forward pass di Bab 3, kini dalam bentuk batch. Kita simpan $\boldsymbol{Z}$ dan $\boldsymbol{H}$ sebagai atribut karena keduanya dibutuhkan lagi saat *backward*.

```
    def forward(self, X):
        self.Z = X @ self.W.T + self.b      # (n, m)
        self.H = sigmoid(self.Z)            # (n, m)
        self.U = self.H @ self.V.T + self.c # (n, k)
        return self.U                       # keluaran linear
            (regresi)
```

Backward pass

Inilah inti bab ini—dan momen pembuktian Bab 4. Setiap baris di bawah adalah salah satu persamaan dari kotak algoritma *backpropagation*, ditulis untuk seluruh batch sekaligus. Pembagian dengan `n` mewujudkan perata-rataan gradien atas semua contoh (Persamaan (4.11)).

```
    def backward(self, X, Y):
        n = X.shape[0]
        dU = (self.U - Y) / n          # delta_out = (yhat - y)/n
        dV = dU.T @ self.H             # dL/dV = delta_out h^T
        dc = dU.sum(axis=0)            # dL/dc = delta_out
        dH = dU @ self.V               # dL/dh = V^T delta_out
        dZ = dH * self.H * (1 - self.H)# delta_hid = dL/dh . sigma'(z)
        dW = dZ.T @ X                  # dL/dW = delta_hid x^T
        db = dZ.sum(axis=0)            # dL/db = delta_hid
        return dW, db, dV, dc
```

Bandingkan baris demi baris dengan kotak di Bagian 4.5:

Kode ↔ rumus

`dU = (U - Y)/n`	$\boldsymbol{\delta}^{\text{out}} = \hat{\boldsymbol{y}} - \boldsymbol{y}$ (dirata-rata)
`dV = dU.T @ H`	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}} = \boldsymbol{\delta}^{\text{out}} \boldsymbol{h}^{\top}$
`dc = dU.sum(axis=0)`	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{c}} = \boldsymbol{\delta}^{\text{out}}$
`dH = dU @ V`	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}} = \boldsymbol{V}^{\top} \boldsymbol{\delta}^{\text{out}}$
`dZ = dH * H * (1 - H)`	$\boldsymbol{\delta}^{\text{hid}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}} \odot \sigma'(\boldsymbol{z})$
`dW = dZ.T @ X`	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} = \boldsymbol{\delta}^{\text{hid}} \boldsymbol{x}^{\top}$
`db = dZ.sum(axis=0)`	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} = \boldsymbol{\delta}^{\text{hid}}$

Penjumlahan `.sum(axis=0)` pada `dc` dan `db` adalah cara batch menjumlahkan kontribusi semua contoh untuk gradien bias; perkalian matriks seperti `dV = dU.T @ H` melakukan penjumlahan itu secara otomatis untuk gradien bobot.

Langkah pembaruan dan loss

Terakhir, satu langkah *gradient descent* (Persamaan (5.1)) dan fungsi loss MSE (Persamaan (4.1)).

```
    def step(self, grads, eta):
        dW, db, dV, dc = grads
        self.W -= eta * dW;  self.b -= eta * db
```

```
        self.V -= eta * dV;  self.c -= eta * dc

def loss_fn(Yhat, Y):
    return 0.5 * np.mean(np.sum((Yhat - Y)**2, axis=1))
```

Dan itulah seluruh jaringannya—tak sampai tiga puluh baris. Keempat metode (`__init__`, `forward`, `backward`, `step`) bersama `loss_fn` adalah perwujudan utuh dari teori lima bab terakhir.

6.3 Melatihnya: membengkokkan sebuah kurva

Mari kita uji pada masalah yang menunjukkan dengan jelas kemampuan jaringan menekuk garis: mempelajari fungsi $y = \sin x$ pada selang $[-3, 3]$ hanya dari contoh-contohnya. Ini regresi (keluaran tunggal, linear), persis kasus yang kita turunkan. Kita bangkitkan datanya, lalu jalankan *lingkaran pelatihan* dari Bab 5.

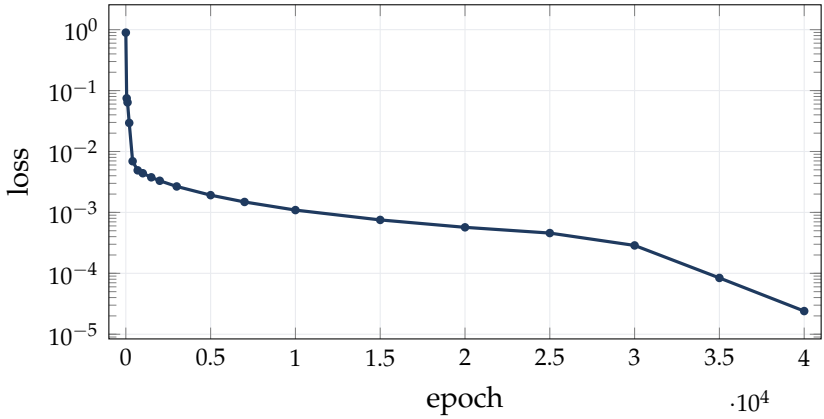
```
# data: 64 titik dari y = sin(x)
X = np.linspace(-3.0, 3.0, 64).reshape(-1, 1)    # (n, d)
Y = np.sin(X)                                     # (n, k)

net = NeuralNetwork(d=1, m=16, k=1, seed=1)       # 16 neuron tersembunyi
eta, epochs = 0.3, 40000

for epoch in range(epochs + 1):
    Yhat = net.forward(X)                         # 1) forward
    L    = loss_fn(Yhat, Y)                       # 2) ukur loss
    grads = net.backward(X, Y)                    # 3) backward
    net.step(grads, eta)                          # 4) update
```

Inilah ketiga roda gigi dari Bab 5—*forward*, *backward*, *update*—berputar bersama, satu putaran tiap epoch. (Di sini kita pakai *full-batch* demi kesederhanaan; memecahnya menjadi mini-batch hanya menambah satu lingkaran dalam.)

6.4 Hasil: loss yang turun dan kurva yang dipelajari

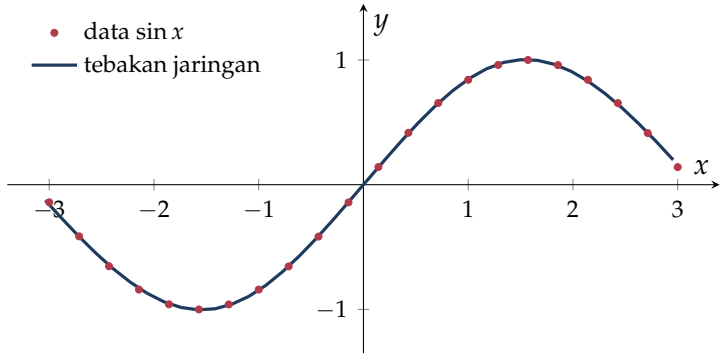
Saat kode ini dijalankan, loss-nya turun dari sekitar 0,89 di awal menjadi hanya 0,000024 setelah 40 000 epoch—penurunan lebih dari tiga orde besaran. Gambar 6.1 memperlihatkan perjalanannya pada skala logaritmik: terjun tajam di awal, lalu pengasahan yang sabar dan terus-menerus.

Tetapi angka loss saja kurang menggugah. Yang benar-benar memuaskan adalah melihat *apa* yang dipelajari jaringan. Gambar 6.2 menumpangkan



Gambar 6.1: Loss selama pelatihan (skala log). Dari $\approx 0{,}89$ menuju $\approx 0{,}00002$. Penurunan curam di ratusan epoch pertama lalu melandai—pola yang khas pada *gradient descent*.

tebakan jaringan di atas data $\sin x$ yang sesungguhnya. Garisnya nyaris berimpit—galat terbesar di seluruh selang hanya 0,019. Jaringan dengan satu *hidden layer* dan 16 neuron bersigmoid telah berhasil "menekuk" garis lurusnya menjadi gelombang sinus yang mulus, persis seperti dijanjikan teorema *universal approximation* di Bab 1.



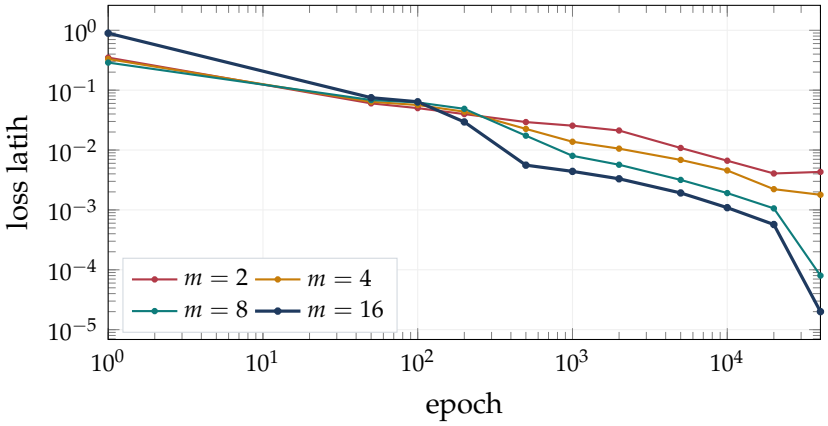
Gambar 6.2: Tebakan jaringan (garis biru) di atas data $\sin x$ (titik merah) setelah pelatihan. Keduanya nyaris berimpit; galat terbesar hanya 0,019. Inilah ketaklinearan sigmoid yang bekerja: garis lurus berhasil ditekuk menjadi gelombang.

⇝ Intuisi

Anda baru saja melatih sebuah *neural network* yang seluruh isinya Anda tulis sendiri dan pahami baris demi baris. Tidak ada satu pun bagian yang "ajaib": data masuk lewat `forward`, kesalahan dihitung lewat `loss_fn`, gradien mengalir mundur lewat `backward`, dan parameter bergeser lewat `step`—persis Bab 3, 4, dan 5 yang menjelma kode.

Pengaruh jumlah neuron tersembunyi

Berapa banyak neuron tersembunyi yang sebenarnya kita butuhkan? Ini menentukan *kapasitas* jaringan—seberapa rumit fungsi yang sanggup ia tiru. Gambar 6.3 mengulang pelatihan sinus dengan beberapa nilai m. Polanya jelas: makin banyak neuron, makin rendah loss akhir yang bisa dicapai. Dengan $m = 2$ jaringan terlalu sederhana dan "mentok" di loss yang relatif tinggi; dengan $m = 16$ ia menekan loss hingga nyaris nol.



Gambar 6.3: Kurva belajar pada masalah sinus untuk beberapa jumlah neuron tersembunyi m. Kapasitas yang lebih besar menekan loss akhir lebih rendah; $m = 2$ mentok di sekitar 0,004 sedangkan $m = 16$ menembus 10^{-5}.

Catatan

Lebih banyak neuron tidak selalu lebih baik. Pada masalah sinus yang bersih ini, kapasitas besar aman. Tetapi pada data nyata yang berderau, jaringan yang terlalu besar bisa mulai *menghafal* derau alih-

alih menangkap pola—masalah *overfitting* yang kita bahas di Bab 12.

6.5 Dari buatan tangan ke dunia nyata

Implementasi ini, betapapun kecilnya, sudah lengkap secara konseptual: ia memuat seluruh mekanisme yang juga dipakai jaringan raksasa. Tetapi ia masih punya beberapa keterbatasan praktis. Pertama, kita melatihnya dengan *full-batch gradient descent* yang paling polos—lambat, dan jauh dari cara orang melatih jaringan sungguhan. Kedua, kita memakainya pada data buatan yang bersih dan mungil; data nyata lebih berantakan dan butuh persiapan. Ketiga, untuk tugas selain regresi—misalnya klasifikasi—kita perlu mengganti "kepala" jaringan: fungsi keluaran *softmax* dan loss *cross-entropy*, yang belum kita turunkan.

Keterbatasan pertama kita atasi lebih dulu di Bab 7, dengan mengimplementasikan mini-batch, momentum, dan Adam dari nol—lalu mengadu kecepatannya pada masalah nyata. Setelah perkakas pelatihan kita lebih tajam, Bagian III menjawab dua keterbatasan sisanya: dimulai dari Bab 8, saat untuk pertama kalinya kita memakai data publik sungguhan—mengklasifikasikan spesies penguin—dan menurunkan softmax serta cross-entropy, lalu membandingkan jaringan buatan tangan kita dengan versi ringkas di atas TensorFlow.

Soal Latihan

6.1 [Kode] Tambahkan metode `predict(self, X)` pada kelas `NeuralNetwork` yang mengembalikan tebakan tanpa menyimpan keadaan antara. Untuk regresi ia cukup memanggil `forward`.

6.2 [Eksperimen] Latih ulang fit kurva sinus dengan jumlah neuron tersembunyi yang berbeda: $m = 2$, $m = 4$, dan $m = 32$. Bagaimana kualitas fit dan loss akhir berubah? Mengapa m yang terlalu kecil gagal?

6.3 [Eksperimen] Coba learning rate $\eta = 0{,}01$, $0{,}3$, dan $3{,}0$ pada masalah sinus. Mana yang lambat, mana yang pas, mana yang menyimpang?

6.4 [Konsep] Mengapa `forward` menyimpan $\boldsymbol{Z}$ dan $\boldsymbol{H}$ sebagai atribut? Apa yang terjadi pada `backward` bila keduanya tidak disimpan?

6.5 [Kode] Ubah lingkaran pelatihan menjadi mini-batch (Bab 5): pada tiap epoch, aduk indeks data lalu perbarui per potongan $B = 16$ contoh.

Bandingkan kurva loss-nya dengan versi *full-batch*.

6.6 [Kode] Terapkan *gradient checking* (Latihan 4.6) pada kelas NumPy untuk memverifikasi bahwa `backward` benar.

7

Optimisasi dari Nol: Mini-batch, Momentum, Adam

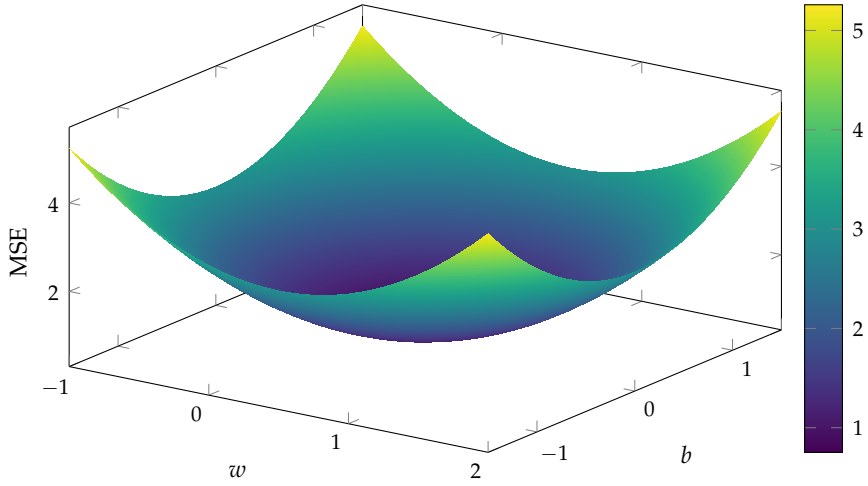
Di Bab 5 kita berkenalan dengan keluarga pengoptimal—mini-batch, momentum, dan Adam—tetapi hanya sebagai gagasan. Lalu di Bab 6 kita melatih jaringan pertama kita dengan bentuk paling polos: *full-batch gradient descent*. Bab ini menutup celah itu. Kita akan mengimplementasikan ketiga pengoptimal tersebut *dari nol*, lalu mengadu mereka pada masalah nyata yang sama untuk melihat dengan mata sendiri perbedaan kecepatan dan kestabilannya.

Ada satu pengamatan pembebas yang memandu seluruh bab ini: *gradiennya tidak pernah berubah*. *Backpropagation* dari Bab 4 tetap menghitung $\frac{\partial \mathcal{L}}{\partial \theta}$ dengan cara yang sama persis. Yang berbeda antar-pengoptimal hanyalah *bagaimana* gradien itu dipakai untuk menggeser parameter—yakni isi metode `step` saja. Maka kelas `NeuralNetwork` kita tidak perlu disentuh; cukup lingkaran pelatihannya yang kita ganti.

7.1 Mengapa pengoptimal penting: bentuk permukaan loss

Bayangkan loss sebagai permukaan di atas ruang parameter: tiap titik $(\theta_1, \theta_2, \dots)$ punya ketinggian $\mathcal{L}$. Melatih jaringan berarti menuruni permukaan itu menuju lembah terendah. Untuk masalah sederhana, permukaannya berupa mangkuk mulus seperti Gambar 7.1, dan gradient descent meluncur turun tanpa banyak drama.

Tetapi permukaan loss sebuah *neural network*—dengan ketaklinearan sigmoid dan puluhan hingga jutaan parameter—jauh lebih berliku: penuh lembah sempit, dataran landai, dan cekungan dangkal. Di medan seperti inilah pemilihan pengoptimal menentukan apakah pelatihan merangkak



Gambar 7.1: Permukaan loss nyata untuk regresi linear satu fitur (kekuatan beton terhadap kandungan semen), MSE sebagai fungsi kemiringan w dan pergeseran b. Untuk model selinear ini permukaannya berupa mangkuk cembung tunggal, dan *gradient descent* pasti meluncur ke dasarnya.

berhari-hari atau selesai dalam hitungan menit. Mari kita bangun tiga perbaikan atas GD polos, satu per satu.

7.2 Mini-batch dari nol

Ingat dari Bab 5: alih-alih menghitung gradien dari seluruh data tiap langkah (*full-batch*), kita pecah data menjadi potongan-potongan kecil dan memperbarui parameter setelah tiap potongan. Implementasinya hanya menambah satu fungsi pembantu yang menghasilkan mini-batch teracak, lalu satu lingkaran dalam.

```
def minibatches(X, Y, B, rng):
    idx = rng.permutation(len(X))          # aduk urutan tiap epoch
    for i in range(0, len(X), B):
        j = idx[i:i+B]
        yield X[j], Y[j]

def train_sgd(net, X, Y, eta=0.2, epochs=200, B=32):
    rng = np.random.default_rng(0)
    for epoch in range(epochs):
        for xb, yb in minibatches(X, Y, B, rng):
            net.forward(xb)
```

```
            net.step(net.backward(xb, yb), eta)    # step = GD
                polos
    return net
```

Perhatikan metode `step` dan `backward` sama persis seperti Bab 6. Yang baru hanyalah: gradien dihitung dari `xb`, `yb` (satu mini-batch) alih-alih seluruh data, dan ada banyak pembaruan per epoch.

7.3 Momentum dari nol

Momentum (Persamaan (5.2)) menambahkan satu "ingatan kecepatan" v untuk tiap parameter. Kita simpan v dalam sebuah *dictionary* yang berkaca pada parameter jaringan, lalu memperbaruinya sebelum tiap langkah.

```
def train_momentum(net, X, Y, eta=0.2, beta=0.9, epochs=200,
    B=32):
    rng = np.random.default_rng(0)
    keys = ["W", "b", "V", "c"]
    v = {k: np.zeros_like(getattr(net, k)) for k in keys}
    for epoch in range(epochs):
        for xb, yb in minibatches(X, Y, B, rng):
            net.forward(xb)
            g = dict(zip(keys, net.backward(xb, yb)))
            for k in keys:
                v[k] = beta * v[k] + g[k]              # v <- b
                    v + grad
                setattr(net, k, getattr(net, k) - eta * v[k])
    return net
```

Arah yang konsisten dari batch ke batch menumpuk di v dan mempercepat; arah yang bolak-balik saling meredam. Itulah "bola berat" dari Bab 5 dalam wujud kode.

7.4 Adam dari nol

Adam (Persamaan (5.3)) memerlukan dua ingatan per parameter—rata-rata gradien m dan rata-rata gradien-kuadrat s—plus koreksi-bias pada langkah-langkah awal. Semuanya muat dalam beberapa baris.

```
def train_adam(net, X, Y, eta=0.01, b1=0.9, b2=0.999, eps=1e-8,
               epochs=200, B=32):
    rng = np.random.default_rng(0)
    keys = ["W", "b", "V", "c"]
    m = {k: np.zeros_like(getattr(net, k)) for k in keys}
    s = {k: np.zeros_like(getattr(net, k)) for k in keys}
```

```
    t = 0
    for epoch in range(epochs):
        for xb, yb in minibatches(X, Y, B, rng):
            net.forward(xb)
            g = dict(zip(keys, net.backward(xb, yb)))
            t += 1
            for k in keys:
                m[k] = b1 * m[k] + (1 - b1) * g[k]
                s[k] = b2 * s[k] + (1 - b2) * g[k]**2
                mhat = m[k] / (1 - b1**t)            # koreksi
                    bias
                shat = s[k] / (1 - b2**t)
                step = eta * mhat / (np.sqrt(shat) + eps)
                setattr(net, k, getattr(net, k) - step)
    return net
```

Catatan

Ketiga fungsi pelatihan ini berbagi tulang punggung yang sama: hitung gradien lewat `net.backward` (tak berubah), lalu terapkan menurut aturan masing-masing. Itu menegaskan kembali pesan Bab 5—pengoptimal hanyalah *kebijakan langkah* yang berbeda di atas gradien yang sama.

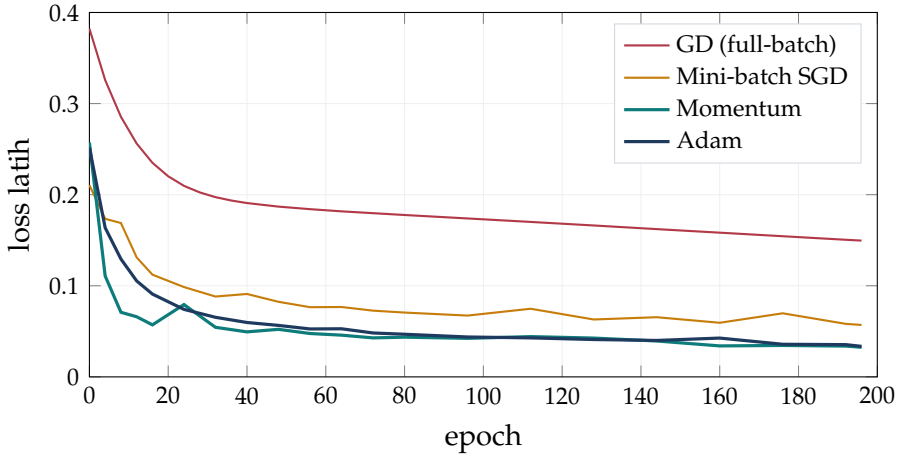
7.5 Adu cepat: empat pengoptimal pada satu masalah

Sekarang bagian yang menyenangkan. Kita latih jaringan $8 \to 16 \to 1$ yang *identik* pada data regresi beton (Bab 9) dengan keempat pengoptimal, masing-masing selama 200 epoch, lalu bandingkan penurunan loss latihnya. Hasilnya pada Gambar 7.2.

Perbedaannya tegas. Dalam anggaran 200 epoch yang sama, GD full-batch baru mencapai loss 0,15, sementara momentum dan Adam sudah turun di bawah 0,035. Tabel 7.1 merangkum angkanya, termasuk RMSE pada data uji dan epoch pertama saat loss latih menembus 0,05.

⇝ Intuisi

Tiga pelajaran terbaca langsung dari gambar. Pertama, beralih dari full-batch ke mini-batch saja sudah memberi lompatan terbesar—karena banyak pembaruan kecil mengalahkan sedikit pembaruan besar. Kedua, momentum dan Adam mempercepat lebih jauh dengan "mengingat" arah-arah sebelumnya. Ketiga, kecepatan ada harganya: lintasan



Gambar 7.2: Empat pengoptimal pada masalah dan jaringan yang sama. GD full-batch (merah) merangkak; mini-batch SGD (amber) jauh lebih cepat tetapi berderau; momentum (teal) dan Adam (biru) meluncur paling cepat dan paling mulus ke loss rendah.

mini-batch berderau, dan Adam menuntut *learning rate* yang jauh lebih kecil (0,01 lawan 0,2) agar tak liar.

7.6 Mana yang sebaiknya dipakai?

Tidak ada pemenang mutlak, tetapi ada panduan praktis yang masuk akal. Untuk sebagian besar masalah, **Adam dengan mini-batch** adalah titik awal yang baik: ia cepat, cukup tahan terhadap pilihan *learning rate*, dan jarang mengecewakan—itulah mengapa ia menjadi baku di TensorFlow dan PyTorch. **Mini-batch SGD dengan momentum** kadang menghasilkan model yang menggeneralisasi sedikit lebih baik dan disukai pada pelatihan skala besar. **Full-batch GD** jarang dipakai pada data besar, tetapi tetap berharga secara pedagogis—ia paling sederhana dan paling mudah dipahami, itulah mengapa kita memulai darinya di Bab 6.

7.7 Mesin yang sama, langkah yang lebih cerdas

Yang patut digarisbawahi: tak satu pun perbaikan di bab ini menyentuh *backpropagation*. Gradien yang mengalir mundur tetap sama persis seperti yang kita turunkan dengan tangan di Bab 4; kita hanya membuat langkah-

Pengoptimal	**Loss akhir**	**RMSE uji (MPa)**	**Epoch capai** 0,05
GD (full-batch)	0,150	8,88	— (tak tercapai)
Mini-batch SGD	0,057	6,04	—
Momentum	0,032	5,94	37
Adam	0,033	5,91	67

Tabel 7.1: Perbandingan empat pengoptimal pada regresi beton, 200 epoch, jaringan $8 \to 16 \to 1$ yang sama. Mini-batch jauh mengungguli full-batch; momentum dan Adam mengungguli keduanya dalam kecepatan maupun mutu akhir.

langkah yang memakainya menjadi lebih cerdas. Itu menegaskan pembagian kerja yang rapi di jantung *deep learning*: *backpropagation* menyediakan arah, pengoptimal memutuskan langkah.

Dengan perkakas yang kini lebih tajam, kita siap menghadapi tiga wajah dari satu model. Bab 8 membuka Bagian III dengan tugas pertama: mengklasifikasikan spesies penguin.

8

Klasifikasi: Spesies Penguin

Kita memasuki Bagian III, tempat satu jaringan yang sama dihadapkan pada tiga tugas berbeda. Bab ini yang pertama: *klasifikasi*—menebak *kategori*, bukan angka. Persisnya, kita akan menebak spesies seekor penguin (Adelie, Chinstrap, atau Gentoo) hanya dari empat ukuran tubuhnya.

Inilah saat benang merah buku ini menampakkan diri. Badan jaringan—masukan → *hidden layer* bersigmoid—tidak berubah sedikit pun dari Bab 6. Yang kita ganti hanya *kepala*-nya: fungsi keluaran menjadi *softmax* dan fungsi loss menjadi *cross-entropy*. Dan kejutan terbesarnya, yang akan kita buktikan, adalah bahwa meski kepalanya berbeda, sinyal yang dikirim mundur ke badan ternyata *sama persis* seperti pada regresi. Akibatnya kode `backward` kita tak perlu diubah satu baris pun.

8.1 Tugas baru: menebak kategori

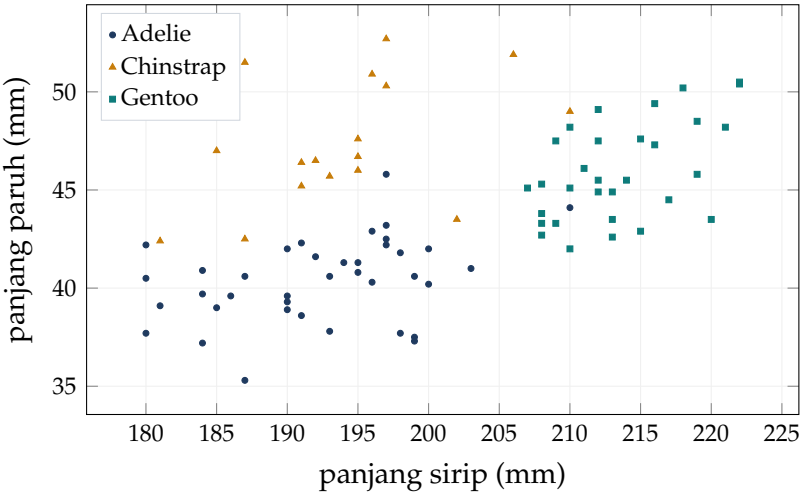
Pada regresi, keluaran adalah angka kontinu dan kita mengukur kesalahan dengan selisih kuadrat. Pada klasifikasi, keluaran adalah salah satu dari beberapa *kelas* yang tak punya urutan alami—Adelie bukan "lebih besar" dari Gentoo. Karena itu kita tidak menebak satu angka, melainkan *peluang* untuk tiap kelas: jaringan keluaran tiga angka yang menyatakan keyakinannya, misalnya 0,97 Adelie, 0,02 Chinstrap, 0,01 Gentoo. Kelas dengan peluang tertinggi menjadi tebakan akhir.

Agar peluang itu sah, dua syarat harus dipenuhi: tiap angka antara 0 dan 1, dan totalnya tepat 1. Lapisan keluaran linear biasa tidak menjamin keduanya. Kita butuh fungsi keluaran baru—softmax—untuk memaksakannya. Tetapi sebelum ke sana, mari berkenalan dengan datanya.

8.2 Data: Palmer Penguins

Palmer Penguins adalah dataset publik berisi pengukuran 344 ekor penguin dari tiga spesies di Kepulauan Palmer, Antartika. Untuk tiap penguin tersedia empat fitur numerik: panjang paruh, kedalaman paruh, panjang sirip, dan massa tubuh. Tugas kita: dari keempat angka itu, tebak spesiesnya.

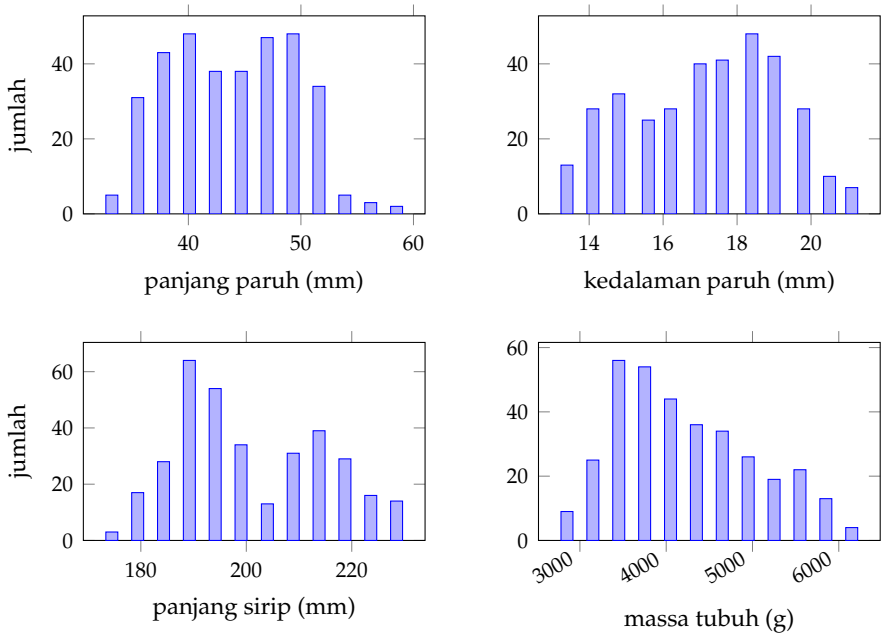
Gambar 8.1 memperlihatkan data pada dua dari empat fiturnya. Sudah dari dua fitur saja, ketiga spesies tampak mengelompok cukup rapi—pertanda tugas ini bisa dipelajari dengan baik. Gentoo menonjol dengan sirip panjang; Adelie berparuh pendek; Chinstrap di antaranya.



Gambar 8.1: Data Palmer Penguins pada dua fitur. Ketiga spesies membentuk gugus yang cukup terpisah—tugas yang menjanjikan untuk jaringan kita. (Model sebenarnya memakai keempat fitur sekaligus, bukan hanya dua yang tergambar di sini.)

Berguna pula melihat sebaran tiap fitur satu per satu. Gambar 8.2 memuat histogram keempatnya. Beberapa berbentuk condong atau bahkan dua-puncak (*bimodal*)—misalnya panjang sirip, yang puncak keduanya mencerminkan Gentoo yang bertubuh lebih besar. Sebaran berskala sangat berbeda (panjang paruh dalam puluhan milimeter, massa tubuh dalam ribuan gram) justru menegaskan perlunya standarisasi sebelum pelatihan.

Sebelum dipakai, data disiapkan dengan beberapa langkah lazim. Dua baris yang nilainya hilang dibuang (menyisakan 342 baris). Tiap fitur *distandarkan*—dikurangi reratanya lalu dibagi simpangan bakunya—agar semuanya berskala sebanding; ini penting karena sigmoid mudah jenuh bila



Gambar 8.2: Sebaran keempat fitur Palmer Penguins. Perhatikan bentuk dua-puncak pada panjang sirip dan massa tubuh, serta skala yang sangat berbeda antar-fitur.

masukannya berskala besar (massa tubuh dalam ribuan gram, misalnya). Label spesies di-*one-hot*-kan: Adelie menjadi $(1, 0, 0)$, Chinstrap $(0, 1, 0)$, Gentoo $(0, 0, 1)$. Terakhir, data dibagi 80% untuk latih dan 20% untuk uji.

8.3 Kepala baru: fungsi softmax

Lapisan tersembunyi tetap menghasilkan $\boldsymbol{h}$ seperti biasa, dan lapisan keluaran tetap menghitung pra-aktivasi $\boldsymbol{u} = \boldsymbol{V}\boldsymbol{h} + \boldsymbol{c}$ dengan $k = 3$ komponen—satu skor mentah per spesies. Yang baru adalah cara mengubah ketiga skor itu menjadi peluang. Fungsi **softmax** melakukannya:

$$\hat{y}_j \;=\; \text{softmax}(\boldsymbol{u})_j \;=\; \frac{e^{u_j}}{\sum_{l=1}^{k} e^{u_l}}. \tag{8.1}$$

Eksponensial menjamin tiap nilai positif, dan pembagian dengan jumlah seluruhnya menjamin totalnya 1. Skor terbesar mendapat porsi peluang terbesar—makin menonjol skornya, makin dekat peluangnya ke 1. Gambar 8.3 memperlihatkannya pada satu penguin uji sungguhan dari model terlatih kita.

Gambar 8.3: Softmax pada satu penguin uji. Skor mentah $\boldsymbol{u} = (4{,}92,\ -1{,}43,\ -4{,}11)$ diubah menjadi peluang $(0{,}998,\ 0{,}002,\ 0{,}000)$ yang menjumlah ke 1. Jaringan sangat yakin penguin ini Adelie—dan memang benar.

8.4 Loss baru: cross-entropy

Untuk regresi kita memakai selisih kuadrat. Untuk peluang, ukuran kesalahan yang tepat adalah **cross-entropy**. Dengan label one-hot $\boldsymbol{y}$ (bernilai 1 pada kelas benar c dan 0 selainnya), loss untuk satu contoh adalah

$$\mathcal{L} \;=\; -\sum_{j=1}^{k} y_j \log \hat{y}_j \;=\; -\log \hat{y}_c. \tag{8.2}$$

Karena $\boldsymbol{y}$ one-hot, semua suku lenyap kecuali untuk kelas benar, menyisakan $-\log \hat{y}_c$.

⇝ Intuisi

Cross-entropy menghukum *ketidakyakinan pada jawaban benar*. Bila jaringan memberi peluang $\hat{y}_c = 1$ pada kelas yang benar, $-\log 1 = 0$: tanpa kesalahan. Bila ia ragu dan memberi, katakanlah, $\hat{y}_c = 0{,}1$, maka $-\log 0{,}1 \approx 2{,}3$: hukuman besar. Dan bila ia yakin pada jawaban yang *salah* ($\hat{y}_c \to 0$), hukumannya meledak menuju tak hingga. Dengan kata lain: percaya dirilah, tetapi hanya kalau benar.

8.5 Gradien yang ternyata sama

Kini bagian yang elegan. Kepala kita berganti total—softmax, bukan identitas; cross-entropy, bukan MSE—jadi kita perlu menurunkan ulang sinyal mundur $\boldsymbol{\delta}^{\text{out}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{u}}$. Untungnya ada jalan pintas yang rapi. Tuliskan ulang loss dengan memasukkan Persamaan (8.1) ke (8.2):

$$\mathcal{L} = -\log \hat{y}_c = -\log \frac{e^{u_c}}{\sum_l e^{u_l}} = -u_c + \log \sum_l e^{u_l}.$$

Sekarang turunkan terhadap satu u_j. Suku pertama, $-u_c$, hanya bergantung pada u_j bila $j = c$, memberi -1 untuk $j = c$ dan 0 selainnya—yang persis $-y_j$. Suku kedua turunannya $e^{u_j} / \sum_l e^{u_l} = \hat{y}_j$. Maka

$$\boxed{\delta_j^{\text{out}} = \frac{\partial \mathcal{L}}{\partial u_j} = \hat{y}_j - y_j, \qquad \text{yakni} \qquad \boldsymbol{\delta}^{\text{out}} = \hat{\boldsymbol{y}} - \boldsymbol{y}.} \tag{8.3}$$

Bandingkan dengan Persamaan (4.2) dari kasus regresi: *bentuknya identik*. Inilah keindahan yang dijanjikan di Bab 4. Meski softmax dan cross-entropy jauh lebih rumit daripada keluaran linear dan MSE, keduanya "dirancang berpasangan" sehingga sinyal mundurnya menyederhana menjadi selisih yang sama, $\hat{\boldsymbol{y}} - \boldsymbol{y}$.

Rute kedua: Jacobian softmax penuh

Jalan pintas di atas memang elegan, tetapi sebagian pembaca mungkin ingin melihat penurunan yang lebih lugas—langsung melalui turunan softmax. Rute ini lebih panjang namun memperlihatkan mesinnya secara penuh, dan hasilnya tentu sama.

Mulai dari softmax $\hat{y}_l = e^{u_l} / S$ dengan $S = \sum_m e^{u_m}$. Kita perlu turunan tiap keluaran $\hat{y}_l$ terhadap tiap pra-aktivasi u_j. Ada dua kasus. Bila $l = j$, dengan aturan hasil-bagi,

$$\frac{\partial \hat{y}_j}{\partial u_j} = \frac{e^{u_j} S - e^{u_j} e^{u_j}}{S^2} = \frac{e^{u_j}}{S}\left(1 - \frac{e^{u_j}}{S}\right) = \hat{y}_j(1 - \hat{y}_j).$$

Bila $l \neq j$, pembilang e^{u_l} tak bergantung pada u_j, sehingga

$$\frac{\partial \hat{y}_l}{\partial u_j} = \frac{0 \cdot S - e^{u_l} e^{u_j}}{S^2} = -\frac{e^{u_l}}{S}\frac{e^{u_j}}{S} = -\hat{y}_l \hat{y}_j.$$

Kedua kasus dapat dipadatkan menjadi satu rumus, yang disebut *Jacobian softmax*:

$$\frac{\partial \hat{y}_l}{\partial u_j} = \hat{y}_l\big(\mathbb{1}[l = j] - \hat{y}_j\big), \tag{8.4}$$

dengan $\mathbb{1}[l = j]$ bernilai 1 jika $l = j$ dan 0 jika tidak.

Sekarang masukkan ini ke aturan rantai. Karena $\mathcal{L} = -\sum_l y_l \log \hat{y}_l$, kita punya $\frac{\partial \mathcal{L}}{\partial \hat{y}_l} = -y_l / \hat{y}_l$, sehingga

$$\frac{\partial \mathcal{L}}{\partial u_j} = \sum_l \frac{\partial \mathcal{L}}{\partial \hat{y}_l}\frac{\partial \hat{y}_l}{\partial u_j} = -\sum_l \frac{y_l}{\hat{y}_l}\,\hat{y}_l\big(\mathbb{1}[l = j] - \hat{y}_j\big) = -\sum_l y_l\big(\mathbb{1}[l = j] - \hat{y}_j\big).$$

Pisahkan penjumlahannya menjadi dua bagian:

$$\frac{\partial \mathcal{L}}{\partial u_j} = \underbrace{-\sum_l y_l \mathbb{1}[l=j]}_{=y_j} + \hat{y}_j \underbrace{\sum_l y_l}_{=1} = \hat{y}_j - y_j.$$

Suku pertama menyisakan y_j (hanya $l = j$ yang bertahan), dan suku kedua memakai fakta bahwa label one-hot menjumlah ke 1. Hasilnya kembali $\delta^{\text{out}} = \hat{y} - y$, persis seperti rute pintas—sebagaimana seharusnya.

⇝ Intuisi

Konsekuensinya luar biasa praktis: seluruh penurunan lapisan tersembunyi di Bab 4—rumus δ^{hid} dan keempat gradiennya—berlaku apa adanya, tanpa satu pun perubahan. Badan jaringan tidak tahu, dan tidak perlu tahu, kepala mana yang sedang dipasang. Itulah arti sesungguhnya dari "satu model, ujungnya yang ganti".

8.6 Kode: perubahan yang nyaris tak ada

Karena gradiennya sama, mengubah jaringan regresi Bab 6 menjadi pengklasifikasi hanya menuntut tiga sentuhan kecil. Pertama, fungsi softmax (dengan pengurangan nilai maksimum demi kestabilan numerik, agar e^u tak meledak):

```
def softmax(U):
    U = U - U.max(axis=1, keepdims=True)   # stabilitas numerik
    e = np.exp(U)
    return e / e.sum(axis=1, keepdims=True)
```

Kedua, `forward` menambahkan satu baris softmax di ujungnya:

```
    def forward(self, X):
        self.Z = X @ self.W.T + self.b
        self.H = sigmoid(self.Z)
        self.U = self.H @ self.V.T + self.c
        self.Yhat = softmax(self.U)        # <-- satu-satunya
            tambahan
        return self.Yhat
```

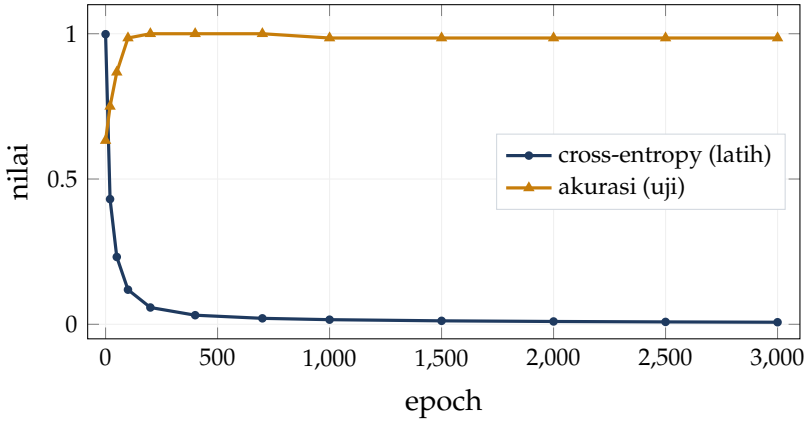
Ketiga, fungsi loss menjadi cross-entropy:

```
def cross_entropy(Yhat, Y):
    return -np.mean(np.sum(Y * np.log(Yhat + 1e-9), axis=1))
```

Dan `backward`? Tidak berubah sama sekali—ia tetap menghitung `dU = (self.Yhat - Y) / n`, persis seperti regresi, karena Persamaan (8.3) menjamin itu benar. Inilah bukti kode dari benang merah kita.

8.7 Hasil di NumPy

Kita latih jaringan dengan $d = 4$ masukan, $m = 8$ neuron tersembunyi, dan $k = 3$ keluaran, memakai *gradient descent* selama 3000 epoch. Cross-entropy pada data latih turun dari $\approx 1{,}0$ menjadi 0,007, sementara akurasi pada data uji yang belum pernah dilihat naik dengan cepat melampaui 98% (Gambar 8.4).



Gambar 8.4: Pelatihan pada Palmer Penguins. Cross-entropy latih (biru) terjun menuju nol seiring akurasi uji (amber) memanjat melampaui 98% hanya dalam ratusan epoch.

Akurasi akhir: 100% pada data latih dan **98,53**% pada data uji— 67 dari 68 penguin uji ditebak benar. Satu-satunya kesalahan terlihat pada *confusion matrix* berikut: seekor Chinstrap keliru ditebak sebagai Adelie (keduanya memang berukuran mirip), selebihnya tepat.

8.8 Versi TensorFlow

Setelah membangunnya dari nol, mari lihat betapa ringkasnya tugas yang sama bila ditulis di atas TensorFlow [19]. Seluruh jaringan—lapisan, fungsi keluaran, loss, dan optimisasi—muat dalam segelintir baris:

```
1 from tensorflow import keras
2
```

	tebakan		
sebenarnya	Adelie	Chinstrap	Gentoo
Adelie	24	0	0
Chinstrap	1	20	0
Gentoo	0	0	23

Tabel 8.1: *Confusion matrix* pada 68 penguin uji. Angka di luar diagonal adalah kesalahan; di sini hanya satu.

```
model = keras.Sequential([
    keras.layers.Input(shape=(4,)),
    keras.layers.Dense(8, activation="sigmoid"),    # hidden layer
    keras.layers.Dense(3, activation="softmax"),    # output layer
])
model.compile(optimizer="adam",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
model.fit(Xtr, Ytr, epochs=200, batch_size=32)
```

Tidak ada `forward`, tidak ada `backward`, tidak ada aturan pembaruan—semuanya ditangani di balik layar. Tetapi kini Anda tahu persis apa yang dikerjakan tiap baris: `Dense(8, "sigmoid")` adalah $\sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ kita; `Dense(3, "softmax")` adalah lapisan keluaran kita; `"categorical_crossentropy"` adalah Persamaan (8.2); dan `fit` menjalankan lingkaran pelatihan Bab 5, kali ini dengan optimisasi Adam alih-alih *gradient descent* polos.

Hasilnya pun sebanding. Tabel 8.2 menyandingkan keduanya: akurasi uji yang sama persis, jumlah parameter yang sama persis—karena memang arsitektur yang sama—hanya kode yang jauh lebih singkat dan optimisasi yang lebih canggih.

⇝ Intuisi

Inilah imbalan dari kerja keras enam bab pertama. Bagi kebanyakan orang, TensorFlow adalah kotak hitam ajaib. Bagi Anda, ia kini transparan: setiap pemanggilan fungsi punya pasangan di matematika dan kode yang Anda tulis sendiri. Anda memakai *framework* bukan karena terpaksa percaya, melainkan karena tahu persis apa yang dipercayakan.

	NumPy (dari nol)	TensorFlow
Arsitektur	$4 \to 8 \to 3$	$4 \to 8 \to 3$
Jumlah parameter	67	67
Optimisasi	gradient descent	Adam
Akurasi uji	98,53%	98,53%
Baris kode inti	≈ 25	≈ 8

Tabel 8.2: Jaringan buatan tangan versus TensorFlow pada tugas yang sama. Jumlah parameter 67 cocok dengan rumus Bab 3: $m(d{+}1) + k(m{+}1) = 8 \cdot 5 + 3 \cdot 9 = 67$.

8.9 Satu wajah selesai

Wajah pertama—klasifikasi—telah tuntas, dan benang merahnya terbukti bukan sekadar slogan: badan jaringan dari Bab 6 kita pakai apa adanya, hanya kepalanya yang berganti menjadi softmax dan cross-entropy, dan bahkan sinyal mundurnya tetap $\hat{\boldsymbol{y}} - \boldsymbol{y}$ yang sama.

Wajah kedua menanti di Bab 9: *regresi*, menebak kekuatan tekan beton dari komposisinya. Di sana kepalanya kembali ke keluaran linear dan loss MSE—persis yang kita turunkan di Bab 4—sehingga, jika benang merah ini benar, perpindahannya akan terasa nyaris tanpa usaha.

Soal Latihan

8.1 [Derivasi] Turunkan Jacobian softmax: $\frac{\partial \hat{y}_l}{\partial u_j} = \hat{y}_l(\mathbb{1}[l = j] - \hat{y}_j)$. Lalu pakai untuk menurunkan ulang $\delta^{\text{out}} = \hat{\boldsymbol{y}} - \boldsymbol{y}$ melalui $\frac{\partial \mathcal{L}}{\partial u_j} = -\sum_l (y_l / \hat{y}_l) \frac{\partial \hat{y}_l}{\partial u_j}$, sebagai alternatif jalan pintas di Bagian 8.5.

8.2 [Hitung] Hitung softmax dari skor $\boldsymbol{u} = (2,\ 1,\ -1)$. Pastikan ketiga hasilnya menjumlah ke 1.

8.3 [Hitung] Untuk tebakan $\hat{\boldsymbol{y}} = (0{,}7,\ 0{,}2,\ 0{,}1)$ dan label benar kelas pertama ($\boldsymbol{y} = (1, 0, 0)$), hitung cross-entropy $\mathcal{L} = -\log \hat{y}_c$.

8.4 [Konsep] Jelaskan mengapa kode `backward` tidak berubah sama sekali saat berpindah dari regresi ke klasifikasi, padahal fungsi keluaran dan loss-nya berbeda total.

8.5 [Eksperimen] Latih ulang pengklasifikasi Penguins hanya dengan dua fitur (mis. panjang sirip dan panjang paruh). Seberapa turun akurasinya

dibanding memakai keempat fitur?

8.6 [Kode] Tulis fungsi yang membangun *confusion matrix* dari label sebenarnya dan tebakan, seperti Tabel 8.1.

9

Regresi: Kekuatan Tekan Beton

Wajah kedua adalah *regresi*: menebak sebuah angka kontinu, bukan kategori. Tugas konkret kita di bab ini adalah memperkirakan *kekuatan tekan* sebatang beton—berapa megapascal tekanan yang sanggup ditahannya sebelum hancur—hanya dari komposisi campuran dan umurnya.

Jika Bab 8 memperkenalkan kepala yang sama sekali baru (softmax dan cross-entropy), bab ini melakukan kebalikannya: ia justru memamerkan betapa *sedikit* yang perlu berubah. Sebab kepala untuk regresi—keluaran linear dengan loss MSE— adalah persis yang kita turunkan di Bab 4 dan kodekan di Bab 6. Maka sebagian besar bab ini bukan tentang hal baru, melainkan tentang menyaksikan benang merah bekerja: badan yang sama, kepala lama yang dipasang kembali, dan data nyata yang baru.

9.1 Data: Concrete Compressive Strength

Dataset *Concrete Compressive Strength* [22] memuat 1030 contoh campuran beton. Untuk tiap campuran tercatat delapan fitur numerik—banyaknya semen, terak (*slag*), abu terbang (*fly ash*), air, *superplasticizer*, agregat kasar, agregat halus (semuanya dalam kg per m^3), serta umur beton dalam hari—dan satu target: kekuatan tekan dalam megapascal (MPa), yang pada data ini berkisar dari sekitar 2 sampai 83 MPa.

Tugas ini menjadi contoh regresi yang bersih karena kedelapan fiturnya sudah berupa angka (tak ada kategori untuk di-*encode*), targetnya intuitif, dan hubungan antara komposisi dan kekuatan bersifat tak-linear—persis jenis pola yang dijanjikan bisa ditangkap oleh satu *hidden layer*.

9.2 Yang tidak berubah: kepala lama dipasang kembali

Inilah inti pesan bab ini. Untuk berpindah dari klasifikasi ke regresi, kita membatalkan dua perubahan yang dibuat di Bab 8 dan kembali ke pengaturan asli Bab 6:

- Fungsi keluaran g kembali menjadi *identitas*: $\hat{\boldsymbol{y}} = \boldsymbol{u}$. Tidak ada softmax—keluaran dibiarkan berupa satu angka mentah, bebas bernilai berapa pun, persis seperti yang dituntut sebuah ramalan kekuatan.
- Fungsi loss kembali menjadi *MSE* (Persamaan (4.1)), bukan cross-entropy.

Dan karena di Bab 4 kita sudah membuktikan bahwa untuk pasangan keluaran-linear dan MSE, sinyal mundurnya $\boldsymbol{\delta}^{\text{out}} = \hat{\boldsymbol{y}} - \boldsymbol{y}$—bentuk yang sama yang ternyata juga berlaku untuk softmax dan cross-entropy—maka *seluruh* kode jaringan, dari `forward` sampai `backward`, sama persis dengan kelas `NeuralNetwork` regresi dari Bab 6. Tak ada satu baris pun yang perlu ditulis ulang. Kita hanya perlu memuat data baru dan menjalankannya.

⇝ Intuisi

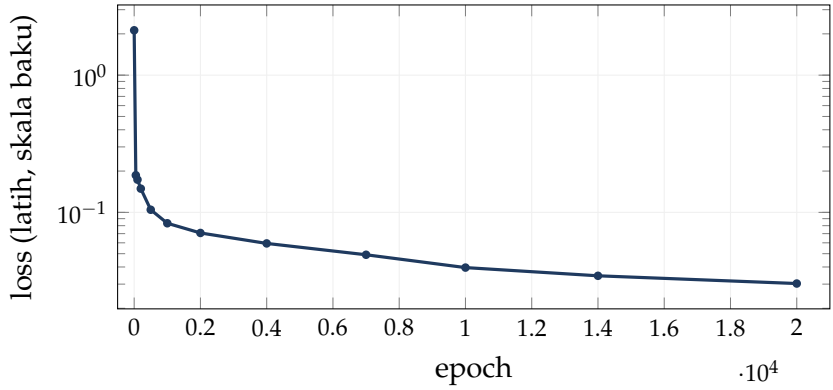
Perhatikan apa yang baru saja terjadi. Kita berpindah dari menebak *spesies* ke menebak *kekuatan beton*—dua dunia yang tampak tak berhubungan—tanpa mengubah arsitektur maupun algoritma sama sekali. Hanya data dan kepala yang berganti. Itulah daya ungkit dari memahami satu model secara tuntas.

9.3 Persiapan data

Seperti pada Penguins, fitur-fiturnya distandarkan (dikurangi rerata, dibagi simpangan baku) memakai statistik dari data latih saja. Karena ini regresi, kita *juga* menstandarkan targetnya: kekuatan dalam MPa diubah ke skala baku saat pelatihan, lalu tebakan jaringan dikembalikan ke satuan MPa ketika dievaluasi. Menstandarkan target membuat skala loss dan gradien wajar, sehingga satu *learning rate* bisa bekerja baik. Seperti biasa, data dibagi 80% latih dan 20% uji.

9.4 Hasil di NumPy

Kita latih jaringan $8 \to 16 \to 1$ (delapan masukan, enam belas neuron tersembunyi, satu keluaran linear) dengan *gradient descent*. Loss MSE pada data latih turun mulus sepanjang pelatihan, seperti tampak pada Gambar 9.1.



Gambar 9.1: Loss pelatihan pada data Concrete (skala log; target terstandar). Terjun tajam di awal lalu perbaikan yang sabar—pola *gradient descent* yang sudah akrab.

Tetapi untuk regresi, cara terbaik menilai mutu adalah membandingkan tebakan dengan nilai sebenarnya secara langsung. Gambar 9.2 memplot kekuatan yang *ditebak* jaringan terhadap kekuatan *sebenarnya* untuk tiap beton di data uji. Titik yang jatuh tepat pada garis diagonal berarti tebakan sempurna; sebaran di sekitar garis itu adalah galatnya. Tampak jelas titik-titik berkerumun rapat di sekitar diagonal sepanjang seluruh rentang 10–80 MPa.

9.5 Mengukur regresi: RMSE dan R^2

Berbeda dari klasifikasi yang punya ukuran tunggal nan jelas (akurasi), mutu regresi lazim diringkas dengan dua angka.

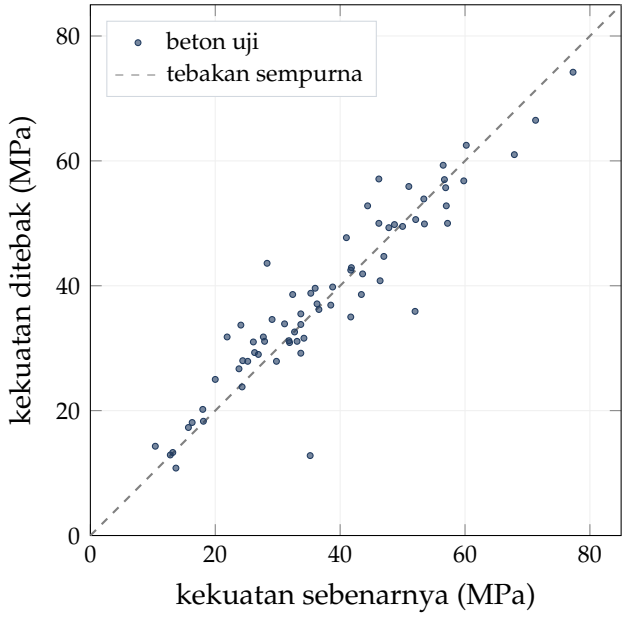
Dua ukuran regresi

RMSE (*root mean squared error*) adalah akar dari rata-rata kuadrat galat,

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_s (\hat{y}_s - y_s)^2},$$

dalam satuan yang sama dengan target (MPa). Ia menyatakan besar galat tipikal.

R^2 (koefisien determinasi) menyatakan proporsi ragam target yang berhasil dijelaskan model: 1 berarti sempurna, 0 berarti tak lebih baik daripada sekadar menebak rerata.



Gambar 9.2: Kekuatan ditebak versus sebenarnya pada data uji. Garis putus-putus adalah tebakan sempurna; titik yang dekat ke garis adalah tebakan baik. Jaringan melacak kekuatan beton dengan rapat di seluruh rentang.

Pada data uji, jaringan kita mencapai **RMSE** 5,55 **MPa** dan $\mathbf{R^2}$ 0,882—artinya galat tipikalnya sekitar 5,5 MPa, dan ia menjelaskan sekitar 88% ragam kekuatan beton. Untuk sebuah jaringan dengan satu *hidden layer* pada data nyata yang berderau, ini hasil yang baik.

9.6 Versi TensorFlow

Berpindah ke TensorFlow pun nyaris tanpa perubahan dari Bab 8. Hanya dua hal yang berbeda: lapisan keluaran kini `Dense(1)` tanpa aktivasi (keluaran linear), dan loss-nya `"mse"`.

```
from tensorflow import keras

model = keras.Sequential([
    keras.layers.Input(shape=(8,)),
    keras.layers.Dense(16, activation="sigmoid"),   # hidden layer
    keras.layers.Dense(1),                          # keluaran linear
])
```

```
model.compile(optimizer="adam", loss="mse")
model.fit(Xtr, Ytr, epochs=1500, batch_size=32)
```

Hasil keduanya sebanding, seperti dirangkum Tabel 9.1: sama-sama menjelaskan sekitar 87–88% ragam kekuatan, dengan galat tipikal di kisaran 5,5–6 MPa. Selisih kecil di antaranya hanyalah akibat perbedaan optimisasi dan lamanya pelatihan, bukan perbedaan kemampuan—arsitekturnya, dan karenanya 161 parameternya, identik.

	NumPy (dari nol)	**TensorFlow**
Arsitektur	$8 \to 16 \to 1$	$8 \to 16 \to 1$
Jumlah parameter	161	161
Optimisasi	gradient descent	Adam
RMSE uji (MPa)	5,55	5,86
R^2 uji	0,882	0,869

Tabel 9.1: Regresi beton: jaringan buatan tangan versus TensorFlow. Jumlah parameter 161 cocok dengan rumus Bab 3: $m(d{+}1) + k(m{+}1) = 16 \cdot 9 + 1 \cdot 17 = 161$.

9.7 Dua wajah selesai

Dua dari tiga wajah kini terbukti. Yang paling berkesan dari bab ini justru *ketiadaan* kejutan: berpindah dari klasifikasi penguin ke regresi beton tidak menuntut satu pun gagasan baru, hanya kepala lama yang dipasang kembali pada badan yang sama. Bila Bab 8 menunjukkan bahwa kepala bisa diganti, Bab 9 menunjukkan bahwa menggantinya kembali sama mudahnya.

Tinggal satu wajah: *time series forecasting* di Bab 10. Sekilas ia tampak paling berbeda—bukankah meramal masa depan menuntut model yang mengingat masa lalu? Kita akan melihat bahwa, dengan satu trik penyusunan data yang sederhana, tugas itu pun menjelma menjadi regresi biasa yang persis bisa ditangani jaringan yang sama ini.

Soal Latihan

9.1 [Konsep] Mengapa pada regresi kita menstandarkan *target*, sedangkan pada klasifikasi tidak? Apa yang bisa terjadi pada pelatihan bila target berskala besar (mis. ribuan) dibiarkan apa adanya?

9.2 [Hitung] Diberi lima tebakan dan nilai sebenarnya (dalam MPa): $(30, 32)$, $(40, 38)$, $(25, 27)$, $(50, 55)$, $(35, 33)$, hitung RMSE-nya.

9.3 [Hitung] Dengan data Latihan 2, hitung pula R^2. (Petunjuk: bandingkan jumlah kuadrat galat dengan jumlah kuadrat simpangan terhadap rerata nilai sebenarnya.)

9.4 [Konsep] Apa beda MSE dan MAE (*mean absolute error*) sebagai ukuran galat? Yang mana lebih peka terhadap pencilan (*outlier*), dan mengapa?

9.5 [Eksperimen] Latih ulang regresi beton dengan $m = 4$, $m = 16$, dan $m = 64$ neuron tersembunyi. Bagaimana R^2 uji berubah? Adakah tanda *overfitting* saat m besar?

9.6 [Kode] Tambahkan *early stopping*: sisihkan sebagian data latih sebagai validasi, dan hentikan pelatihan saat loss validasi berhenti membaik.

10

Time Series: Meramal Suhu

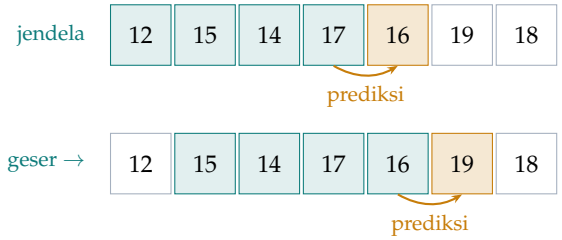
Wajah ketiga, dan yang paling menantang tampilannya: *time series forecasting*—meramal nilai masa depan sebuah deret waktu. Tugas konkret kita: memperkirakan suhu minimum esok hari dari catatan suhu hari-hari sebelumnya.

Sekilas ini terasa berbeda dari dua wajah sebelumnya. Bukankah meramal masa depan menuntut model yang *mengingat* masa lalu—sesuatu seperti jaringan berulang (*recurrent*) yang punya memori internal? Bab ini menunjukkan bahwa, dengan satu trik penyusunan data yang sangat sederhana, deret waktu menjelma menjadi tabel regresi biasa—dan jaringan satu *hidden layer* yang sama, tanpa memori apa pun, sudah bisa meramalkannya dengan baik.

10.1 Trik jendela geser

Kuncinya adalah *jendela geser* (*sliding window*). Alih-alih memperlakukan deret sebagai satu kesatuan, kita potong-potong menjadi banyak contoh kecil: ambil L nilai berturut-turut sebagai *masukan*, dan jadikan nilai *berikutnya* sebagai *target*. Lalu geser jendela satu langkah dan ulangi. Gambar 10.1 menggambarkannya.

Begitu data tersusun seperti ini, kita sudah punya tabel regresi biasa: tiap baris adalah vektor masukan berdimensi L, dengan satu target kontinu. Tidak ada lagi yang istimewa tentang "waktu"—bagi jaringan, ini sekadar regresi dengan L fitur. Dan regresi sudah kita kuasai sepenuhnya di Bab 9.



Gambar 10.1: Jendela geser dengan $L = 4$ (digambar kecil demi kejelasan; model kita memakai $L = 14$). Empat nilai berturut-turut (teal) menjadi masukan, nilai berikutnya (amber) menjadi target. Lalu jendela bergeser satu langkah, menciptakan contoh baru. Sebuah deret panjang dengan demikian melahirkan ribuan pasangan (masukan, target).

⇝ Intuisi

Inilah penutup benang merah buku ini. Klasifikasi, regresi, time series—ketiganya ternyata satu tugas yang sama bagi jaringan kita, hanya dengan kemasan data dan kepala yang berbeda. Time series bahkan tak butuh kepala khusus: jendela geser mengubahnya menjadi regresi, lalu kepala regresi dari Bab 9 menanganinya apa adanya.

10.2 Data: suhu harian Melbourne

Kita pakai deret *Daily Minimum Temperatures*—suhu minimum harian di Melbourne, Australia, selama sepuluh tahun (1981–1990), berisi 3650 pengamatan dalam derajat Celsius. Deret ini menunjukkan pola musiman yang jelas (dingin di pertengahan tahun, hangat di ujung tahun, mengikuti musim belahan bumi selatan) berbalut derau harian yang cukup besar.

Dengan jendela $L = 14$ (dua minggu ke belakang untuk meramal satu hari ke depan), deret 3650 titik itu berubah menjadi 3636 contoh, masing-masing dengan 14 fitur dan satu target.

10.3 Satu aturan penting: jangan mengacak waktu

Pada Penguins dan beton, kita mengacak data sebelum membaginya menjadi latih dan uji. Untuk deret waktu, itu *kesalahan serius*. Bila kita mengacak, sebagian contoh dari masa depan akan bocor ke data latih, dan jaringan akan dinilai pada hari-hari yang "tetangga waktunya" sudah ia lihat—memberi gambaran mutu yang palsu dan terlalu optimistis.

Karena itu kita membagi secara *kronologis*: 80% pertama (tahun-tahun awal) untuk latih, 20% terakhir (tahun-tahun akhir) untuk uji. Jaringan dilatih hanya pada masa lalu, lalu diuji kemampuannya meramal masa depan yang betul-betul belum pernah disentuh—persis situasi peramalan sungguhan.

Catatan

Aturan "jangan mengacak" ini berlaku umum untuk semua data berurut-waktu. Bocornya informasi masa depan ke masa lalu (*data leakage*) adalah salah satu jebakan paling sering dan paling menipu dalam peramalan: akurasi tampak gemilang saat pengembangan, lalu runtuh di dunia nyata.

10.4 Baseline: tebakan paling malas

Sebelum menilai jaringan, kita perlu pembanding. Untuk deret waktu, pembanding paling jujur adalah tebakan *persistensi* [25]: "ramalkan suhu esok sama dengan suhu hari ini". Tebakan malas ini mengejutkan sulit dikalahkan, karena suhu memang berubah perlahan dari hari ke hari. Sebuah model baru layak disebut berguna bila ia bisa mengalahkan persistensi.

Pada data uji, persistensi mencapai RMSE 2,48 °C. Itulah ambang yang harus dilewati jaringan kita.

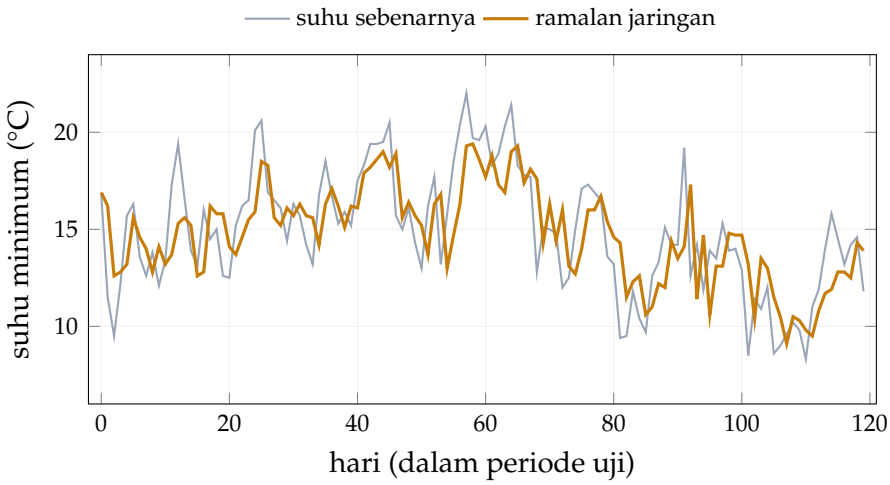
10.5 Hasil di NumPy

Kita latih jaringan $14 \to 16 \to 1$ dengan *gradient descent*—arsitektur yang sama, kelas `NeuralNetwork` regresi yang sama dari Bab 6 dan 8, hanya dengan dimensi masukan 14. Pada data uji ia mencapai RMSE 2,24 °C, mengungguli baseline persistensi (2,48 °C). Jaringan benar-benar belajar sesuatu yang melampaui sekadar mengulang nilai kemarin.

Gambar 10.2 menumpangkan ramalan jaringan di atas suhu sebenarnya untuk 120 hari pertama data uji. Garis ramalan mengikuti naik-turun suhu dengan cukup setia, meski—sebagaimana wajar untuk cuaca—ia melembutkan lonjakan harian yang paling tajam.

Catatan

Berbeda dari kurva sinus mulus di Bab 6 yang bisa di-*fit* nyaris sempurna, loss di sini berhenti turun di nilai yang jauh dari nol. Itu bukan



Gambar 10.2: Ramalan satu-hari-ke-depan (amber) versus suhu sebenarnya (biru) untuk 120 hari pertama periode uji. Jaringan menangkap irama naik-turunnya, sambil melembutkan puncak dan lembah paling ekstrem—bagian cuaca yang memang tak terprediksi.

kegagalan: cuaca harian mengandung keacakan yang *tak tereduksi*—tak ada model, betapa pun canggih, yang bisa meramal suhu esok dengan tepat. Tugas kita bukan mencapai galat nol, melainkan mendekati batas keterprediksian itu, dan mengalahkan tebakan malas seperti persistensi.

10.6 Versi TensorFlow

Seperti dapat diduga sekarang, versi TensorFlow nyaris tak berbeda dari regresi beton di Bab 9—hanya dimensi masukannya yang berubah menjadi 14:

```
from tensorflow import keras

model = keras.Sequential([
    keras.layers.Input(shape=(14,)),                # jendela 14 hari
    keras.layers.Dense(16, activation="sigmoid"),
    keras.layers.Dense(1),                          # ramalan suhu
])
```

```
model.compile(optimizer="adam", loss="mse")
model.fit(Xtr, Ytr, epochs=300, batch_size=32)
```

Hasilnya kembali sebanding, seperti dirangkum Tabel 10.1: keduanya mencapai RMSE sekitar 2,2 °C dan sama-sama mengalahkan persistensi, dengan 257 parameter yang identik.

Model	RMSE uji (°C)
Persistensi (baseline)	2,48
NumPy (dari nol, $14 \to 16 \to 1$)	2,24
TensorFlow ($14 \to 16 \to 1$)	2,20

Tabel 10.1: Peramalan suhu: jaringan (buatan tangan dan TensorFlow) mengalahkan baseline persistensi. Jumlah parameter 257 cocok dengan rumus Bab 3: $m(d{+}1) + k(m{+}1) = 16 \cdot 15 + 1 \cdot 17 = 257$.

10.7 Tiga wajah, satu model

Benang merah buku ini kini terbukti sepenuhnya. Kita memakai *satu* jaringan feedforward dengan satu *hidden layer* bersigmoid untuk tiga tugas yang tampak sangat berbeda:

- **Klasifikasi** (Bab 8): kepala softmax + cross-entropy, menebak spesies penguin.
- **Regresi** (Bab 9): kepala linear + MSE, menebak kekuatan beton.
- **Time series** (bab ini): jendela geser + kepala regresi, meramal suhu.

Badannya tidak pernah berubah. Yang berganti hanya kemasan data di depan dan kepala di belakang—dan bahkan sinyal mundur yang menghubungkan keduanya, $\delta^{\text{out}} = \hat{\boldsymbol{y}} - \boldsymbol{y}$, tetap sama di seluruh tugas. Inilah inti yang ingin disampaikan buku ini: sekali Anda benar-benar memahami satu model, Anda tidak mempelajari tiga hal yang berbeda, melainkan satu hal yang sama dalam tiga samaran.

Sejauh ini kita baru meramal satu deret dari masa lalunya sendiri. Bab 11 memperdalam wajah ketiga ini ke kasus *multivariat*— meramal dari masa lalu banyak variabel sekaligus—dan akan kita lihat bahwa jendela geser yang sama menanganinya pula. Lalu bagian terakhir buku, Bab 12, melangkah mundur untuk merangkum kiat-kiat praktis yang membuat pelatihan berjalan mulus, menjelaskan mengapa orang akhirnya beralih ke jaringan

yang lebih dalam dan aktivasi selain sigmoid, dan menunjuk ke mana Anda bisa melangkah setelah menutup buku ini.

Soal Latihan

10.1 [Konsep] Mengapa data deret waktu *tidak boleh* diacak sebelum dibagi latih/uji? Jelaskan apa itu *data leakage* dalam konteks ini.

10.2 [Hitung] Diberi deret $[10, 12, 11, 14, 13, 16]$ dan jendela $L = 3$, tuliskan semua pasangan (masukan, target) yang dihasilkan jendela geser.

10.3 [Eksperimen] Latih ulang peramal suhu dengan panjang jendela $L = 3$, $L = 7$, dan $L = 30$. Bagaimana RMSE uji berubah? Apakah jendela yang lebih panjang selalu lebih baik?

10.4 [Konsep] Apa itu baseline *persistensi*, dan mengapa sebuah model peramalan baru layak disebut berguna hanya bila mengalahkannya?

10.5 [Kode] Ubah peramalan menjadi *multi-langkah*: ramalkan dua hari ke depan dengan memberi makan kembali ramalan hari pertama sebagai masukan untuk ramalan hari kedua. Bagaimana galatnya menumpuk?

11

Time Series Multivariat: Banyak Deret, Satu Ramalan

Bab 10 meramal sebuah deret dari masa lalunya sendiri—satu variabel saja. Tetapi dunia nyata jarang sesederhana itu. Polusi udara hari esok tidak hanya bergantung pada polusi kemarin, melainkan juga pada angin, suhu, tekanan udara, dan hujan. Bab ini memperluas peramalan ke kasus *multivariat*: meramal satu target dari masa lalu *banyak* variabel sekaligus.

Penting ditegaskan di muka: ini bukan "wajah keempat". Ia hanyalah pendalaman wajah ketiga. Seperti akan kita lihat, trik jendela geser yang sama mengubah persoalan ini pun menjadi regresi biasa—hanya dengan masukan yang lebih lebar. Jaringan yang kita pakai tetap jaringan yang itu-itu juga.

11.1 Jendela geser, kini untuk banyak deret

Pada kasus univariat, jendela berisi L nilai masa lalu dari satu deret. Pada kasus multivariat, di tiap langkah waktu kita punya *vektor* berisi F fitur—misalnya $(\text{polusi}, \text{suhu}, \text{angin}, \dots)$. Jendela sepanjang L langkah kini memuat $L \times F$ angka. Kita cukup *meratakannya* (*flatten*) menjadi satu vektor masukan berdimensi $L \cdot F$, lalu memperlakukannya persis seperti regresi biasa:

$$\underbrace{\left[\, \boldsymbol{x}_{t-L},\, \boldsymbol{x}_{t-L+1},\, \dots,\, \boldsymbol{x}_{t-1} \,\right]}_{L \times F \text{ angka, diratakan}} \longrightarrow \hat{y}_t.$$

Itu saja perubahannya. Dimensi masukan d membengkak dari L menjadi $L \cdot F$, tetapi badan jaringan, *backpropagation*, dan kepalanya (regresi linear + MSE) tak berubah sama sekali. Bahkan kode penyusun jendela hanya

berganti satu baris: alih-alih mengiris satu kolom, kita iris dan ratakan seluruh blok.

```
def make_windows_multi(S, L):      # S: (n, F) -- banyak deret
    X = np.stack([S[i:i+L, :].ravel() for i in range(len(S)-L)])
    y = S[L:, 0:1]                 # kolom 0 = target (mis.
        polusi)
    return X, y
```

11.2 Seberapa jauh ke depan? Horizon ramalan

Ada satu sumbu baru yang patut diperhatikan: *horizon* H, yakni berapa langkah ke depan yang kita ramal. Meramal *satu jam* ke depan ($H = 1$) sangat berbeda dari meramal *sehari* ke depan ($H = 24$). Untuk horizon pendek, nilai target saat ini sudah hampir cukup—besok pagi mirip pagi ini. Tetapi makin jauh ke depan, nilai sekarang makin tak relevan, dan di situlah variabel-variabel lain mulai berbicara: angin kencang malam ini menyapu polusi besok, tekanan udara menandai datangnya front cuaca. Kita akan melihat justru pada horizon panjang keunggulan multivariat paling terasa.

11.3 Data: kualitas udara Beijing

Kita pakai dataset polusi udara Beijing [24]: pengukuran *per jam* selama lima tahun (2010–2014), berisi sekitar 43 800 baris. Targetnya konsentrasi PM2.5 (partikel halus, dalam $\mu g/m^3$); fiturnya, selain PM2.5 itu sendiri, mencakup titik embun, suhu, tekanan udara, kecepatan angin kumulatif, jam bersalju, dan jam hujan—tujuh deret numerik seluruhnya. Nilai PM2.5 yang hilang diisi dari tetangga waktunya, lalu seperti biasa data dibagi secara *kronologis* (Bab 10): 80% awal untuk latih, 20% akhir untuk uji.

Dengan jendela $L = 8$ jam dan $F = 7$ fitur, tiap contoh memiliki $8 \times 7 = 56$ nilai masukan. Jaringannya $56 \to 32 \to 1$. Karena datanya besar, kita melatih dengan *mini-batch Adam*—persis pengoptimal yang kita bangun di Bab 7.

11.4 Apakah variabel tambahan menolong?

Untuk menjawabnya secara jujur, kita bandingkan tiga peramal pada beberapa horizon: *persistensi* (ramalkan target H jam ke depan sama dengan nilai sekarang), jaringan *univariat* (jendela PM2.5 saja), dan jaringan *multivariat* (jendela ketujuh fitur). Hasilnya pada Tabel 11.1.

Horizon	Persistensi	Univariat	Multivariat
$H = 1$ jam	22,0	21,5	21,5
$H = 12$ jam	82,5	73,5	**71,4**
$H = 24$ jam	99,4	84,9	**84,0**

Tabel 11.1: RMSE uji (dalam $\mu g/m^3$) untuk peramalan PM2.5 pada tiga horizon. Pada $H = 1$ ketiganya nyaris seri; pada horizon panjang, multivariat unggul dan keduanya jauh mengalahkan persistensi.

Polanya bercerita banyak. Pada $H = 1$ jam, ketiga peramal nyaris seri—nilai PM2.5 sekarang sudah hampir menjelaskan satu jam ke depan, dan variabel cuaca menambah sedikit saja. Tetapi pada $H = 12$ jam, jurangnya melebar: persistensi melonjak ke 82,5, jaringan univariat menekannya ke 73,5, dan jaringan multivariat turun lebih jauh ke 71,4. Variabel cuaca, yang tak berguna untuk satu jam ke depan, menjadi berharga untuk dua belas jam ke depan—persis seperti diduga.

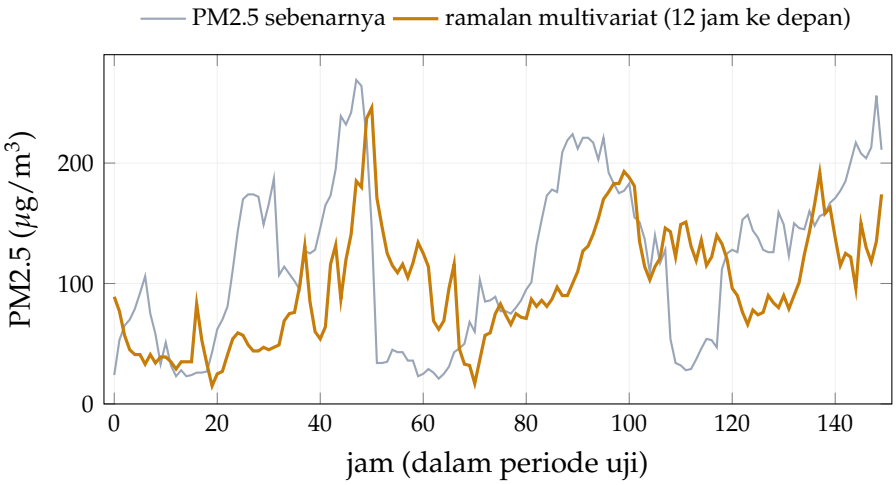
⇝ Intuisi

Inilah pelajaran jujur dari bab ini: variabel tambahan *tidak selalu* menolong, dan seberapa besar manfaatnya bergantung pada apa yang ditanyakan. Untuk horizon pendek, masa lalu target sudah cukup; untuk horizon panjang, konteks dari variabel lain mulai menentukan. Bagian dari menjadi praktisi yang baik adalah mengukur—bukan mengandaikan—apakah kerumitan tambahan benar-benar berbayar.

Gambar 11.1 memperlihatkan ramalan multivariat 12 jam ke depan terhadap nilai sebenarnya untuk sepenggal periode uji. Ramalannya menangkap naik-turun besar konsentrasi polusi, meski—wajar untuk horizon sepanjang itu— ia melewatkan sebagian lonjakan paling tajam.

11.5 Tetap mesin yang sama

Sekali lagi benang merah terbukti lentur. Kita berpindah dari satu deret ke tujuh, dari meramal satu jam ke meramal sehari penuh, dengan satu-satunya perubahan struktural berupa masukan yang lebih lebar—56 angka alih-alih 8. Tidak ada arsitektur baru, tidak ada algoritma baru; jendela geser tetap menerjemahkan waktu menjadi tabel regresi, dan jaringan satu *hidden layer* yang sama menanganinya. Wajah ketiga ternyata punya kedalaman yang



Gambar 11.1: Ramalan polusi multivariat 12 jam ke depan (amber) versus nilai sebenarnya (biru) untuk 150 jam pertama periode uji. Jaringan menangkap pola besar naik-turunnya polusi, meski melewatkan sebagian puncak paling tajam yang sulit diramal sejauh dua belas jam.

jauh melampaui contoh pertamanya, namun tetap dapat ditangani mesin yang itu-itu juga.

Dengan ini lengkap sudah penjelajahan kita atas penerapan jaringan. Bab terakhir, Bab 12, melangkah mundur untuk merangkum kiat praktis dan menunjuk ke mana Anda bisa melangkah setelah menutup buku ini.

12

Kiat Praktis dan Langkah Berikutnya

Kita telah membangun sebuah *neural network* dari nol, memahaminya baris demi baris, dan menerapkannya pada tiga tugas nyata. Bab penutup ini melangkah mundur sejenak untuk dua hal. Pertama, kiat-kiat praktis yang membuat pelatihan berjalan mulus—hal-hal yang sudah diam-diam kita pakai dan kini pantas diberi nama. Kedua, peta ringkas tentang mengapa dunia *deep learning* melangkah jauh melampaui satu *hidden layer* bersigmoid kita, dan ke mana Anda bisa melanjutkan.

12.1 Kiat praktis yang sudah kita pakai

Beberapa keputusan yang tampak sepele sepanjang buku sebenarnya menentukan berhasil-tidaknya pelatihan.

Standarisasi masukan. Pada Penguins, beton, dan suhu, kita selalu mengurangi rerata dan membagi simpangan baku tiap fitur. Ini bukan formalitas: sigmoid mudah *jenuh* bila masukannya berskala besar (massa tubuh dalam ribuan gram akan mendorong pra-aktivasi ke ekor datar sigmoid, tempat gradien nyaris nol). Menstandarkan menjaga semua nilai di kisaran tempat jaringan masih bisa belajar.

Inisialisasi bobot. Kita memulai bobot dengan angka acak kecil, bukan nol (Bab 6), untuk mematahkan simetri. Versi yang lebih cermat dari gagasan ini— menyetel skala acaknya menurut banyaknya masukan tiap neuron, dikenal sebagai inisialisasi *Xavier* [8] atau *He* [9]—membantu sinyal mengalir stabil di jaringan yang lebih dalam.

Learning rate. Seperti dilihat di Bab 5, inilah satu tombol paling berpengaruh. Terlalu kecil: lambat. Terlalu besar: menyimpang. Dalam praktik

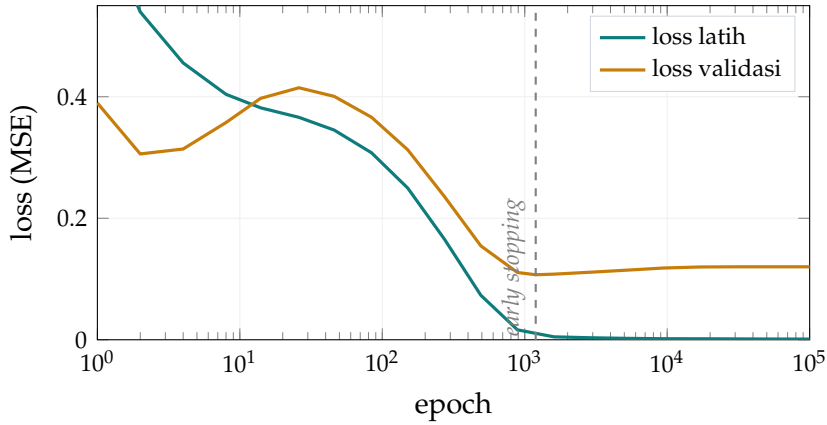
orang mencoba beberapa nilai, dan sering *menurunkannya* secara bertahap selama pelatihan (*learning rate schedule*) agar langkah mengecil saat mendekati dasar.

Mini-batch dan pemantauan. Memecah data menjadi mini-batch (Bab 5) mempercepat dan menstabilkan pelatihan. Dan selalu pantau loss pada data *validasi* yang terpisah, bukan hanya data latih—karena keduanya bisa bercerita sangat berbeda, seperti kita lihat berikut.

12.2 Overfitting: ketika menghafal mengalahkan memahami

Ada satu bahaya yang mengintai setiap model yang cukup lentur: *overfitting*. Jaringan dengan banyak parameter bisa saja *menghafal* data latih—termasuk deraunya—alih-alih menangkap pola yang sesungguhnya. Loss latihnya terus turun, tetapi kemampuannya pada data baru justru memburuk.

Gambar 12.1 memperlihatkannya pada percobaan sungguhan: sebuah jaringan berkapasitas besar dilatih pada sedikit titik berderau. Loss latih (teal) turun tanpa henti menuju nol, tetapi loss validasi (amber) mula-mula turun lalu *berbalik naik*. Selisih yang melebar di antara keduanya itulah tanda khas overfitting: jaringan kian pandai pada soal yang sudah dilihatnya, kian buruk pada yang belum.



Gambar 12.1: Overfitting nyata. Loss latih (teal) terus turun, tetapi loss validasi (amber) berbalik naik setelah titik tertentu. Garis putus-putus menandai saat loss validasi terendah—tempat ideal untuk berhenti.

Beberapa penangkalnya, dari yang paling sederhana:

- **Lebih banyak data.** Obat paling ampuh: makin banyak contoh, makin sulit menghafal dan makin terpaksa jaringan menangkap pola sejati.

- **Berhenti lebih awal** (*early stopping*). Hentikan pelatihan di titik loss validasi terendah, sebelum jaringan mulai menghafal derau— persis garis putus-putus pada gambar.
- **Regularisasi.** Tambahkan hukuman pada bobot yang besar (*weight decay* / regularisasi L2) agar jaringan memilih solusi yang lebih sederhana dan mulus.
- **Dropout** [10]**.** Selama latihan, matikan acak sebagian neuron tiap langkah, memaksa jaringan tidak bergantung berlebihan pada satu jalur.

⇝ Intuisi

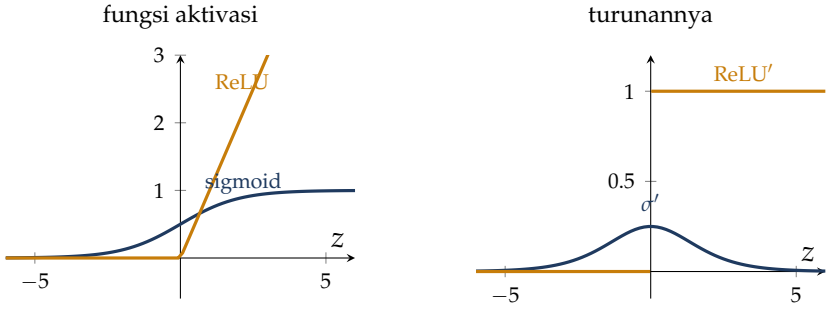
Tanda samar overfitting sebenarnya sudah muncul di Bab 8: akurasi uji Penguins sempat menyentuh 100% lalu turun ke 98,53% saat pelatihan berlanjut, padahal loss latih terus mengecil. Itulah jaringan mulai mengasah diri pada kekhususan data latih yang tak menggeneralisasi.

12.3 Mengapa orang meninggalkan sigmoid

Sepanjang buku, sigmoid melayani kita dengan baik—turunannya yang manis membuat *backpropagation* mudah diturunkan dan dipahami. Tetapi dalam praktik modern, sigmoid jarang dipakai di lapisan tersembunyi, dan alasannya bisa kita lihat langsung dari turunannya.

Perhatikan Gambar 12.2 (kiri): di kedua ekornya, sigmoid mendatar, sehingga turunannya (kanan) anjlok mendekati nol. Saat *backpropagation* melewatkan gradien mundur melalui banyak lapisan, ia mengalikannya dengan turunan-turunan kecil ini berulang kali; hasilnya gradien menyusut menuju nol sebelum sampai ke lapisan-lapisan awal. Inilah masalah *vanishing gradient* [11], dan ia membuat jaringan dalam yang ber-sigmoid nyaris mustahil dilatih.

Penggantinya yang kini menjadi baku adalah **ReLU** [7] (*Rectified Linear Unit*), $\mathrm{ReLU}(z) = \max(0, z)$. Bentuknya sesederhana mungkin, tetapi turunannya tetap 1 untuk seluruh masukan positif (Gambar 12.2, kanan), sehingga gradien tidak menyusut saat dilewatkan mundur. Itulah, lebih dari apa pun, yang memungkinkan jaringan benar-benar dalam dilatih. Yang penting bagi kita: mengganti sigmoid dengan ReLU *tidak mengubah satu pun* mekanika yang sudah kita pelajari—hanya satu fungsi aktivasi dan turunannya di Persamaan (4.9) yang berganti.



Gambar 12.2: Sigmoid versus ReLU. Kiri: bentuk fungsinya. Kanan: turunannya. Turunan sigmoid (biru) mengecil ke nol di kedua ekor—sumber *vanishing gradient*— sedangkan turunan ReLU (amber) tetap 1 untuk semua masukan positif, sehingga gradien mengalir utuh.

12.4 Mengapa orang menambah kedalaman

Teorema *universal approximation* (Bab 1) menjamin bahwa satu *hidden layer* sudah cukup untuk mendekati fungsi apa pun. Lalu mengapa dunia beralih ke jaringan *dalam* dengan puluhan lapisan?

Jawabannya: "cukup secara prinsip" tidak sama dengan "efisien dalam praktik". Untuk masalah rumit, satu lapisan mungkin menuntut jumlah neuron yang sangat besar dan sulit dilatih, sementara jaringan dalam mencapai hasil yang sama dengan jauh lebih sedikit neuron. Lapisan-lapisan yang bertumpuk membangun fitur secara *berlapis*: lapisan awal menangkap pola sederhana, lapisan berikutnya merangkainya menjadi pola yang makin kompleks—tepi menjadi bentuk, bentuk menjadi objek.

Dan inilah kabar yang menenangkan: menambah lapisan *tidak* menuntut matematika baru. *Forward pass* sekadar menerapkan Persamaan (3.4) berkali-kali; *backpropagation* sekadar menerapkan aturan rantai bercabang yang sama, lapis demi lapis, dari belakang ke depan. Jaringan dengan lima puluh lapisan bekerja dengan prinsip yang persis sama seperti yang Anda turunkan dengan tangan di Bab 4—hanya diulang lima puluh kali.

12.5 Arsitektur khusus, dalam sekilas

Jaringan *feedforward* kita adalah pondasi, tetapi tugas-tugas tertentu memunculkan arsitektur yang dirancang khusus—semuanya tetap dilatih dengan *backpropagation* dan *gradient descent* yang sama.

Convolutional neural network (CNN) [13] memanfaatkan struktur ruang

pada citra, berbagi bobot melintasi posisi sehingga mata yang mengenali sebuah pola di satu sudut gambar juga mengenalinya di sudut lain. *Recurrent neural network* (RNN) beserta penerusnya seperti LSTM [12] dan, lebih mutakhir, *Transformer* [14], dirancang untuk data berurut—teks dan deret waktu—dengan memori sejati yang melampaui trik jendela geser sederhana kita di Bab 10. Tetapi semuanya, tanpa kecuali, dibangun dari blok yang sama: lapisan linear, aktivasi tak-linear, fungsi loss, dan gradien yang mengalir mundur.

12.6 Ke mana melangkah berikutnya

Jika buku ini berhasil, Anda kini memandang *neural network* bukan sebagai kotak hitam ajaib, melainkan sebagai mesin yang setiap rodanya Anda kenal. Itu fondasi yang kokoh untuk melangkah lebih jauh. Beberapa arah yang masuk akal:

- **Berlatih pada lebih banyak data.** Cobalah dataset publik lain dan ulangi alur tiga-wajah buku ini. Keakraban tumbuh dari pengulangan.
- **Dalami sebuah *framework*.** Pelajari PyTorch atau TensorFlow secara serius—kini Anda bisa membaca dokumentasinya dengan tahu persis apa yang dirujuk tiap istilah.
- **Naik ke jaringan dalam.** Tambahkan lapisan, ganti ke ReLU, dan saksikan prinsip yang sama berlaku pada skala yang lebih besar.
- **Telusuri arsitektur khusus.** Begitu pondasinya kokoh, CNN, RNN, dan Transformer akan terasa sebagai variasi, bukan misteri.
- **Perdalam dengan buku rujukan.** Untuk teori yang lebih lengkap, *Deep Learning* [15] dan *Neural Networks and Deep Learning* [16] adalah lanjutan yang sangat baik, dengan *Pattern Recognition and Machine Learning* [17] sebagai rujukan klasik yang lebih luas.

12.7 Penutup

Kita memulai buku ini dengan sebuah janji: membuka kap mesin sebuah *neural network*, dan tidak meninggalkan satu pun rumus yang turun dari langit. Kita menurunkan *backpropagation* suku demi suku, membangun jaringannya sendiri dengan NumPy, lalu memakainya untuk mengklasifikasikan penguin, meramal kekuatan beton, dan memprediksi suhu—tiga wajah dari satu model yang sama.

Yang Anda peroleh bukan sekadar kemampuan melatih satu jaringan kecil, melainkan sebuah *cara memandang*. Setiap sistem *deep learning* yang akan Anda temui—betapapun besar dan rumitnya—pada hakikatnya adalah variasi dan perbesaran dari mesin yang baru saja Anda bongkar dan rakit kembali dengan tangan sendiri. Lapisannya mungkin lebih banyak, datanya lebih besar, namanya lebih megah; tetapi di jantungnya tetap berdetak tiga roda gigi yang sama: sinyal yang mengalir maju, kesalahan yang merambat mundur, dan bobot yang bergeser sedikit demi sedikit menuju kebenaran.

Selamat. Mesinnya kini milik Anda.

Soal Latihan

12.1 [Konsep] Sebutkan tanda-tanda *overfitting* yang bisa dilihat dari kurva loss latih dan loss validasi. Pada titik mana sebaiknya pelatihan dihentikan?

12.2 [Konsep] Jelaskan bagaimana ReLU mengatasi masalah *vanishing gradient* yang dialami sigmoid pada jaringan dalam.

12.3 [Derivasi] Tunjukkan bahwa turunan sigmoid $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ mencapai nilai maksimum 0,25 tepat di $z = 0$. (Petunjuk: $\sigma(0) = \frac{1}{2}$, dan tinjau $p(1 - p)$ untuk $p \in (0, 1)$.)

12.4 [Konsep] Jika satu *hidden layer* secara prinsip sudah cukup (universal approximation), mengapa praktik modern memakai jaringan yang sangat dalam? Berikan dua alasan.

12.5 [Eksperimen] Reproduksi percobaan overfitting (Gambar 12.1) dengan kode `overfitting_demo.py`. Pada epoch berapa loss validasi mencapai minimum? Apa yang terjadi pada loss latih setelah titik itu?

A

Menyiapkan Lingkungan Kerja

Seluruh kode dalam buku ini ditulis dengan Python 3 dan hanya bergantung pada sedikit pustaka. Apendiks ini merangkum cara menyiapkannya agar Anda bisa menjalankan sendiri tiap contoh.

A.1 Yang dibutuhkan

Anda memerlukan Python versi 3.9 atau lebih baru. Untuk memeriksanya:

```
python3 --version
```

Untuk versi *from scratch*—inti buku ini—hanya `numpy` yang diperlukan. Untuk perbandingan TensorFlow di tiap studi kasus, diperlukan `tensorflow`. Dataset Penguins diambil lewat `palmerpenguins`, dan beberapa skrip memakai `pandas` untuk membaca CSV.

A.2 Lingkungan virtual (sangat dianjurkan)

Memasang pustaka langsung ke instalasi Python sistem bisa menimbulkan konflik dengan paket lain yang sudah ada di mesin Anda. Cara yang lebih bersih adalah membuat *lingkungan virtual* yang terisolasi khusus untuk buku ini.

Menggunakan `venv` (bawaan Python)

```
# Buat lingkungan virtual bernama "nn-env"
python3 -m venv nn-env

# Aktifkan (Linux / macOS)
source nn-env/bin/activate
```

```
# Aktifkan (Windows)
nn-env\Scripts\activate

# Prompt berubah menjadi (nn-env) -- tanda lingkungan aktif
```

Setelah aktif, semua perintah `pip` dan `python3` bekerja dalam lingkungan yang terisolasi. Untuk keluar:

```
deactivate
```

Menggunakan `conda` (Anaconda / Miniconda)

Jika Anda menggunakan distribusi Anaconda atau Miniconda:

```
conda create -n nn-env python=3.11
conda activate nn-env
```

A.3 Pemasangan

Dengan lingkungan virtual aktif, pasang pustaka yang diperlukan:

```
pip install numpy pandas tensorflow palmerpenguins
```

Bila hanya ingin menjalankan bagian *from scratch*, cukup:

```
pip install numpy pandas palmerpenguins
```

Catatan

TensorFlow berukuran cukup besar (ratusan megabyte). Pada mesin tanpa GPU, paket `tensorflow` versi CPU sudah memadai untuk seluruh contoh di buku ini, yang semuanya kecil dan selesai dalam hitungan detik hingga menit.

A.4 Menjalankan kode

Semua skrip yang menghasilkan angka dan gambar disertakan dalam folder `code/`. Pastikan Anda menjalankannya *dari dalam* folder tersebut agar referensi ke berkas data (misalnya `concrete.csv`) ditemukan:

```
cd code/
python3 sin_fit.py                # Bab 6: fit kurva sinus
```

```
python3 optimizers_compare.py      # Bab 7: adu pengoptimal
python3 penguins_numpy.py          # Bab 8: klasifikasi (NumPy)
python3 concrete_numpy.py          # Bab 9: regresi (NumPy)
python3 timeseries_numpy.py        # Bab 10: time series (NumPy)
python3 multivariate_timeseries.py  # Bab 11: time series
    multivariat
```

Tiap skrip memuat datanya sendiri (lihat Apendiks B), melatih jaringan, dan mencetak metrik serta koordinat untuk gambar. Karena seluruh keacakan diberi *seed* tetap, hasilnya dapat direproduksi.

A.5 Mengatasi masalah umum

ModuleNotFoundError: No module named `numpy` (atau pustaka lain). Pastikan lingkungan virtual sudah diaktifkan sebelum menjalankan skrip. Cek dengan `which python3`—hasilnya harus menunjuk ke `nn-env`, bukan ke instalasi sistem.

Skrip tidak menemukan berkas data (`FileNotFoundError`). Skrip-skrip studi kasus membaca berkas CSV relatif terhadap direktori kerja. Jalankan selalu dari dalam folder `code/`, bukan dari direktori lain.

Peringatan atau galat saat mengimpor TensorFlow. TensorFlow sering mencetak pesan log yang panjang. Ini normal dan tidak memengaruhi hasil. Jika terjadi galat saat instalasi, coba spesifikasikan versi yang diketahui kompatibel:

```
pip install "tensorflow≥2.12,<2.16"
```

Hasil sedikit berbeda dari yang tertulis di buku. Versi pustaka yang berbeda bisa menghasilkan angka yang sedikit bergeser. Perbedaan kecil (kurang dari satu persen) dalam metrik adalah normal dan tidak berarti kode berjalan keliru. Struktur dan tren hasilnya harus tetap sama.

B

Sumber Data

Semua dataset dalam buku ini bersifat publik dan dapat diunduh bebas. Apendiks ini mencantumkan sumber dan atribusinya.

Dataset	Tugas	Dipakai di
$y = \sin x$ (sintetis)	Regresi (demo)	Bab 6
Palmer Penguins	Klasifikasi	Bab 8
Concrete Compressive Strength	Regresi	Bab 9
Daily Minimum Temperatures	Time series	Bab 10
Beijing PM2.5 (polusi udara)	Time series multivariat	Bab 11

Tabel B.1: Dataset yang dipakai sepanjang buku.

B.1 Data sintetis $\sin x$

Dibangkitkan langsung di kode sebagai 64 titik $y = \sin x$ pada $[-3, 3]$; lihat `code/sin_fit.py`. Tak perlu diunduh.

B.2 Palmer Penguins

Diambil lewat paket `palmerpenguins` [21], yang membundel datanya:

```
from palmerpenguins import load_penguins
df = load_penguins()
```

Data dikumpulkan oleh Dr. Kristen Gorman dan Palmer Station, Antarctica LTER [20].

B.3 Concrete Compressive Strength

Berisi 1030 campuran beton (8 fitur, target kekuatan tekan). Aslinya dari *UCI Machine Learning Repository* [23], disumbangkan oleh I-Cheng Yeh [22]. Diunduh dari salinan CSV publik:

```
https://raw.githubusercontent.com/stedy/
  Machine-Learning-with-R-datasets/master/concrete.csv
```

Salinan disertakan di `code/concrete.csv`.

B.4 Daily Minimum Temperatures

Deret 3650 suhu minimum harian di Melbourne (1981–1990). Diunduh dari:

```
https://raw.githubusercontent.com/jbrownlee/
  Datasets/master/daily-min-temperatures.csv
```

Salinan disertakan di `code/temps.csv`.

B.5 Beijing PM2.5 (polusi udara)

Pengukuran per jam selama lima tahun (2010–2014, ∼43,800 baris) berisi PM2.5 dan enam variabel cuaca [24]. Diunduh dari:

```
https://raw.githubusercontent.com/jbrownlee/
  Datasets/master/pollution.csv
```

Salinan disertakan di `code/pollution.csv`.

Catatan

URL salinan publik bisa berubah sewaktu-waktu. Karena itu berkas datanya juga disertakan langsung dalam folder `code/`, sehingga seluruh contoh tetap dapat dijalankan meski tautan daringnya suatu saat tak lagi tersedia.

C

Kode Lengkap

Sepanjang buku, kode jaringan diperkenalkan sepotong demi sepotong. Apendiks ini menyatukannya menjadi satu modul rujukan ringkas yang menangani *baik* regresi maupun klasifikasi sekaligus—benang merah buku ini dalam wujud kode: satu badan jaringan, dengan kepala yang dipilih lewat satu argumen `task`.

C.1 Modul jaringan

Berkas berikut tersedia di `code/nn_lib.py`.

```
import numpy as np

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

def softmax(U):
    U = U - U.max(axis=1, keepdims=True)   # stabilitas numerik
    e = np.exp(U)
    return e / e.sum(axis=1, keepdims=True)

class NeuralNetwork:
    """Feedforward satu hidden layer (sigmoid) untuk regresi / klasifikasi."""
    def __init__(self, d, m, k, task="regression", seed=0):
        r = np.random.default_rng(seed)
        self.W = r.normal(0, 0.5, (m, d)); self.b = np.zeros(m)
        self.V = r.normal(0, 0.5, (k, m)); self.c = np.zeros(k)
        self.task = task

    def forward(self, X):
        self.Z = X @ self.W.T + self.b
        self.H = sigmoid(self.Z)
```

```
        self.U = self.H @ self.V.T + self.c
        if self.task == "classification":
            self.Yhat = softmax(self.U)
        else:
            self.Yhat = self.U                  # keluaran linear
        return self.Yhat

    def backward(self, X, Y):
        n = X.shape[0]
        dU = (self.Yhat - Y) / n                # delta_out =
            yhat - y
        dV = dU.T @ self.H
        dc = dU.sum(axis=0)
        dH = dU @ self.V
        dZ = dH * self.H * (1 - self.H)         # delta_hid
        dW = dZ.T @ X
        db = dZ.sum(axis=0)
        return dW, db, dV, dc

    def step(self, grads, eta):
        dW, db, dV, dc = grads
        self.W -= eta * dW;  self.b -= eta * db
        self.V -= eta * dV;  self.c -= eta * dc

def mse(Yhat, Y):
    return 0.5 * np.mean(np.sum((Yhat - Y)**2, axis=1))

def cross_entropy(Yhat, Y):
    return -np.mean(np.sum(Y * np.log(Yhat + 1e-9), axis=1))

def train(net, X, Y, eta=0.3, epochs=3000):
    for _ in range(epochs):
        net.forward(X)
        net.step(net.backward(X, Y), eta)
    return net
```

Perhatikan bahwa `backward` sama persis untuk kedua tugas—hanya `forward` (lewat `task`) dan pilihan fungsi loss yang berbeda. Itulah bukti kode dari benang merah "satu model, ujungnya yang ganti".

Catatan

Modul ini sengaja minimal demi kejelasan: ia memakai *full-batch gradient descent* dan mengasumsikan data sudah distandarkan. Menambahkan mini-batch, momentum, atau Adam (Bab 7) tinggal memodifikasi fungsi `train`—struktur intinya tidak berubah.

C.2 Daftar skrip studi kasus

Selain `nn_lib.py`, folder `code/` berisi skrip mandiri untuk tiap studi kasus. Masing-masing berdiri sendiri—tidak bergantung pada `nn_lib.py`—sehingga kode jaringan yang persis dibahas per bab dapat dibaca secara langsung.

Berkas	Bab	Tujuan
`sin_fit.py`	Bab 6	Demo fit $y = \sin x$, regresi dasar
`optimizers_compare.py`	Bab 7	Perbandingan SGD, Momentum, Adam
`penguins_numpy.py`	Bab 8	Klasifikasi penguin, *from scratch*
`penguins_tensorflow.py`	Bab 8	Klasifikasi penguin, TensorFlow
`concrete_numpy.py`	Bab 9	Regresi kekuatan beton, *from scratch*
`concrete_tensorflow.py`	Bab 9	Regresi kekuatan beton, TensorFlow
`timeseries_numpy.py`	Bab 10	Peramalan suhu harian, *from scratch*
`timeseries_tensorflow.py`	Bab 10	Peramalan suhu harian, TensorFlow
`multivariate_timeseries.py`	Bab 11	Peramalan PM2.5 multivariat, Adam
`overfitting_demo.py`	Bab 12	Demonstrasi *overfitting* dan *early stopping*
`extra_figures.py`	(pelengkap)	Permukaan loss 3D dan kurva belajar

Tabel C.1: Seluruh skrip dalam folder `code/` beserta keterangan singkatnya.

Berikut uraian singkat setiap skrip, termasuk arsitektur jaringan yang dipakainya.

`sin_fit.py` — Fit kurva $y = \sin x$. Data sintetis 64 titik, regresi satu keluaran. Arsitektur: d=1, m=8, k=1, aktivasi sigmoid, keluaran linear, *full-batch gradient descent*, 5 000 epoch. Dipakai untuk membangun intuisi awal tentang forward pass dan backward pass sebelum data nyata diperkenalkan.

`optimizers_compare.py` — Perbandingan pengoptimal. Melatih jaringan yang sama (d=8, m=16, k=1) pada dataset Concrete dengan tiga pengoptimal—SGD, Momentum, dan Adam—dan mencetak kurva loss untuk tiap-tiapnya. Berguna untuk melihat secara langsung perbedaan kecepatan konvergensi yang dibahas di Bab 7.

`penguins_numpy.py` dan `penguins_tensorflow.py` — Klasifikasi penguin. Dataset Palmer Penguins, 333 contoh, 4 fitur, 3 kelas. Arsitektur *from scratch*: d=4, m=8, k=3, aktivasi sigmoid, kepala softmax, *full-batch*, η=0,5, 3 000 epoch. Versi TensorFlow menggunakan `keras.Sequential` dengan lapisan dan optimizer yang setara untuk perbandingan langsung. Kedua skrip mencetak akurasi latih dan uji.

`concrete_numpy.py` dan `concrete_tensorflow.py` — Regresi kekuatan beton. Dataset UCI Concrete, 1 030 contoh, 8 fitur, target kekuatan tekan dalam MPa. Arsitektur *from scratch*: d=8, m=16, k=1, keluaran linear, MSE, *full-batch*, η=0,3, 2 000 epoch. Skrip mencetak RMSE dan R^2 pada set latih dan uji, serta membandingkannya dengan versi TensorFlow (Adam, 1 500 epoch).

`timeseries_numpy.py` dan `timeseries_tensorflow.py` — Peramalan suhu harian. Dataset Melbourne (3 650 hari). Jendela geser L=14 hari menghasilkan tabel regresi dengan d=14, k=1. Arsitektur *from scratch*: m=16, keluaran linear, pembagian kronologis 80/20 (tanpa pengacakan). Skrip mencetak RMSE dalam satuan derajat Celsius dan membandingkannya dengan *baseline* persistensi.

`multivariate_timeseries.py` — Peramalan PM2.5 multivariat. Dataset Beijing PM2.5, ∼43 800 baris, 7 variabel. Jendela masukan L=8 jam, cakrawala prediksi H=12 jam ke depan menghasilkan d=56 ($= L \times 7$). Arsitektur: m=64, k=1, pengoptimal Adam (*mini-batch* B=128, η=0,01, 60 epoch). Satu-satunya skrip yang menggunakan Adam secara penuh sebagai *from scratch*.

`overfitting_demo.py` — Demonstrasi *overfitting*. Data sintetis 12 titik berderau dari $\sin(1{,}4x)$. Jaringan besar (m=40) dilatih hingga 100 000 epoch sehingga loss latih terus turun sementara loss validasi mulai naik kembali. Skrip mencetak log loss latih dan validasi di titik-titik waktu logaritmis, yang dipakai untuk membuat gambar kurva *overfitting* di Bab 12.

`extra_figures.py` — Gambar pelengkap. Skrip utilitas yang menghasilkan dua berkas data untuk figur buku: (1) permukaan loss 3D ($\mathcal{L}$ vs. w dan b) pada regresi linear satu fitur, dan (2) kurva belajar untuk berbagai ukuran hidden layer m pada masalah $\sin x$. Tidak perlu dijalankan ulang kecuali figur perlu diperbarui.

D

Glosarium

Buku ini sengaja membiarkan banyak istilah teknis dalam Bahasa Inggris aslinya, karena begitulah istilah-istilah tersebut paling sering dijumpai dalam literatur, dokumentasi *framework*, dan diskusi praktisi. Glosarium ini merangkumnya secara alfabetis (menurut istilah Indonesia atau istilah yang dipakai dalam teks) sebagai rujukan cepat.

Adam (*Adaptive Moment Estimation*) Pengoptimal yang memberi tiap parameter *learning rate* yang menyesuaikan diri, dengan menggabungkan rata-rata gradien dan rata-rata gradien-kuadrat. Lihat Bab 7.

Backpropagation Prosedur menghitung gradien loss terhadap setiap parameter jaringan, dengan menjalankan aturan rantai mundur dari lapisan keluaran ke lapisan masukan. Lihat Bab 4.

Batch Sekumpulan contoh data yang diproses bersama dalam satu langkah pembaruan parameter. Lihat juga **Mini-batch**.

Cross-entropy Fungsi loss baku untuk klasifikasi, mengukur seberapa jauh distribusi peluang tebakan dari label sebenarnya. Lihat Bab 8.

Data leakage (kebocoran data) Situasi saat informasi dari data uji—atau dari masa depan, pada deret waktu—secara tak sengaja ikut memengaruhi pelatihan, membuat evaluasi tampak lebih baik dari kemampuan sebenarnya.

Dropout Teknik regularisasi yang mematikan sebagian neuron secara acak selama pelatihan, untuk mencegah *overfitting*.

Early stopping Strategi menghentikan pelatihan saat loss validasi berhenti membaik, sebelum jaringan mulai menghafal data latih.

Epoch Satu lintasan penuh melalui seluruh data latih.

Feedforward Arsitektur jaringan di mana informasi mengalir satu arah, dari masukan ke keluaran, tanpa gelung balik.

Forward pass Proses menghitung keluaran jaringan dari suatu masukan, lapisan demi lapisan ke depan.

Gradient descent Algoritma optimisasi yang menggeser parameter sedikit demi sedikit ke arah berlawanan dengan gradien loss, menuju nilai loss yang lebih rendah. Lihat Bab 5.

Hidden layer (lapisan tersembunyi) Lapisan neuron di antara masukan dan keluaran, tempat representasi internal jaringan dibangun.

Hyperparameter Pengaturan yang ditentukan sebelum pelatihan (mis. *learning rate*, jumlah neuron) dan tidak dipelajari oleh jaringan itu sendiri.

Jendela geser (*sliding window*) Teknik mengubah deret waktu menjadi tabel regresi, dengan mengiris L nilai berturutan sebagai masukan dan nilai berikutnya sebagai target. Lihat Bab 10.

Learning rate Bilangan yang mengatur seberapa besar langkah pembaruan parameter pada tiap iterasi *gradient descent*.

Loss function (fungsi kerugian) Fungsi yang mengukur seberapa jauh tebakan jaringan dari nilai sebenarnya; pelatihan berarti meminimalkan fungsi ini.

MAE (*Mean Absolute Error*) Ukuran galat regresi: rata-rata nilai absolut selisih antara tebakan dan nilai sebenarnya.

Mini-batch Potongan kecil dari data latih yang dipakai untuk satu langkah pembaruan parameter, kompromi antara *batch* penuh dan satu contoh saja (*stochastic*). Lihat Bab 7.

Momentum Teknik optimisasi yang mengakumulasi arah gradien dari langkah-langkah sebelumnya untuk mempercepat dan menstabilkan pelatihan.

MSE (*Mean Squared Error*) Ukuran galat regresi: rata-rata kuadrat selisih antara tebakan dan nilai sebenarnya. Lihat Bab 9.

One-hot Cara mengkodekan label kategori sebagai vektor biner, dengan nilai 1 pada posisi kelas yang benar dan 0 di posisi lainnya.

Overfitting Kondisi saat jaringan menghafal data latih—termasuk derau di dalamnya—alih-alih menangkap pola sejati, sehingga kemampuannya pada data baru memburuk. Lihat Bab 12.

Persistensi (*persistence baseline*) Peramal naif untuk deret waktu yang menebak nilai mendatang sama dengan nilai saat ini; menjadi ambang pembanding untuk model yang lebih canggih.

ReLU (*Rectified Linear Unit*) Fungsi aktivasi $\max(0, z)$ yang kini menjadi baku di jaringan dalam karena turunannya tidak menyusut untuk masukan positif.

RMSE (*Root Mean Squared Error*) Akar dari MSE; dinyatakan dalam satuan yang sama dengan target, sehingga lebih mudah ditafsirkan.

R^2 (koefisien determinasi) Proporsi ragam target yang berhasil dijelaskan oleh model regresi; 1 berarti sempurna.

Sigmoid Fungsi aktivasi $\sigma(z) = 1/(1 + e^{-z})$ yang memetakan sembarang bilangan real ke selang $(0, 1)$; aktivasi utama yang dipakai di buku ini.

Softmax Fungsi yang mengubah sekumpulan skor mentah menjadi distribusi peluang yang menjumlah ke 1, dipakai sebagai kepala keluaran untuk klasifikasi banyak kelas.

Standarisasi Mengubah skala suatu fitur dengan mengurangi reratanya lalu membaginya dengan simpangan baku, agar seluruh fitur berskala sebanding sebelum dipakai melatih jaringan.

Time series forecasting (peramalan deret waktu) Tugas memprediksi nilai mendatang suatu deret berdasarkan nilai-nilai masa lalunya. Lihat Bab 10 dan 11.

Universal approximation theorem Teorema yang menjamin jaringan dengan satu *hidden layer* (dengan neuron secukupnya) dapat mendekati hampir semua fungsi kontinu sebaik yang diinginkan. Lihat Bab 1.

Vanishing gradient Masalah yang muncul saat gradien menyusut menuju nol ketika dilewatkan mundur melalui banyak lapisan, membuat lapisan-lapisan awal jaringan dalam sulit dilatih. Lihat Bab 12.

Daftar Pustaka

[1] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323, 533–536.

[2] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4), 303–314.

[3] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), 359–366.

[4] Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5), 1–17.

[5] Robbins, H., & Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3), 400–407.

[6] Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*.

[7] Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. *International Conference on Machine Learning (ICML)*.

[8] Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *International Conference on Artificial Intelligence and Statistics (AISTATS)*.

[9] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. *International Conference on Computer Vision (ICCV)*.

[10] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15, 1929–1958.

[11] Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), 157–166.

[12] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.

[13] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324.

[14] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*.

[15] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

[16] Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

[17] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

[18] Harris, C. R., et al. (2020). Array programming with NumPy. *Nature*, 585, 357–362.

[19] Abadi, M., et al. (2016). TensorFlow: A system for large-scale machine learning. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[20] Gorman, K. B., Williams, T. D., & Fraser, W. R. (2014). Ecological sexual dimorphism and environmental variability within a community of Antarctic penguins (genus *Pygoscelis*). *PLoS ONE*, 9(3), e90081.

[21] Horst, A. M., Hill, A. P., & Gorman, K. B. (2020). *palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. R package. `https://allisonhorst.github.io/palmerpenguins/`

[22] Yeh, I.-C. (1998). Modeling of strength of high-performance concrete using artificial neural networks. *Cement and Concrete Research*, 28(12), 1797–1808.

[23] Dua, D., & Graff, C. (2019). *UCI Machine Learning Repository*. University of California, Irvine, School of Information and Computer Sciences.

[24] Liang, X., Zou, T., Guo, B., Li, S., Zhang, H., Zhang, S., Huang, H., & Chen, S. X. (2015). Assessing Beijing's PM2.5 pollution: Severity, weather impact, APEC and winter heating. *Proceedings of the Royal Society A*, 471(2182), 20150257.

[25] Hyndman, R. J., & Athanasopoulos, G. (2021). *Forecasting: Principles and Practice* (3rd ed.). OTexts.

Tentang Penulis

Dr. Mohammad Jamhuri adalah akademisi di bidang matematika dengan minat pada pemodelan matematika, komputasi ilmiah, optimisasi, sistem dinamik, dan pembelajaran mesin. Dalam kegiatan akademiknya, penulis banyak terlibat dalam pengajaran, penyusunan bahan ajar, pendampingan mahasiswa, serta pengembangan kajian matematika terapan dan komputasi.

Penulis memiliki perhatian khusus pada penyajian materi matematika dan pembelajaran mesin dalam bahasa Indonesia yang runtut, mudah diikuti, tetapi tetap menjaga ketepatan konsep. Melalui buku ini, penulis berupaya menunjukkan bahwa *neural network* tidak harus dipelajari sebagai kotak hitam. Dengan penurunan matematis yang bertahap, penjelasan yang cukup, dan kode yang dapat dijalankan secara mandiri, pembaca dapat memahami cara kerja jaringan saraf tiruan mulai dari dasar.

Buku ini disusun untuk pembaca yang ingin memahami *neural network* secara lebih mendalam, terutama mahasiswa dan pembelajar awal yang telah memiliki bekal dasar kalkulus, aljabar linear, dan pemrograman. Fokus utama buku ini bukan sekadar menjalankan pustaka siap pakai, melainkan memahami proses di balik *forward pass*, fungsi aktivasi, fungsi *loss*, gradien, *backpropagation*, dan pembaruan bobot.

Penulis berharap buku ini dapat menjadi jembatan antara teori matematika dan implementasi komputasi, sehingga pembaca tidak hanya mampu menggunakan model, tetapi juga memahami alasan matematis di balik setiap langkahnya.