Implementasi Pustaka *Deep Learning* Dari Dasar

Aria Ghora Prabono

Implementasi Pustaka *Deep Learning* Dari Dasar

Aria Ghora Prabono

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

Daftar Isi

Pendahuluan

Implementasi Dari Dasar

Catatan

- TODO
- Makna "dari dasar"
- Bedanya dibanding buku lain dengan klaim "dari dasar"
- Bahasa pengantar: Py
- Pake NumPy
- API referensi: pytorch

Sasaran Pembaca Buku ini

Untuk Siapa Buku ini Ditujukan

Buku ini ditujukan untuk praktisi rekayasa perangkat lunak yang bekerja dengan sistem *machine learning* dan mahasiswa ilmu komputer (atau serumpun) akhir tahun atau tingkat lanjut yang

- Terbiasa menulis program di bahasa pemrograman Python (atau bahasa pemrograman serbaguna lainnya)
- Terbiasa membaca dan menulis kode tak trivial
- Memiliki dasar kalkulus dan aljabar linier

Pembaca tidak harus menguasai materi *deep learning*, namun baiknya termotivasi untuk memahami mekanismenya di balik layar. Penulis juga menyasar pembaca yang mungkin penasaran mengapa pustaka seperti PyTorch didesain sebagaimana adanya dan melakukan sesuatu dengan cara tertentu. Mungkin Anda bosan menerima sistem autodiff apa adanya sebagai ilmu sihir atau kotak hitam, atau mungkin Anda sedang membangun sistem *machine learning* dan ingin memahami implikasi suatu implementasi terhadap performa.

Untuk Siapa Buku ini (Mungkin) Tidak Ditujukan

Buku ini bersifat semi-praktis namun tingkat lanjutan. Buku ini tidak akan mengajarkan materi fundamental *machine learning* maupun *deep learning*, tidak pula mencoba meyakinkan pembaca bahwa *neural network* adalah teknologi yang ajaib dan berguna bagi peradaban.

Jika pembaca mencari materi yang bersifat pengenalan pada *machine learning*, silakan memulainya dengan sumber bacaan atau literatur yang lain. Pembaca yang hanya ingin cukup menggunakan kerangka kerja *machine learning* tanpa ada keperluan memahami implementasi internalnya mungkin tidak terlalu membutuhkan buku ini. Jika pembaca tidak terlalu nyaman dengan contoh kode dengan kompleksitas menengah, mungkin buku ini juga tidak terlalu cocok bagi Anda.

Notasi

Ragam Blok Penyerta Teks

Catatan

Blok catatan

Pendalaman

Blok pendalaman

Contoh: Judul blok Contoh

Blok contoh

Peringatan

Blok Peringatan

Potongan kode

Potongan kode ditampung dalam blok dengan teks yang disorot. Kadang potongan kode disertai contoh luaran yang bersesuaian dalam blok abu-abu dengan label "Luaran".

```
print("Halo dunia")

for i in ["satu", "dua", "tiga"]:
    print(i)
```

Luaran

```
Halo dunia
satu
dua
tiga
```

Notasi Matematika

- Skalar $x, y, a, b, x_{ij} \in \mathbf{X}$
- Vektor $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{a}, \boldsymbol{b}$
- Matriks $\mathbf{X}, \mathbf{Y}, \mathbf{A}, \mathbf{B}$
- Himpunan $\mathcal{X}, \mathcal{Y}, \mathcal{A}, \mathcal{B}$

Bab 1. Optimasi pada *Deep Learning* dan Permasalahannya

Tiap inovasi di bidang kecerdasan buatan, mulai dari pengenalan wajah, deteksi objek, hingga penerjemahan bahasa, semua berujung pada penyelesaian masalah fundamental, yaitu **optimasi**. Ketika *neural network* belajar membedakan citra kucing dan anjing, menghasilkan teks manusiawi, atau memprediksi harga saham, ia akan aktif melakukan pencarian di suatu ruang parameter demi meminimalkan galat prediksi. Pencarian kombinasi dari jutaan hingga triliunan parameter ini adalah soalan utama di dunia kecerdasan buatan modern.

Mekanisme *deep learning* masih buram bagi banyak praktisi. Kita mudah saja menggunakan pustaka, memanggil `loss.backward()` dan `optimizer.step()` *à la* PyTorch, kemudian berharap secara ajaib performa model yang kita latih akan meningkat. Namun, apa yang sebenarnya terjadi saat pemanggilan fungsi-fungsi tersebut? Bagaimana komputer secara efektif menjelajah parameter dalam dimensi tinggi? Bab ini akan menjadi pengantar untuk mengulasnya.

1.1. Permasalahan Proses Pembelajaran Mesin

Pada intinya, *deep learning* ada persoalan mencari suatu fungsi yang memetakan masukan ke luaran tanpa harus secara eksplisit mengimplementasikan fungsi itu secara manual. Jika diberi citra digit tertulis tangan, prediksi angkanya. Jika diberi teks Bahasa Inggris, beri terjemahan Bahasa Indonesianya. Kita tidak pula perlu secara manual melakukan rekayasa fitur pada data masukan karena model akan mempelajarinya sendiri.

"Namun, apa sebetulnya makna dari '**mempelajari** suatu fungsi'?"

Pada konteks *neural network*, "mempelajari fungsi" ini adalah soal mencari parameter model yang tepat sehingga model dapat melakukan prediksi yang juga tepat.

Perhatikan pengklasifikasi citra digit berikut ini.

```
import torch
import torch.nn as nn
import numpy as np

class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)  # 784*128 + 128 = 100,480 parameters
        self.fc2 = nn.Linear(128, 64)   # 128*64 + 64 = 8,256 parameters
        self.fc3 = nn.Linear(64, 10)    # 64*10 + 10 = 650 parameters
        # Total: 109,386 parameters

    def forward(self, x):
```

```
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

model = SimpleNet()
total_params = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total_params:,}")
```

Arsitektur yang nampak sederhana ini memiliki 109,386 parameter yang perlu dipelajari (*learnable parameter*). Melatih model ini berarti mencari konfigurasi parameter optimal. Pertanyaannya, **optimal terhadap apa?**

Dalam dunia *neural network*, kita bisa mengukur "seberapa buruk" prediksi model kita saat dilatih dengan melihat seberapa menyimpang prediksi model dari jawaban sebenarnya. Fungsi untuk mengukurnya kita sebut fungsi *loss*. Untuk tugas klasifikasi seperti MNIST, kita biasanya menggunakan ukuran *cross-entropy loss*:

```
def cross_entropy_loss(predictions, targets):
    """
    Fungsi loss cross-entropy (yang disederhanakan)

    predictions: (batch_size, num_classes) - luaran model (logits)
    targets: (batch_size,) - label kelas sebenarnya
    """
    # Konversi logits menjad nilai probabilitas
    exp_preds = np.exp(predictions - np.max(predictions, axis=1, keepdims=True))
    probs = exp_preds / np.sum(exp_preds, axis=1, keepdims=True)

    # Ekstraksi probabilitas dari kelas sebenarnya
    batch_size = predictions.shape[0]
    true_class_probs = probs[range(batch_size), targets]

    # Negative log likelihood
    return -np.mean(np.log(true_class_probs + 1e-8))
```

Melatih *neural network* berarti mencari nilai parameter yang meminimalkan suatu fungsi *loss*. Jika θ merepresentasikan suatu parameter pada model, kita perlu mencari nilai optimalnya, θ^*, dengan menyelesaikan

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, \ell(\theta) \tag{1.1}$$

Jadi, bisa kita pahami bahwa pencarian parameter optimal yang kita bicarakan adalah **optimal terhadap fungsi *loss***. Inilah inti dari optimasi, penggerak utama *deep learning*.

Mulai dari sini, semua akan semakin menantang, menarik, dan... rumit.

1.2. Kutukan Dimensionalitas (*The Curse of Dimensionality*)

Untuk lebih memahami situasi permasalahannya, mari kita visualisasikan apa yang sebenarnya sedang kita hadapi. Untuk masalah optimasi klasik, kita bisa meminimalkan suatu fungsi dari satu hingga

dua variabel. Kita bisa gambarkan fungsinya di atas selembar kertas, amati lanskapnya, dan identifikasi puncak dan lembahnya. Namun, pengklasifikasi MNIST kita memiliki 109,386 parameter (109,386 dimensi parameter yang perlu dioptimalkan). Jelas kita tidak bisa menyelesaikan ini dengan *brute force*. Kita membutuhkan pendekatan yang lebih efektif dan efisien.

Untungnya, dengan sedikit kalkulus, kita tidak perlu mencari di setiap sudut ruang parameter secara acak. Kita dapat mencari gradien suatu fungsi untuk mengarahkan pencarian. Gradien akan menunjukkan arah menuju ketinggian paling curam pada kurva *loss*, dan kita bisa mengambil arah sebaliknya untuk menuju nilai *loss* yang lebih kecil.

Hal ini yang menjadi cikal-bakal aturan pembaruan parameter pada *gradient descent*:

$$\theta := \theta - \alpha \frac{\partial l(\theta)}{\partial \theta} \tag{1.2}$$

di mana α adalah *learning rate*. Jangan lupa, θ adalah satu nilai skalar. Untuk model MNIST, bagaimana menangani turunan 109,386 parameter θ, di mana ada banyak fungsi rumit yang terlibat dalam prediksi dan penghitungan *loss*?

1.3. Solusi Naif: Diferensiasi Numerik

Saat dihadapkan dengan masalah komputasi turunan, insting pertama beberapa dari kita mungkin mengimplementasikan turunan sebarang fungsi berdasarkan definisi standar,

$$f'(x) = \lim_{h \to 0} \frac{[f(x+h) - f(x)]}{h} \tag{1.3}$$

Untuk komputasi yang lebih praktis, kita gunakan hampiran sebagai berikut.

$$\frac{\partial l(\theta_i)}{\partial \theta_i} \approx \frac{l(\theta + h \cdot e_i) - l(\theta - h \cdot e_i)}{2h}, \tag{1.4}$$

di mana e_i adalah vektor satuan untuk arah ke-i. Mari kita implementasikan untuk masalah sederhana untuk memahami mekanismenya. Untuk contoh ini, kita mencoba mencari berapa nilai w dan b yang memenuhi persamaan $f(x) = 2x + 1$. Untuk mensimulasikan derau, kita tambahkan suku 0.1ε pada fungsi, di mana $\varepsilon \sim \mathcal{N}(0, 1)$.

```
import numpy as np
import time

# Simple linear regression: y = w*x + b
np.random.seed(42)
n_samples = 100
x = np.linspace(0, 10, n_samples)
y_true = 2 * x + 1 + 0.1 * np.random.randn(n_samples)  # True: w=2, b=1


def mse_loss(params, x, y):
    """_Mean squared error loss_ untuk regresi linier"""
    w, b = params
    y_pred = w * x + b
```

```
    return np.mean((y - y_pred) ** 2)


def numerical_gradient(f, params, h=1e-5):
    """Compute gradient using finite differences"""
    grad = np.zeros_like(params)

    for i in range(len(params)):
        # Create perturbation vectors
        params_plus = params.copy()
        params_minus = params.copy()

        params_plus[i] += h
        params_minus[i] -= h

        # Finite difference approximation
        grad[i] = (f(params_plus, x, y_true) - f(params_minus, x, y_true)) / (2 * h)

    return grad


# Training with numerical gradients
params = np.array([0.0, 0.0])  # Initialize w=0, b=0
learning_rate = 0.01

print("Training with numerical differentiation:")
print(f"{'Epoch':<6} {'Loss':<10} {'w':<10} {'b':<10} {'Time':<10}")
print("-" * 46)

for epoch in range(1000):
    start_time = time.time()

    loss = mse_loss(params, x, y_true)
    grad = numerical_gradient(mse_loss, params)
    params -= learning_rate * grad

    elapsed = (time.time() - start_time) * 1000
    if epoch % 100 == 0:
        print(
            f"{epoch:<6} {loss:<10.4f} {params[0]:<10.3f} {params[1]:<10.3f}
{elapsed:<10.2f}"
        )
```

Luaran

```
Training with numerical differentiation:
Epoch  Loss       w          b          Time
──────────────────────────────────────────────
0      154.8335   1.439      0.220      0.09
100    0.0499     2.062      0.579      0.03
200    0.0236     2.038      0.738      0.02
300    0.0138     2.024      0.834      0.07
400    0.0103     2.015      0.892      0.02
500    0.0089     2.010      0.928      0.03
600    0.0084     2.006      0.949      0.02
700    0.0083     2.004      0.962      0.02
800    0.0082     2.003      0.970      0.02
900    0.0082     2.003      0.975      0.02
```

Nampaknya berhasil, parameter yang dicari menghampiri nilai sebenarnya: $w = 2$ dan $b = 1$. Untuk contoh main-main seperti ini, diferensiasi numerik mungkin saja bekerja. Mengapa kita tidak menggunakan ini saja untuk *neural network*, khususnya *deep learning*?

1.3.1. Ledakan Komputasi

Kekurangan terbesar dari diferensiasi numerik akan nampak jelas saat skala permasalahannya mulai realistis. Tiap komputasi gradien untuk masing-masing skalar paling tidak membutuhkan dua kali laluan maju pada keseluruhan model: satu untuk perturbasi positif, satu untuk negatif.

Untuk `SimpleNet` kita dengan jumlah parameter 109,386 akan ada 218,772 laluan maju. Asumsikan, dengan optimis, satu laluan maju (eksekusi $f(x)$) 1 ms. Waktu yang dibutuhkan untuk menghitung seluruh gradien adalah 219 detik, atau sekitar 3.65 menit, **untuk satu *mini-batch***. Itu untuk satu iterasi latih saja, dan untuk model yang kecil untuk standar modern. Sebagai perbandingan, ResNet-50 [1] memiliki 25,600,000 parameter, BERT-Base [2] memiliki 110,000,000, dan GPT-3 [3] memiliki 175,000,000,000. Pada Tabel 1, kita bisa melihat perkiraan waktu yang dibutuhkan untuk menghitung gradien seluruh parameter model dengan asumsi satu *mini-batch* berukuran 32 sampel.

Model	Dataset	Jumlah Sampel Data Latih	Waktu Total Per *Epoch*
SimpleNet	MNIST	60,000	113.9 Jam
ResNet-50	ImageNet	1,281,167	65 Tahun
BERT-Base	BookCorpus	3,300,000	719.4 Tahun
GPT-3	Common Crawl	300,000,000,000	104,047,754,946 Tahun

Tabel 1: Perkiraan waktu yang dibutuhkan untuk menghitung seluruh gradien

1.3.2. Batasan Presisi Numerik

Katakanlah kita diizinkan berfantasi memiliki sumber daya komputasi tak hingga. Diferensiasi numerik akan tetap menghadapi batasan lain, yaitu presisi numerik yang terbatas. Simak dan jalankan kode berikut.

```python
import numpy as np

def f(x):
    return (x + 1e-10)**20 - (x - 1e-10)**20

def numerical_gradient(x, h):
    return (f(x + h) - f(x - h)) / (2 * h)

x = 1.0
print(f"Function: (x + 1e-10)^20 - (x - 1e-10)^20")
print(f"Evaluation point: x = {x}")
print(f"True derivative: {40 * x**19 * 1e-10}")
print(f"\nNumerical gradients:")
print(f"{'h':<15} {'Gradient':<20} {'Error vs True':<20}")
print("-" * 60)

true_grad = 40 * x**19 * 1e-10
for h in [1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8, 1e-10, 1e-12]:
    grad = numerical_gradient(x, h)
    error = abs(grad - true_grad)
    print(f"{h:<15.0e} {grad:<20.2e} {error:<20.2e}")
```

Luaran

```
Function: (x + 1e-10)^20 - (x - 1e-10)^20
Evaluation point: x = 1.0
True derivative: 4e-09

Numerical gradients:
h               Gradient            Error vs True
------------------------------------------------------------
1e-03           7.60e-08            7.20e-08
1e-04           7.60e-08            7.20e-08
1e-05           7.60e-08            7.20e-08
1e-06           7.60e-08            7.20e-08
1e-07           7.61e-08            7.21e-08
1e-08           7.22e-08            6.82e-08
1e-10           0.00e+00            4.00e-09
1e-12           1.11e-03            1.11e-03
1e-13           1.11e-02            1.11e-02
1e-14           0.00e+00            4.00e-09
1e-15           1.11e+00            1.11e+00
```

Dari hasil di atas, kita melihat bahwa untuk fungsi sederhana ini, diferensiasi numerik mulai tidak stabil ketika h terlalu kecil. Ini terjadi karena *"catastrophic cancellation"*, yaitu saat kita mengurangi dua bilangan yang hampir identik, sehingga kehilangan semua digit signifikan. Hal ini terjadi dalam persamaan turunan numerik ketika h terlalu kecil, sehingga $f(x+h)$ dan $f(x-h)$ menjadi hampir

identik dan pengurangannya kehilangan digit signifikan. Hasilnya, pembilang yang seharusnya kecil malah didominasi oleh derau presisi *floating-point* dan membuat hampiran turunan menjadi tidak akurat.

Masalah ini jauh lebih parah pada model yang memiliki jutaan parameter dengan skala yang sangat bervariasi. Tidak ada nilai h tunggal yang optimal untuk semua parameter. Sementara sebagian parameter berada di *"sweet spot"*, yang lain akan mengalami *catastrophic cancellation* seperti contoh di atas. Ditambah dengan waktu komputasi yang lama, diferensiasi numerik menjadi tidak praktis dan tidak dapat diandalkan untuk *deep learning*.

1.4. Secuplik Solusi

Alih-alih memperlakukan jaringan saraf sebagai fungsi black box dan mengujinya dengan perturbasi, bagaimana jika kita bisa mengintip ke dalam dan melihat persis bagaimana output bergantung pada setiap parameter?

Setiap komputasi *neural network* hanyalah sederet operasi elementer:

```
# Yang terlihat seperti operasi jaringan saraf yang kompleks...
output = model(input)
loss = criterion(output, target)

# ...sebenarnya adalah urutan operasi sederhana:
h1 = input @ W1 + b1          # Transformasi affine
h2 = torch.relu(h1)           # Nonlinearitas element-wise
h3 = h2 @ W2 + b2             # Transformasi affine lain
h4 = torch.relu(h3)           # Nonlinearitas lain
output = h4 @ W3 + b3         # Transformasi akhir
loss = cross_entropy(output, target)  # Komputasi loss
```

Setiap operasi ini memiliki bentuk matematis yang terdefinisi dengan baik, dan yang lebih penting, turunan yang terdefinisi dengan baik. Aturan rantai dari kalkulus memberi tahu kita bagaimana menggabungkan turunan-turunan ini. Ide utamanya adalah, alih-alih memperlakukan jaringan saraf sebagai black box dan mengujinya dengan perturbasi, kita dapat melacak setiap operasi saat terjadi. Lihat contoh berikut.

```
a = 2.0
b = 3.0
c = a * b      # c = 6.0
d = c + a      # d = 8.0
```

Saat kita menghitung maju, kita secara implisit membangun graf dependensi. Nilai `d` bergantung pada `c` dan `a`, sedangkan `c` bergantung pada `a` dan `b`. Jika kita melacak dependensi ini, kita dapat kemudian menelusurinya mundur untuk menghitung gradien secara efisien menggunakan aturan rantai.

Ini adalah esensi ***automatic differentiation***, yang selanjutnya kita sebut **autodiff**: membangun graf komputasi selama laluan maju, kemudian menelusurinya mundur untuk menghitung semua gradien dalam satu sapuan. Tidak ada perturbasi, tidak ada aproksimasi, semua dilakukan dengan turunan eksak yang dihitung secara efisien. Di bab berikutnya, kita akan membangun sistem ini dari

awal, dimulai dengan tensor yang dapat melacak riwayat komputasinya sendiri. Anda akan melihat persis bagaimana ide elegan ini mengubah komputasi gradien dari masalah yang tidak praktis menjadi algoritma yang efisien.

Bab 2. Implementasi Tensor dan Penghitungan Turunan Otomatis (Autodiff)

"Tensor" menjadi dasar representasi data untuk suatu sistem *deep learning*. Kita akan membangun fitur-fitur secara bertahap, mulai dari struktur data dasar hingga sistem propagasi balik otomatis dasar.

2.1. Terminologi Tensor dalam Konteks *Deep Learning*

Sederhananya, tensor adalah larik berdimensi n (atau *n-dimensional array*), di mana seluruh elemennya memiliki tipe data dasar yang sama. Artinya, larik ini bisa berdimensi sembarang, tidak terbatas pada dimensi 1, 2, dan 3 saja. Pustaka *deep learning* populer yang mengadopsi konsep tensor antara lain, PyTorch[1], Tensorflow[2], dan burn[3].

Bagi pembaca dengan latar belakang fisika dan matematika murni: Anda boleh melupakan dulu definisi tensor yang Anda kenal. Di dunia *deep learning*, tensor memiliki makna yang jauh lebih sederhana. Kata "tensor" dipakai di tiga domain berbeda dengan makna yang **sangat** berbeda:

Domain	**Definisi Tensor**	**Contoh**
Matematika Murni	Objek multilinear yang memetakan vektor ke skalar	Tensor metrik Riemann
Fisika	Besaran yang bertransformasi dengan cara tertentu saat koordinat berubah	Tensor *stress/strain*
Deep Learning	larik multidimensi array	`torch.tensor([[[1,2],[3,4]]])`

Tabel 2: Pemaknaan tensor di tiga bidang yang berbeda

Istilah tensor adalah upaya penamaan generalisasi cara untuk menata nilai-nilai dalam suatu ruang. Jika kumpulan skalar dalam ruang berdimensi satu disebut vektor dan ruang berdimensi dua disebut matriks, bagaimana dengan dimensi tiga dan lebih tinggi? Dari sinilah beragam penelitian dan pustaka menggunakan istilah "Tensor", terlebih sejak dipopulerkan penggunaannya oleh Theano dan Tensorflow. Di buku ini, tensor mengacu pada sistem larik multidimensi. Tidak ada transformasi koordinat, tidak ada kovarian/kontravarian, tidak ada basis vektor. Hanya ada larik multidimensi.

Pembaca mungkin sudah familiar dengan larik multidimensi milik NumPy (`NDArray`), pustaka numerik populer untuk Python. Sekilas, sifat tensor mirip dengan larik NumPy. Lantas, apa bedanya? Sejatinya, **nyaris tidak ada bedanya**. Pembeda utamanya adalah, biasanya, tensor diimplementasikan agar mampu melacak riwayat komputasi (sebagai struktur graf) dan gradien. Pustaka yang lebih mutakhir juga mendesain tensor yang bisa beroperasi di mesin dengan *graphical processing unit* (GPU) untuk pemrosesan yang jauh lebih cepat.

[1] https://pytorch.org/
[2] https://www.tensorflow.org/
[3] https://burn.dev/

Setiap tensor paling tidak memiliki properti "ranking" (*rank*) dan "bentuk" (*shape*). Ranking mengacu pada jumlah dimensi tensor dan bentuk bentuk tensor menjelaskan ukuran tensor untuk tiap dimensi. Tabel 3 menunjukkan contoh tensor dengan ranking yang berbeda-beda.

Ranking	Nama	Bentuk	Contoh Penggunaan
0	Skalar	()	Loss value, learning rate
1	Vektor	$(n,)$	Bias, data 1D
2	Matriks	(n, m)	matriks bobot, citra berskala abu-abu
3	Tensor 3D	(n, m, p)	Tumpukan teks, video berskala abu-abu
4	Tensor 3D	(n, m, p, q)	Tumpukan citra RGB

Tabel 3: Contoh tensor dengan berbagai ranking

Pendalaman

TODO:

Sejarah penamaan tensor

2.2. Pemanfaatan NumPy Sebagai Landasan Utama Tensor

NumPy telah menjadi standar *de facto* untuk komputasi numerik di Python selama lebih dari dua dekade. Pustaka ini menyediakan implementasi larik multidimensi yang efisien dengan dukungan operasi matematika yang telah dioptimalkan dalam bahasa C. Kita juga membutuhkan kemampuan serupa untuk implementasi tensor kita. Namun, alih-alih menulis ulang semua operasi larik dari nol, strategi yang lebih bijak adalah memanfaatkan kekuatan NumPy sebagai fondasi dan menambahkan kemampuan yang dibutuhkan untuk *deep learning*.

NumPy menawarkan beberapa keunggulan fundamental yang membuatnya ideal sebagai *back-end* untuk implementasi tensor kita:

- **Performa tinggi**: Operasi NumPy diimplementasikan dalam C dan memanfaatkan pustaka aljabar linier yang telah dioptimalkan seperti BLAS dan LAPACK. Hal ini memberi akses kecepatan eksekusi yang mendekati bahasa pemrograman tingkat rendah tanpa mengorbankan kemudahan Python.
- **API yang lengkap dan stabil**: NumPy menyediakan ratusan fungsi untuk manipulasi larik, mulai dari operasi dasar seperti penjumlahan hingga operasi kompleks seperti transformasi Fourier. API ini telah teruji dan familiar bagi komunitas Python, termasuk praktisi dan peneliti.
- ***Broadcasting***: Mekanisme *broadcasting* NumPy memungkinkan operasi antara larik dengan bentuk berbeda tanpa perlu replikasi data eksplisit. Potongan kode di bawah menunjukkan contoh *broadcasting* pada NumPy. Fitur ini penting untuk efisiensi memori dan kecepatan komputasi dalam *deep learning*. Pola yang jamak ditemukan adalah penjumlahan hasil transformasi linier (perkalian antara matriks masukan dengan matriks bobot) dan larik bias, `y = x @ w + b`.

```
# Contoh broadcasting NumPy
a = np.array([[1, 2, 3]])      # Bentuk: (1, 3)
```

```
b = np.array([[1], [2], [3]])  # Bentuk: (3, 1)
c = a + b                      # Broadcasting menghasilkan larik berbentuk (3, 3)
print(c)
```

Luaran

```
[[2, 3, 4],
 [3, 4, 5],
 [4, 5, 6]]
```

Walaupun andal, kita tidak bisa secara langsung menggunakan larik NumPy di ranah *deep learning* modern, karena ia hanya bertanggung jawab untuk melakukan komputasi numerik dan tidak memiliki kemampuan untuk melacak gradien. Pendekatan kita adalah membungkus larik NumPy dalam kelas Tensor yang menambahkan kemampuan yang dibutuhkan untuk operasi-operasi *deep learning* dengan tetap memanfaatkan efisiensi NumPy:

```
# Larik NumPy biasa
np_array = np.array([[1., 2.], [3., 4.]])
result = np_array @ np_array.T  # Perkalian matriks

# Tensor kita: NumPy + kemampuan autodiff
tensor = Tensor([[1., 2.], [3., 4.]], requires_grad=True)
result = tensor @ tensor.T()     # Juga perkalian matriks, namun memungkinkan
                                 # untuk eksekusi laluan mundur

result.backward()                # Gradien dihitung otomatis
print(tensor.grad)               # Gradien tersedia!
```

2.3. Kelas Tensor

Kelas ini perlu kita rancang untuk dapat merekam riwayat operasi dan dependensi-dependensinya (berupa tensor masukan) Berikut sketsa permulaan implementasinya.

```
from __future__ import annotations

from typing import Any, Callable, Self

import numpy as np
from numpy.typing import NDArray

BackwardFn = Callable[[NDArray], list[NDArray | None]]

class Tensor:
    def __init__(
        self,
        value: Any,
        requires_grad: bool = False,
    ):
        self.data: NDArray = _ensure_numpy(value)
        self.grad: NDArray | None = None
```

```
        self.requires_grad: bool = requires_grad
        self.backward_fn: BackwardFn | None = None
        self.inputs: list[Tensor] = []
        self.set_requires_grad(requires_grad)

    def set_requires_grad(self, val: bool):
        self.grad = np.zeros_like(self.data) if val else None
        self.requires_grad = val

    @property
    def shape(self) -> Tuple[int]:
        return self.data.shape

    def __repr__(self) -> str:
        return self.data.__repr__()


def _ensure_numpy(value: Any) -> NDArray:
    try:
        value = np.array(value)
        return value
    except Exception:
        raise ValueError(f"Cannot convert value to numpy array: {value}")
```

2.3.1. Anatomi Kelas `Tensor`

Mari kita bedah kelas `Tensor` secara mendetail untuk memahami peran setiap komponen dalam sistem autodiff kita.

2.3.1.1. Atribut Utama

Dua atribut berikut ini adalah inti dari tensor:

```
...
    def __init__(
        ...
    ):
        self.data: NDArray = _ensure_numpy(value)
        self.grad: NDArray | None = None
        ...
...
```

Atribut `data` menyimpan nilai tensor sebenarnya dalam format NumPy array, sedangkan `grad` Menyimpan gradien yang dihitung saat laluan balik. Perlu diperhatikan bahwa jika `grad` tidak bernilai `None`, maka ia akan selalu memiliki **bentuk yang sama dengan** `data`. Mengapa demikian?

Ingat bahwa gradien menunjukkan "sensitivitas" setiap elemen tensor terhadap luarannya. Bayangkan tensor `x` berukuran (2, 3). Jika kita melakukan operasi `y = x * 2`, maka nilai `x.data[i,j]` dipetakan ke `y.data[i,j]`. Gradien `x.grad[i,j]` menunjukkan "sensitivitas", yang artinya, "jika

`x.data[i,j]` berubah, seberapa besar pengaruhnya?" Secara matematis, `x.grad[i,j]` menyimpan $\frac{\partial y}{\partial x_{i,j}}$. Karena ada **tepat** satu nilai gradien untuk setiap elemen data, maka haruslah benar bahwa `x.grad.shape == x.data.shape`.

2.3.1.2. Mekanisme Pelacakan Gradien

Tiga atribut ini mengontrol sistem autodiff kita:

```
...
    def __init__(
        ...
    ):
        ...
        self.requires_grad: bool = requires_grad
        self.inputs: list[Tensor] = []
        self.backward_fn: BackwardFn | None = None
        ...
...
```

Atribut `requires_grad` adalah penanda yang menentukan apakah tensor ini perlu dihitung gradiennya. Tensor yang demikian, misalnya tensor bobot (*weight*) dan tensor bias untuk keperluan melatih model *neural network*. Namun, tidak semua tensor perlu gradien, misalnya, data masukan atau skalar konstanta.

Atribut `inputs` akan melacak daftar masukan dari suatu operasi. Misal, pada `c = a + b`, `c.inputs` akan berisi `[a, b]`. Pada operasi `y = exp(x)`, `y.input` akan berisi `[x]`. Seperti yang kita ulas di bab sebelumnya, penting bagi kita untuk melacak riwayat masukan dari satu operasi ke operasi lain untuk implementasi autodiff, khususnya autodiff eksak.

Atribut `backward_fn` mengimplementasikan prosedur penghitungan gradien untuk tensor-tensor yang menjadi masukan tensor ini. Misalnya, jika tensor ini hasil dari `a + b`, maka `backward_fn` tahu cara mendistribusikan gradien ke `a` dan `b`. Jika tensor ini hasil dari `a * b`, maka `backward_fn` tahu harus mengalikan gradien dengan pasangan yang sesuai (`a.grad` dikalikan nilai `b.data`, gradien `b.grad` dikalikan nilai `a.data`)

2.3.1.3. Metode dan Fungsi Bantuan

Metode `shape` hanya *syntactic sugar* yang memudahkan akses ke informasi bentuk larik NumPy yang mendasarinya. Dengan ini, untuk mengakses `shape` dari data, kita cukup memanggil `tensor.shape` alih-alih `tensor.data.shape`.

Fungsi `_ensure_numpy_` bertugas mengonversi masukan apapun menjadi larik NumPy. Ini memungkinkan fleksibilitas dalam membuat tensor:

```
# Semua ini valid:
t1 = Tensor(5)                  # Dari skalar
t2 = Tensor([1, 2, 3])          # Dari larik Python biasa
t4 = Tensor([[1, 2], [3, 4]])   # Dari larik Python bersarang
t3 = Tensor(np.eye(3))          # Dari larik NumPy
```

Metode `set_requires_grad` mengalokasikan atau mendealokasikan buffer gradien. Jika argumen bernilai `True`, alokasikan `grad` dengan zeros (siap diakumulasi) dan atur `requires_grad` menjadi `True`. Sebaliknya, atur `grad` menjadi `None` dan `requires_grad` menjadi `False`.

Metode repr adalah fungsi kenyamanan saja untuk memperoleh representasi `str` yang informatif, terutama saat kita mencetak nilai tensor di layar. Implementasi lengkap biasanya menampilkan data dan status gradien

Catatan

Pembaca dibebaskan untuk menentukan struktur proyek pustaka ini. Penulis berasumsi pembaca telah familiar dengan struktur kode proyek python. Kode referensi buku ini sendiri mengacu pada rekomendasi pustaka `uv`[4] yang menggunakan `pyproject` untuk manajemen *dependency*. Secara garis besar (tanpa mengulasnya kembali di bagian-bagian selanjutnya), berikut ini struktur implementasi referensi:

```
lantern/
├── examples/
│   └── linear_regression/
│       └── main.py
├── lantern/
│   ├── __init__.py
│   ├── tensor.py
│   ├── ...
│   ├── ...
│   └── nn.py
├── tests/
│   ├── __init__.py
│   └── test_tensor.py
├── pyproject.toml
├── README.md
└── uv.lock
```

2.4. Sistem Penghitungan Turunan Otomatis

Gradien menentukan bagaimana luaran suatu fungsi berubah terhadap masukannya. Pada konteks *deep learning*, nilai gradien akan menentukan seberapa banyak kita mengubah parameter pada model.

Saat melatih model *neural network*, gradien akan dihitung dengan mencari turunan nilai *loss* terhadap parameter model (berupa `NDArray` pada atribut `data` kelas `Tensor`). Model *deep learning* belajar dengan cara mengatur parameter-parameter di dalamnya berdasarkan seberapa besar nilai *loss* yang diperolehnya. Seberapa banyak pengaturan parameter yang perlu dilakukan, ditentukan oleh gradien. Gradien ini akan "memandu" proses pembaruan parameter sehingga parameter tersebut bisa meminimalkan galat.

Pada konteks *deep learning*, kita perlu menghitung gradien fungsi *loss* terhadap masing-masing parameter. Hal ini sangat tidak mungkin dilakukan secara manual satu per satu mengingat model-model *deep learning* memiliki ribuan hingga jutaan parameter (bahkan miliaran untuk model-model kiwari). Maka dari itu, kita membutuhkan suatu cara untuk menghitung gradien secara otomatis.

[4] https://docs.astral.sh/uv/

Di sinilah sistem autodiff berperan. Sistem autodiff menyediakan cara sistematis untuk menghitung gradien **eksak** dengan cara memecah komputasi yang kompleks menjadi operasi-operasi elementer yang lebih sederhana. Tiap operasi tersebut mengimplementasikan aturan penghitungan gradiennya sendiri. Sistem autodiff kemudian mengakumulasi gradien-gradien tiap operasi untuk menghitung gradien keseluruhan model.

Catatan

Kita bisa mengimplementasikan autodiff dalam dua cara: metode maju (*forward*) dan mundur (*backward*). Metode maju melakukan penghitungan gradien bersamaan dengan proses laluan maju (*forward pass*). Sementara itu, metode mundur melakukan laluan maju terlebih dahulu dan "merekam" operasi-operasi yang terlibat. Kemudian, ia mempropagasikan gradien ke belakang operasi yang terekam tersebut. Di buku ini penulis fokus pada implementasi metode mundur yang juga digunakan oleh pustaka *deep learning* seperti PyTorch.

Fondasi utama proses kerja autodiff adalah aturan rantai. Aturan rantai adalah metode kalkulus yang digunakan untuk mencari turunan dari fungsi komposisi (fungsi yang dibentuk dari fungsi lain Aturan rantai menjelaskan cara menghitung gradien melalui fungsi komposisi dengan mengalikan turunan-turunan sepanjang alur komputasi. Sebagai contoh, jika kita memiliki untaian operasi seperti $y = h(g(x))$, aturan rantai menetapkan

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x} \tag{2.1}$$

Di mana $u = g(x)$. Pustaka kita akan memiliki fitur autodiff yang menerapkan prinsip aturan rantai secara otomatis.

Contoh: *Neural network* sebagai fungsi komposisi

Neural network adalah contoh fungsi komposisi. Suatu *neural network* terdiri dari deretan lapisan, di mana tiap lapisan merupakan suatu fungsi. Lapisan 1 menerima masukan, melakukan transformasi (*fully connected*), mengumpankannya ke lapisan 2 (fungsi aktivasi), kemudian lapisan 2 mengumpankannya ke lapisan selanjutnya, dan seterusnya. Secara matematis, *neural network* n-lapis dengan masukan x diekspresikan dengan $f(x) = f_n(f_{n-1}(\dots f_1(x)))$.

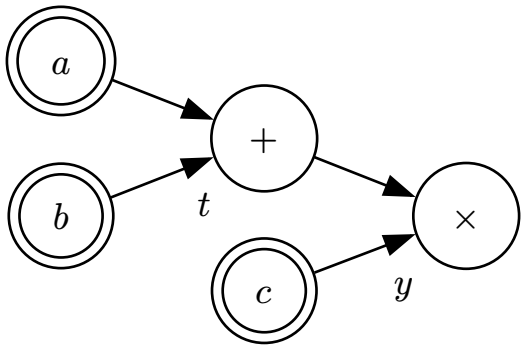
2.5. Operasi Aritmetika Sebagai Proses Konstruksi Graf

Untuk mengimplementasikan autodiff, kita perlu cara untuk melacak semua operasi yang dilakukan pada tensor. Bayangkan kita perlu menghitung gradien secara manual untuk ekspresi matematika yang kompleks dan perlu mengingat setiap langkah komputasi dan hubungannya. Ini tidak realistis untuk operasi tensor yang kompleks, terutama saat mengembangkan model *deep learning*.

Untuk menerapkan aturan rantai pada ribuan parameter, kita perlu cara untuk: 1) melacak urutan operasi yang dilakukan; 2) menyimpan informasi tentang bagaimana menghitung turunan setiap operasi; dan 3) mengetahui dependensi antar operasi. Solusinya adalah dengan membangun

graf komputasi. Setiap kali pengguna melakukan operasi aritmetika, pustaka kita merekam operasi tersebut dalam struktur graf di belakang layar. Graf ini nantinya menjadi "peta" untuk propagasi gradien saat laluan balik.

Mari kita amati bagaimana ekspresi matematika sederhana $y = (a + b) \times c$ dapat direpresentasikan sebagai graf komputasi. Ekspresi ini memiliki dua operasi: penjumlahan menghasilkan nilai antara, kemudian perkalian menghasilkan hasil akhir.



Gambar 1: Operasi aritmetika $y = (a + b) \times c$ sebagai representasi graf

Proses pembentukan graf adalah sebagai berikut:

1. Saat operasi $a + b$ dilakukan:
   - Tensor baru `t` dibuat dengan nilai `a.data + b.data`
   - Tensor `t` menyimpan `[a, b]` pada atribut `inputs`
   - Tensor `t` menyimpan fungsi untuk propagasi gradien penjumlahan
2. Saat operasi $t \times c$ dilakukan:
   - Tensor baru (`y`) dibuat dengan nilai `t.data * c.data`
   - Tensor `y` menyimpan `[t, c]` pada atribut `inputs`
   - Tensor `y` menyimpan fungsi untuk propagasi gradien perkalian

Maka, graf terbentuk secara implisit melalui referensi dalam atribut `inputs`. Tidak perlu kelas khusus atau struktur graf terpisah, karena graf terbentuk dalam hubungan antar tensor. Dengan fondasi ini, kita siap mengimplementasikan operasi-operasi dasar yang akan membangun graf komputasi kita.

Catatan

Nantinya kita akan menambahkan *operator overloading* (`__add__` , `__mul__`) agar aktivitas menulis kode menjadi nyaman, dan pengguna bisa menulis `a + b` alih-alih menulis `add(a, b)` , `mul(a, b)` , dan seterusnya.

2.6. Laluan Mundur

Laluan mundur adalah langkah di maka kita menghitung gradien dengan menelusuri graf dari simpul luaran ke semua simpul masukan. Sebelum mengimplementasikannya, mari ingat kembali intuisi di balik laluan mundur. Misalkan kita sedang menghitung turunan fungsi komposisi $f(g(h(x)))$. Secara manual, kita akan:

1. Menghitung turunan terluar: $f'(g(h(x)))$

2. Mengalikan dengan turunan lapisan berikutnya: $f'(g(h(x)))c \cdot g'(h(x))$
3. Terus mengalikan hingga mencapai variabel awal: $f'(g(h(x)))c \cdot g'(h(x))c \cdot h'(x)$

Autodiff mengotomatisasi proses ini saat laluan mundur. Gradien "mengalir" dari simpul luaran ke simpul masukan, ujung ke ujung, dengan setiap operasi bertanggung jawab mendistribusikan gradien yang diterimanya ke operasi-operasi pendahulunya. simpul-simpul dalam graf akan diproses dengan urutan secara **topologis**.

Berikut implementasi pengurutan simpul secara topologis.

```
def _topo_sort(root: Tensor):
    visited: set[Tensor] = set()
    topo_order: list[Tensor] = []

    def _build(node: Tensor):
        if node not in visited:
            visited.add(node)
            if node.backward_fn is not None:
                for input in node.inputs:
                    _build(input)
            topo_order.append(node)

    _build(root)
    return topo_order
```

Fungsi ini menggunakan *depth-first search* (DFS) dengan sedikit modifikasi. Alur kerjanya:

1. Mulai dari simpul akar (*root*, biasanya tensor nilai *loss*/simpul luaran paling ujung)
2. Rekursi ke masukan-masukan simpulnya (dependensi dari simpul). Untuk setiap simpul, kunjungi dulu semua dependensinya
3. Setelah semua dependensi dikunjungi, tambahkan simpul ke daftar simpul yang telah diproses

Hasilnya adalah daftar simpul, di mana setiap simpul diletakkan setelah simpul-simpul masukannya (simpul yang bergantung padanya). Saat kita membalik urutan ini (`reversed(topo_order)`), kita mendapat urutan yang benar untuk propagasi gradien: dari luaran ke masukan, ujung ke ujung.

Mari kita modifikasi kelas `Tensor` dengan menambah metode `backward` dan kita telaah bagian per bagian.

```
class Tensor:
    ...

    def backward(self, upstream_grad: NDArray | None = None):
        if not self.requires_grad:
            return

        if upstream_grad is None:
            upstream_grad = np.ones_like(self.data)

        self.grad = upstream_grad
        topo_order = _topo_sort(self)
```

```
        # Akumulasi gradien
        for node in reversed(topo_order):
            if node.backward_fn is None or node.grad is None:
                continue

            inputs = node.inputs
            grads = node.backward_fn(node.grad)
            for input, input_grad in zip(inputs, grads):
                if input_grad is not None:
                    input.grad += input_grad

    ...
```

Pada metode ini, pertama, jika tensor tidak memerlukan gradien, tidak ada yang perlu dilakukan.

```
        ...
        if not self.requires_grad:
            return
        ...
```

Kemudian, untuk kasus khusus output (biasanya nilai *loss* (*loss*)) gradien terhadap dirinya sendiri adalah 1. Ini titik awal propagasi gradien kita.

```
        ...
        if upstream_grad is None:
            upstream_grad = np.ones_like(self.data)
        ...
```

Dengan urutan topologis terbalik kita lakukan iterasi untuk pemrosesan tiap simpul. Lompati simpul yang tidak memiliki fungsi backward (simpul daun) atau tidak memiliki gradien. Setelahnya, kita panggil metode `backward_fn` dari tiap simpul. Di sinilah "sihir" terjadi.

Metode `backward_fn` mengendalikan bagaimana cara menghitung gradien lokal. Ia menerima gradien *upstream* (dari operasi selanjutnya) dan menghitung gradien untuk setiap masukan operasi ini.

```
        ...
        # Akumulasi gradien
        for node in reversed(topo_order):
            if node.backward_fn is None or node.grad is None:
                continue

            inputs = node.inputs
            grads = node.backward_fn(node.grad)
            for input, input_grad in zip(inputs, grads):
                if input_grad is not None:
                    input.grad += input_grad
        ...
```

Perhatikan **bagian penting** ini: kita menambahkan (`+=`) gradien, bukan mengganti (`=`). Mengapa? Karena satu tensor bisa saja digunakan di beberapa operasi, dan setiap penggunaan berkontribusi pada gradien total.

Catatan

metode `backward_fn` berisi logika spesifik untuk menghitung gradien lokal setiap operasi. Operasi-operasi akan menyimpan fungsi ini saat laluan maju. Detail implementasinya akan kita lihat saat membahas operasi penjumlahan, perkalian, dan lainnya.

Pendalaman

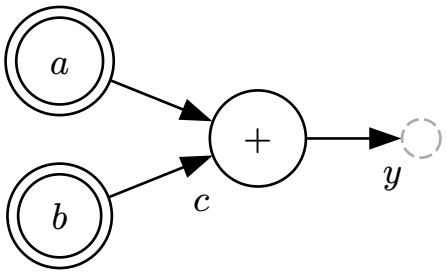
Dengan jumlah simpul (tensor) V dan jumlah koneksi E, kompleksitas waktu laluan mundur kita adalah $O(V + E)$ karena tiap simpul dan koneksi dikunjungi **tepat satu kali**. Kompleksitas memori prosedur ini $O(V)$ untuk menyimpan urutan topologis, namun tidak terlalu nampak karena jauh didominasi oleh penyimpanan tensor-tensor pada keseluruhan operasi.

2.7. Implementasi Operasi Pertama: Penjumlahan

Dengan infrastruktur laluan balik yang sudah siap, mari implementasikan operasi pertama: penjumlahan (`add`). Fungsi `add` mengimplementasikan laluan maju dan mundur untuk operasi penjumlahan.

```
def add(a: Tensor, b: Tensor) -> Tensor:
    def backward_fn(upstream_grad: NDArray):
        a_grad = upstream_grad if a.requires_grad else None
        b_grad = upstream_grad if b.requires_grad else None
        return [a_grad, b_grad]

    result = Tensor(a.data + b.data, requires_grad=a.requires_grad or b.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [a, b]
    return result
```

Perhatikan bahwa hasil akan melacak gradien (`requires_grad=True`) jika salah satu masukannya melacak gradien. Hal ini memastikan gradien bisa mengalir mundur melalui operasi ini jika diperlukan. Operasi `add` ditunjukkan oleh Gambar 2, dengan simpul bergaris putus-putus yang merupakan kemungkinan simpul selanjutnya.



Gambar 2: Representasi simpul penjumlahan

$$\begin{aligned}\frac{\partial y}{\partial a} &= \frac{\partial y}{\partial c} \cdot \frac{\partial c}{\partial a} \\ &= \frac{\partial y}{\partial c} \cdot \frac{\partial (a+b)}{\partial a} \\ &= \frac{\partial y}{\partial c} \qquad (\rightarrow \text{gradien } \textit{upstream})\end{aligned} \tag{2.2}$$

dan untuk `b.grad`:

$$\begin{aligned}\frac{\partial y}{\partial b} &= \frac{\partial y}{\partial c} \cdot \frac{\partial c}{\partial b} \\ &= \frac{\partial y}{\partial c} \cdot \frac{\partial (a+b)}{\partial b} \\ &= \frac{\partial y}{\partial c} \qquad (\rightarrow \text{gradien } \textit{upstream})\end{aligned} \tag{2.3}$$

Dari dua persamaan di atas, kita tahu bahwa turunan penjumlahan selalu 1. Di implementasinya, ini berarti `backward_fn` cukup meneruskan `upstream_grad` langsung ke kedua input. Variabel kembalian `a_grad` akan menjadi gradien lokal untuk tensor masukan `a`, begitu juga variabel kembalian `b`. Saat pemanggilan `backward()`, `a_grad` akan diakumulasikan dengan `a.grad`.

Catatan

Biasanya, dalam pengembangan model *deep learning*, kita akan bekerja dengan `Tensor` dengan data internal yang memiliki beberapa nilai sekaligus. Di situasi ini, semua gradien dari output terhadap masing-masing nilai akan dihitung juga sekaligus. Misal, untuk operasi penjumlahan tensor peringkat dua (matriks) $\mathbf{A} \in \mathbb{R}^{2\times 2}$ dan $\mathbf{B} \in \mathbb{R}^{2\times 2}$ dengan $\forall a_{ij} \in \mathbf{A}$ dan $\forall b_{ij} \in \mathbf{B}$ kita memperoleh

$$\begin{aligned}\mathbf{Y} &= \mathbf{A} + \mathbf{B} \\ &= \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \\ &= \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} \\ a_{21}+b_{21} & a_{22}+b_{22} \end{pmatrix}\end{aligned} \tag{2.4}$$

dengan $\forall y_{ij} \in \mathbf{Y}$. Jika kita ingin mencari gradien seluruh elemen $\mathbf{Y}$ terhadap seluruh elemen $\mathbf{A}$ yang saling berkorespondensi, maka karena setiap elemen luaran $y_{ij} = a_{ij} + b_{ij}$, kita bisa memperoleh $\frac{\partial y_{ij}}{\partial a_{ij}} = 1$. Sehingga, gradien yang dihitung untuk matriks $\mathbf{A}$ adalah matriks berukuran sama dengan $\mathbf{A}$ dengan setiap elemennya bernilai 1:

$$\begin{aligned}\begin{pmatrix} \frac{\partial y_{11}}{\partial a_{11}} & \frac{\partial y_{12}}{\partial a_{12}} \\ \frac{\partial y_{21}}{\partial a_{21}} & \frac{\partial y_{22}}{\partial a_{22}} \end{pmatrix} &= \begin{pmatrix} \frac{\partial (a_{11}+b_{11})}{\partial a_{11}} & \frac{\partial (a_{12}+b_{12})}{\partial a_{12}} \\ \frac{\partial (a_{21}+b_{21})}{\partial a_{21}} & \frac{\partial (a_{22}+b_{22})}{\partial a_{22}} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}\end{aligned} \tag{2.5}$$

Perhatikan contoh dengan nilai tensor berupa larik berikut ini.

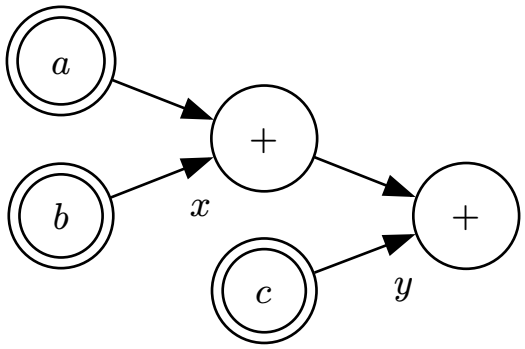
```
a = Tensor([1.0, 2.0, 3.0], requires_grad=True)
b = Tensor([4.0, 5.0, 6.0], requires_grad=True)
y = add(a, b)

y.backward()
print(f"{a.grad=}")
print(f"{b.grad=}")
```

Luaran

```
a.grad=array([1., 1., 1.])
b.grad=array([1., 1., 1.])
```

Hasilnya sesuai harapan: $\frac{\partial y}{\partial a} = 1$ dan $\frac{\partial y}{\partial b} = 1$ untuk setiap elemen.

Kita bahkan bisa melakukan operasi yang lebih kompleks, misalnya, penjumlahan tiga variabel $y = a + b + c$. Kekuatan autodiff muncul saat kita merangkai operasi yang demikian (dan yang lebih kompleks). Graf komputasi yang terbentuk sebagai berikut.



Gambar 3: Representasi graf penjumlahan tiga variabel $y = a + b + c$

Berikut ini implementasinya dalam kode Python:

```
a = Tensor([1.0], requires_grad=True)
b = Tensor([2.0], requires_grad=True)
c = Tensor([3.0], requires_grad=True)

x = add(a, b)  # a + b
y = add(x, c)  # (a + b) + c
```

```
y.backward()
print(f"{a.grad=}")
print(f"{b.grad=}")
print(f"{c.grad=}")
```

dengan luaran berikut ini:

Luaran

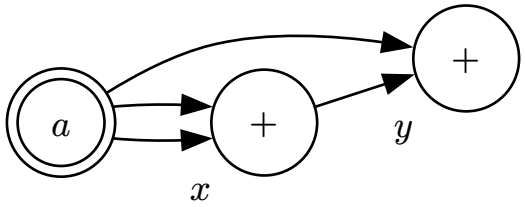
```
a.grad=array([1.])
b.grad=array([1.])
c.grad=array([1.])
```

Gradien mengalir dari y ke x dan c, kemudian dari x ke a dan b. Setiap variabel menerima gradien 1, sesuai dengan turunan penjumlahan.

Hal menarik terjadi saat variabel yang sama muncul beberapa kali dalam komputasi. Perhatikan contoh di bawah.

Contoh: Penggunaan variabel berulang

Diberikan $y = a + a + a$ dengan representasi graf beserta kodenya di bawah ini.



Gambar 4: Representasi graf penggunaan variabel berulang $y = a + a + a$

```
a = Tensor([2.0], requires_grad=True)
x = add(a, a)  # a + a
y = add(x, a)  # (a + a) + a

# Tentu saja kita bisa menulis y=add(add(a, a), a). Namun, untuk mempermudah
penjelasan
# dan visualisasi, penulis menggunakan variabel perantara `x`.

result.backward()
print(f"{a.grad=}")
```

Luaran

```
a.grad=array([3.])
```

Kita memperoleh jawaban yang benar!

Secara matematis, mudah saja kita memahami mengapa `a.grad` bernilai 3. Seperti kita ketahui, `a.grad` merepresentasikan $\frac{\partial y}{\partial a}$, yang berarti

$$\begin{aligned}\frac{\partial y}{\partial a} &= \frac{\partial(a+a+a)}{\partial a} \\ &= \frac{\partial(3a)}{\partial a} \\ &= 3\end{aligned} \tag{2.6}$$

Mudah pula untuk kita memahami dengan pendekatan graf komputasi. Jika kita perhatikan Gambar 4, simpul a terkoneksi ke simpul luaran y melalui tiga jalur berbeda dalam graf komputasi. Jalur pertama adalah $a \to y$ (langsung menuju simpul luaran), jalur kedua adalah $a \to x \to y$, dan jalur ketiga adalah $a \to x \to y$ namun dengan koneksi yang berbeda dari a ke x.

1. Jalur langsung $a \to y$ memberi kontribusi gradien 1
2. Jalur $a \to x \to y$, dengan tepi $a \to x$ yang pertama, memberi kontribusi gradien 1
3. Jalur $a \to x \to y$, dengan tepi $a \to x$ yang kedua, memberi kontribusi gradien 1

Jika dijumlahkan seluruhnya, $1 + 1 + 1 = 3$, yang konsisten dengan $\frac{\partial(3a)}{\partial a} = 3$ Contoh di atas adalah alasan mengapa kode kita menggunakan operator penetapan-penjumlahan (`+=`) pada fungsi `add_backward` (dan juga di fungsi lainnya nanti). Jika kita menggunakan operator penetap biasa (`=`) alih-alih `+=` , gradien awal akan ditimpa oleh gradien yang diperoleh dari jalur lain. Ini hal yang tidak kita harapkan! Karenanya, kontribusi gradien dari tiap jalur **harus** diakumulasikan untuk memperoleh gradien keseluruhan.

Catatan

Pola akumulasi gradien ini akan muncul di semua operasi. Selalu gunakan `+=` untuk memastikan gradien dari berbagai jalur diakumulasikan dengan benar.

Tidak semua tensor perlu melacak gradien. Konstanta atau input yang tidak perlu dioptimasi bisa menghemat memori:

```
a = Tensor([1.0, 2.0, 3.0], requires_grad=True)
b = Tensor([4.0, 5.0, 6.0])  # requires_grad=False secara default
y = add(a, b)

y.backward()
print(f"{a.grad=}")
print(f"{b.grad=}")
```

Luaran

```
a.grad=array([1., 1., 1.])
b.grad=None
```

Salah satu keunggulan sistem autodiff adalah kemampuan membangun operasi kompleks dari operasi yang lebih sederhana. Mari kita lihat bagaimana perkalian bisa disimulasikan menggunakan penjumlahan berulang:

```
a = Tensor([[1, 2], [3, 4]], requires_grad=True)

# b = a * 8
#   = a + a + a ... + a (8 kali)
b = Tensor(0)
for _ in range(8):
    b = add(b, a)
b.backward()

print("a:")
print(a)
print("a.grad:")
print(a.grad)
```

Luaran

```
a:
Tensor([[1 2]
        [3 4]], requires_grad=True)
a.grad:
[[8. 8.]
 [8. 8.]]
```

Perhatikan bagaimana kita "membangun" operasi perkalian $b = 8a$ hanya dengan menggunakan operasi penjumlahan. Sistem autodiff kita secara otomatis menangani kompleksitas ini. Setiap iterasi menambahkan jalur baru dalam graf komputasi, dan semua kontribusi gradien terakumulasi dengan benar. Hasilnya, gradien `a` adalah 8, persis seperti yang kita harapkan dari $\frac{\partial(8a)}{\partial a} = 8$.

Catatan

Tentu saja, pada praktiknya kita akan mengimplementasikan operasi perkalian yang lebih layak di bab selanjutnya yang jauh lebih efisien daripada penjumlahan iteratif. Namun contoh ini menunjukkan prinsip penting bahwa operasi kompleks bisa dibangun dari operasi sederhana, dan sistem autodiff akan menghitung gradien dengan benar selama setiap operasi dasar mengimplementasikan aturan gradiennya dengan tepat.

Prinsip komposisi ini adalah fondasi deep learning modern. Lapisan *neural network* yang kompleks sekalipun pada dasarnya adalah komposisi dari operasi-operasi elementer seperti penjumlahan, perkalian, dan fungsi aktivasi.

2.8. Fitur penunjang kenyamanan

Bermatematika dalam kode akan lebih mudah bila kita bisa menulis dengan alami, seperti halnya menulis persamaan di atas kertas. Python membolehkan kita untuk melakukan *operator overloading*, sehingga alih-alih menulis `y = add(add(a, b), c)` kita bisa menulis `y = a + b + c` (notasi infiks). Kelas `Tensor` bisa dimodifikasi.

```
class Tensor:
```

```
    ...

    def __repr__(self) -> str:
        return self.data.__repr__()

    def __add__(self, other):
        return add(self, other)

    def __radd__(self, other):
        other = _ensure_tensor(other)
        return add(other, self)


def _ensure_tensor(value: Any) -> Tensor:
    if isinstance(value, Tensor):
        return value
    try:
        value = Tensor(value)
        return value
    except Exception:
        raise ValueError(f"Cannot convert value to Tensor: {value}")
```

Sekarang sekarang kita bisa melakukan ini:

```
a = Tensor([1], requires_grad=True)
y = a + a + a

y.backward()
print(a.grad)
```

Luaran

```
[3]
```

Di bab selanjutnya kita akan mengimplementasikan hal serupa untuk operator lain seperti `sub`, `mul`, dan lainnya.

Bab 3. Operasi Tensor

Pada bab sebelumnya, kita telah membangun fondasi sistem autodiff dengan kelas `Tensor` dan operasi penjumlahan sederhana. Namun, implementasi kita masih memiliki kelemahan fundamental, salah satunya, ketidakmampuan menangani *broadcasting* dengan benar saat menghitung gradien. Bab ini akan memperbaiki kelemahan tersebut dan memperluas pustaka kita dengan berbagai operasi tensor yang penting untuk *deep learning*.

3.1. Menangani *Broadcasting* untuk Operasi Biner Per Elemen (*Element-Wise Binary Operation*)

Broadcasting adalah seperangkat aturan dalam melakukan operasi larik multidimensi dengan bentuk berbeda. Ide utamanya adalah, larik yang kecil akan dibentangkan (seolah direplikasi) agar bentuknya menyesuaikan larik yang lebih besar. Secara konseptual, *broadcasting* bekerja dengan "menggandakan" tensor yang lebih kecil hingga bentuknya kompatibel dengan tensor yang lebih besar. Pustaka NumPy sudah menangani *broadcasting*. Contoh:

```
import numpy as np
a = np.array([1.0, 2.0, 3.0])
b = 2.0
print(a * b)
```

Luaran

```
array([2.,  4.,  6.])
```

Jika diilustrasikan, hal ini yang seolah terjadi:

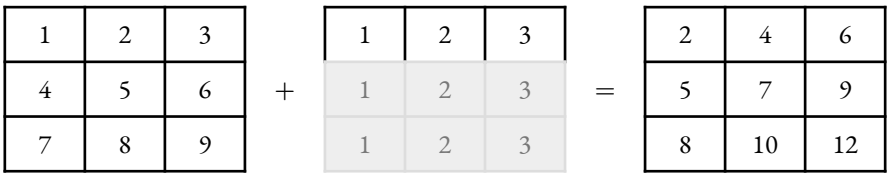
Gambar 5: Ilustrasi perkalian dua larik berbeda bentuk dengan mekanisme *broadcasting*. Larik `b` akan digandakan hingga memiliki bentuk kompatibel dengan `a`

Contoh lain *broadcasting* untuk larik dimensi dua dan satu, di mana larik salah satunya (secara konseptual) digandakan sepanjang dimensi baris, dan diilustrasikan oleh Gambar 6:

```
a = np.array([[1,2,3], [4,5,6], [7,8,9]])
b = np.array([1,2,3])
print(a + b)
```

Luaran

```
array([[ 2,  4,  6],
       [ 5,  7,  9],
       [ 8, 10, 12]])
```



Gambar 6: Larik `b` akan digandakan sepanjang dimensi baris hingga memiliki bentuk kompatibel dengan `a`

Catatan

Pembentangan yang nampak seperti replikasi ini sebetulnya konseptual saja. Implementasi NumPy dalam bahasa C sangat efisien, sehingga tidak ada proses *copy* data eksak yang terjadi di memori.

3.1.1. Kelemahan Implementasi Kita

Operasi penjumlahan yang kita implementasikan di Bab 2 sudah dapat menangani *broadcasting* untuk laluan maju karena NumPy menanganinya secara otomatis. Namun, ada masalah krusial pada saat laluan mundur. Perhatikan contoh berikut:

Contoh: Galat operasi tanpa penanganan *broadcasting*

```
# Menggunakan implementasi add dari Bab 2 (tanpa unbroadcast)
a = Tensor([[1, 2, 3], [4, 5, 6]], requires_grad=True)  # Shape: (2, 3)
b = Tensor([10, 20, 30], requires_grad=True)            # Shape: (3,)

c = add(a, b)  # c = [[11, 22, 33], [14, 25, 36]], shape: (2, 3)

# Lihat apa yang terjadi saat backward
c.backward()  # upstream_grad di sini berbentuk (2, 3)
```

Luaran

```
ValueError: non-broadcastable output operand with shape (3,) doesn't
match the broadcast shape (2,3)
```

Galat ini terjadi karena kita mencoba menambahkan larik gradien *upstream* berbentuk `(2, 3)` (`upstream_grad`) ke `b.grad` yang berbentuk `(3,)`. NumPy tidak bisa melakukan operasi ini karena aturan *broadcasting* tidak berlaku untuk operasi `+=` ketika operand kiri memiliki bentuk yang tidak kompatibel.

Perhatikan implementasi kita yang masih naif:

```
def add(a: Tensor, b: Tensor) -> Tensor:
    def backward_fn(upstream_grad: NDArray):
        a_grad = upstream_grad if a.requires_grad else None  # a_grad.shape: (2, 3)
        b_grad = upstream_grad if b.requires_grad else None  # b_grad.shape: (2, 3)
        return [a_grad, b_grad]
```

```
    ...
...
```

Untuk contoh di atas, saat pemanggilan `backward` (yang juga mengeksekusi `backward_fn`), `a_grad` memiliki bentuk $(2, 3)$, begitu pun `b_grad` karena berasal dari `c.grad` mula-mula yang juga berukuran $(2, 3)$. Saat tahapan akumulasi gradien:

```
    def backward(self, upstream_grad: NDArray | None = None):
        ...

        # Akumulasi gradien
        for node in reversed(topo_order):
            ...
            for input, input_grad in zip(inputs, grads):
                if input_grad is not None:
                    input.grad += input_grad # a.grad (2, 3) += a_grad (2, 3) -> OK
                                             # b.grad (3,) += b_grad (2, 3) -> ERROR
```

Saat akumulasi `a.grad += a_grad`, tak ada masalah dengan kompatibilitas bentuk. Namun, saat akumulasi `b.grad += b_grad`, terjadi inkompatibilitas bentuk yang menyebabkan *run time error*.

Yang **seharusnya** terjadi adalah:

- **`a.grad`**: memiliki `shape` berupa `(2, 3)` dan dengan nilai `array([[1, 1, 1], [1, 1, 1]])`, yang saat ini sudah benar
- **`b.grad`**: memiliki `shape` berupa `(3,)` dan dengan nilai `np.array([2, 2, 2])`. Saat ini implementasi kita tidak memperoleh hasil yang seharusnya.

Mengapa `b.grad` seharusnya `[2, 2, 2]`? Karena ketika `b` di-broadcast dalam operasi maju:

- `b[0] = 10` berkontribusi bagi luaran di posisi `c[0,0]` dan `c[1,0]`
- `b[1] = 20` berkontribusi bagi luaran di posisi `c[0,1]` dan `c[1,1]`
- `b[2] = 30` berkontribusi bagi luaran di posisi `c[0,2]` dan `c[1,2]`

Maka, gradien untuk setiap elemen `b` seharusnya adalah jumlah gradien dari semua posisi di mana elemen tersebut digunakan:

```
b_grad[0] = c_grad[0,0] + c_grad[1,0] = 1 + 1 = 2
b_grad[1] = c_grad[0,1] + c_grad[1,1] = 1 + 1 = 2
b_grad[2] = c_grad[0,2] + c_grad[1,2] = 1 + 1 = 2
```

Solusi permasalahan ini adalah dengan **"membalik"** efek *broadcasting* pada gradien. Jika suatu tensor di-*broadcast* selama laluan maju, gradiennya harus dijumlahkan sepanjang dimensi yang di-*broadcast* selama operasi laluan mundur sehingga memiliki dimensi yang sesuai dengan data tensornya.

3.1.2. Membalik Efek *Broadcasting* dengan "*Unbroadcasting*"

Ketika tensor di-*broadcast* selama laluan maju, ada dua kemungkinan transformasi yang terjadi:

1. Penambahan dimensi baru: Tensor dengan dimensi lebih sedikit mendapat dimensi tambahan di sebelah kiri
2. Pembentangan dimensi *singleton*: Dimensi berukuran 1 dibentangkan menjadi lebih besar

Masalah ini bisa diatasi dengan mengimplementasikan fungsi `_unbroadcast_grad`. Fungsi ini bertugas "membalik" efek *broadcasting* yang terjadi selama laluan maju dengan menjumlahkan gradien sepanjang dimensi-dimensi yang di-*broadcast*.

```
def _unbroadcast_grad(grad: NDArray, shape: tuple[int, ...]) -> NDArray:
    # Kasus 1: Menangani perbedaan jumlah dimensi
    # Jika grad memiliki lebih banyak dimensi, jumlahkan dimensi tambahan
    ndim_add = grad.ndim - len(shape)
    for _ in range(ndim_add):
        grad = grad.sum(axis=0)

    # Kasus 2: Menangani dimensi singleton (berukuran 1)
    # Jumlahkan sepanjang dimensi yang di-broadcast dari 1
    for i, dim in enumerate(shape):
        if dim == 1:
            grad = grad.sum(axis=i, keepdims=True)

    # Atribut `grad.shape` sekarang sudah sesuai dengan nilai argumen `shape`
    return grad
```

Fungsi `_unbroadcast_grad` di atas dapat menangani semua kasus broadcasting dengan hanya beberapa baris kode.

3.2. Pemutakhiran Fungsi Penjumlahan

Dengan fungsi `_unbroadcast_grad`, kita dapat memperbaiki implementasi operasi penjumlahan:

```
def add(a: Tensor, b: Tensor) -> Tensor:
    def backward_fn(upstream_grad: NDArray):
        # Penyesuaian upstream_grad terhadap bentuk a.grad sebelum akumulasi
        a_grad = _unbroadcast_grad(
            upstream_grad, a.shape
        ) if a.requires_grad else None
        # Penyesuaian upstream_grad terhadap bentuk b.grad sebelum akumulasi
        b_grad = _unbroadcast_grad(
            upstream_grad, b.shape
        ) if b.requires_grad else None
        return [a_grad, b_grad]

    result = Tensor(a.data + b.data, requires_grad=a.requires_grad or b.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [a, b]
    return result
```

Mari kita uji implementasi yang telah diperbaiki dengan berbagai contoh kasus.

Contoh: Penjumlahan tanpa broadcasting

```
a = Tensor([[1, 2], [3, 4]], requires_grad=True)
b = Tensor([[5, 6], [7, 8]], requires_grad=True)

c = add(a, b)  # [[6, 8], [10, 12]]
c.backward()

print(f"a.grad:\n{a.grad}")
print(f"b.grad:\n{b.grad}")
```

Luaran

```
a.grad:
[[1. 1.]
 [1. 1.]]
b.grad:
[[1. 1.]
 [1. 1.]]
```

Tanpa broadcasting, gradien langsung diteruskan tanpa perubahan.

Contoh: Broadcasting vektor ke matriks

Skenario ini cukup umum terjadi di *neural network* saat menambahkan vektor bias ke hasil transformasi linier. Anggap `a` adalah tensor hasil perkalian matriks masukan dan matriks bobot.

```
a = Tensor([[1, 2, 3], [4, 5, 6]], requires_grad=True)  # (2, 3)
b = Tensor([10, 20, 30], requires_grad=True)            # (3,)

c = add(a, b)  # [[11, 22, 33], [14, 25, 36]]
c.backward()

print(f"a.grad:\n{a.grad}")
print(f"b.grad: {b.grad}")
```

Luaran

```
a.grad:
[[1. 1. 1.]
 [1. 1. 1.]]
b.grad: [2. 2. 2.]
```

Sekarang gradien `b` sudah dijumlahkan sepanjang dimensi yang di-*broadcast* dengan benar .

Contoh: Broadcasting skalar

```
a = Tensor([[1.0, 2.0], [3.0, 4.0]], requires_grad=True)  # (2, 2)
b = Tensor(10.0, requires_grad=True)  # skalar

c = add(a, b)  # [[11, 12], [13, 14]]
c.backward()

print(f"a.grad:\n{a.grad}")
print(f"b.grad: {b.grad}")
```

Luaran

```
a.grad:
[[1. 1.]
 [1. 1.]]
b.grad: 4.0
```

Skalar `b` digunakan 4 kali (untuk setiap elemen matriks 2×2) sehingga gradiennya adalah 4.

3.3. Implementasi Beberapa Operasi Biner Per Elemen Lainnya

Untuk selanjutnya, asumsikan $y = g(f(a, b))$, di mana f akan diimplementasikan sebagai operasi biner yang dibahas di sub-bab berikut ini dan g merupakan fungsi yang bergantung pada f (yaitu, kemungkinan simpul operasi selanjutnya setelah f).

3.3.1. Pengurangan

Pengurangan adalah penjumlahan dengan operand kanan ternegasi: $f(a, b) = a - b = a + (-b)$. Implementasinya mirip dengan operasi penjumlahan di mana turunan-turunannya adalah

$$\frac{\partial y}{\partial a} = \frac{\partial (a-b)}{\partial a} \cdot \frac{\partial y}{\partial c} = \frac{\partial y}{\partial c} \tag{3.1}$$

dan

$$\frac{\partial y}{\partial b} = \frac{\partial (a-b)}{\partial b} \cdot \frac{\partial y}{\partial c} = -\frac{\partial y}{\partial c} \tag{3.2}$$

Bedanya ada pada operator negasi untuk `b_grad`.

```
def sub(a: Tensor, b: Tensor) -> Tensor:
    def backward_fn(upstream_grad: NDArray):
        a_grad = _unbroadcast_grad(
            upstream_grad, a.shape
        ) if a.requires_grad else None
        b_grad = -_unbroadcast_grad(
            upstream_grad, b.shape
        ) if b.requires_grad else None
        return [a_grad, b_grad]
```

```
    result = Tensor(a.data - b.data, requires_grad=a.requires_grad or b.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [a, b]
    return result
```

3.3.2. Perkalian

Untuk $f(a, b) = a \cdot b$,

$$\begin{aligned} \frac{\partial y}{\partial a} &= \frac{\partial (a \cdot b)}{\partial a} \cdot \frac{\partial y}{\partial c} \\ &= b \cdot \frac{\partial y}{\partial c} \end{aligned} \tag{3.3}$$

dan

$$\begin{aligned} \frac{\partial y}{\partial b} &= \frac{\partial (a \cdot b)}{\partial b} \cdot \frac{\partial y}{\partial c} \\ &= a \cdot \frac{\partial y}{\partial c} \end{aligned} \tag{3.4}$$

Implementasi kode:

```
def mul(a: Tensor, b: Tensor) -> Tensor:
    def backward_fn(upstream_grad: NDArray):
        a_grad = (
            _unbroadcast_grad(b.data * upstream_grad, a.shape)
            if a.requires_grad
            else None
        )
        b_grad = (
            _unbroadcast_grad(a.data * upstream_grad, b.shape)
            if b.requires_grad
            else None
        )
        return [a_grad, b_grad]

    result = Tensor(a.data * b.data, requires_grad=a.requires_grad or b.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [a, b]
    return result
```

3.3.3. Pembagian

Untuk $f(a, b) = \frac{a}{b}$,

$$\begin{aligned}\frac{\partial y}{\partial a} &= \frac{\partial c}{\partial a} \cdot \frac{\partial y}{\partial c} \\ &= \left[\frac{\partial}{\partial a}\left(\frac{a}{b}\right)\right] \cdot \frac{\partial y}{\partial c} \\ &= \frac{1}{b} \cdot \frac{\partial y}{\partial c}\end{aligned} \tag{3.5}$$

dan

$$\begin{aligned}\frac{\partial y}{\partial b} &= \frac{\partial c}{\partial b} \cdot \frac{\partial y}{\partial c} \\ &= \left[\frac{\partial}{\partial b}\left(\frac{a}{b}\right)\right] \cdot \frac{\partial y}{\partial c} \\ &= a \cdot (-b^{-2}) \cdot \frac{\partial y}{\partial c} \\ &= -\frac{a}{b^2} \cdot \frac{\partial y}{\partial c}\end{aligned} \tag{3.6}$$

Implementasinya:

```
def div(a: Tensor, b: Tensor) -> Tensor:
    def backward_fn(upstream_grad: NDArray):
        a_grad = (
            _unbroadcast_grad(upstream_grad / b.data, a.shape)
            if a.requires_grad
            else None
        )
        b_grad = (
            _unbroadcast_grad(-upstream_grad * a.data / (b.data**2), b.shape)
            if b.requires_grad
            else None
        )
        return [a_grad, b_grad]

    result = Tensor(a.data / b.data, requires_grad=a.requires_grad or b.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [a, b]
    return result
```

3.4. Implementasi Beberapa Operasi Uner (*Unary Operation*) Biasa

Untuk subbab operasi uner biasa, asumsikan $y = g(f(x))$ dan $u = f(x)$, di mana $f(x)$ akan diimplementasikan sebagai operasi uner, yang dibahas di sub-bab berikut ini. Gradien *upstream* dinotasikan dengan $g'(f(x)) = \frac{dy}{du}$.

3.4.1. Negasi

Negasi adalah operasi paling sederhana: $f(x) = -x$, dengan turunan $\frac{du}{dx} = -1$. Maka

$$\begin{aligned}\frac{dy}{dx} &= \frac{dy}{du}\frac{du}{dx} \\ &= -\frac{dy}{du}\end{aligned} \tag{3.7}$$

```
def neg(x: Tensor) -> Tensor:
    x = _ensure_tensor(x)

    def backward_fn(upstream_grad: NDArray):
        x_grad = -upstream_grad if x.requires_grad else None
        return [x_grad]

    result = Tensor(-x.data, requires_grad=x.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [x]
    return result
```

3.4.2. Eksponensial

Fungsi eksponensial $f(x) = e^x$ lazim digunakan di beragam aspek *deep learning*, seperti fungsi sigmoid dan *softmax*. Fungsi ini memiliki sifat unik, yaitu turunannya sama dengan fungsi itu sendiri. Maka

$$\frac{dy}{dx} = \frac{dy}{du}e^x \tag{3.8}$$

dengan implementasi sebagai berikut.

```
def exp(x: Tensor) -> Tensor:
    x = _ensure_tensor(x)
    exp_data = np.exp(x.data)

    def backward_fn(upstream_grad: NDArray):
        x_grad = upstream_grad * exp_data if x.requires_grad else None
        return [x_grad]

    result = Tensor(exp_data, requires_grad=x.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [x]
    return result
```

3.4.3. Akar Kuadrat

Untuk $f(x) = \sqrt{x}$,

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{1}{2\sqrt{x}} \tag{3.9}$$

```
def sqrt(x: Tensor) -> Tensor:
    x = _ensure_tensor(x)
    sqrt_data = np.sqrt(x.data)
```

```
    def backward_fn(upstream_grad: NDArray):
        # d/dx sqrt(x) = 1 / (2 * sqrt(x))
        x_grad = upstream_grad * (0.5 / sqrt_data) if x.requires_grad else None
        return [x_grad]

    result = Tensor(sqrt_data, requires_grad=x.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [x]
    return result
```

3.4.4. Logaritma Natural

Untuk $f(x) = \ln(x)$,

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{1}{x} = \left(\frac{dy}{du}\right)/x \tag{3.10}$$

```
def log(x: Tensor) -> Tensor:
    x = _ensure_tensor(x)

    def backward_fn(upstream_grad: NDArray):
        x_grad = upstream_grad / x.data if x.requires_grad else None
        return [x_grad]

    result = Tensor(np.log(x.data), requires_grad=x.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [x]
    return result
```

3.5. Implementasi Beberapa Operasi Reduksi

Operasi reduksi mengubah tensor menjadi bentuk yang lebih kecil dengan mengagregasi nilai sepanjang dimensi tertentu. Tantangan utama dalam implementasi gradien untuk operasi reduksi adalah memahami bagaimana gradien "disebarkan kembali" dari bentuk tereduksi ke bentuk asli.

Pada prinsipnya, jika laluan maju mengagregasi banyak nilai menjadi satu, maka laluan mundur harus mendistribusikan gradien dari *satu nilai* ke *banyak nilai* yang berkontribusi.

3.5.1. Penjumlahan Total (*Summation*)

Operasi penjumlahan total menjumlahkan elemen tensor sepanjang dimensi tertentu. Mari kita mulai dengan memahami mekanisme operasi ini sebelum membahas propagasi gradiennya.

Untuk tensor $\mathbf{X} \in \mathbb{R}^{m \times n}$ dengan elemen x_{ij}, kita dapat melakukan penjumlahan sepanjang dimensi berbeda. Untuk tensor rankin 2 (matriks), kita bisa melakukan penjumlahan sepanjang kolom (mereduksi dimensi kedua):

$$\mathbf{y} = \sum_{j=1}^{n} \mathbf{X}_{:,j} = \begin{pmatrix} \sum_{j=1}^{n} x_{1j} \\ \sum_{j=1}^{n} x_{2j} \\ \vdots \\ \sum_{j=1}^{n} x_{mj} \end{pmatrix} \in \mathbb{R}^m \tag{3.11}$$

Perhatikan bahwa setiap elemen y_i merupakan penjumlahan dari semua elemen di baris ke-i:

$$y_i = x_{i1} + x_{i2} + \cdots + x_{in} = \sum_{j=1}^{n} x_{ij} \tag{3.12}$$

Misalkan kita memiliki *loss* skalar ℓ yang bergantung pada $\mathbf{y} = \text{sum}(\mathbf{X}, \text{dim} = 1)$. Kita ingin menghitung $\frac{\partial \ell}{\partial x_{ij}}$ untuk setiap elemen di $\mathbf{X}$. Pertama, mari kita analisis kontribusi setiap elemen. Elemen x_{ij} hanya berkontribusi pada satu elemen output, yaitu y_i:

$$y_i = x_{i1} + x_{i2} + \cdots + x_{ij} + \cdots + x_{in} \tag{3.13}$$

Dengan menggunakan aturan rantai:

$$\frac{\partial \ell}{\partial x_{ij}} = \frac{\partial \ell}{\partial y_i} \cdot \frac{\partial y_i}{\partial x_{ij}} \tag{3.14}$$

Karena $y_i = \sum_{k=1}^{n} x_{ik}$, maka:

$$\frac{\partial y_i}{\partial x_{ij}} = \frac{\partial}{\partial x_{ij}} \sum_{k=1}^{n} x_{ik} = 1 \tag{3.15}$$

Sehingga:

$$\frac{\partial \ell}{\partial x_{ij}} = \frac{\partial \ell}{\partial y_i} \cdot 1 = \frac{\partial \ell}{\partial y_i} \tag{3.16}$$

Ini menunjukkan bahwa setiap elemen di baris ke-i menerima gradien yang sama, yaitu $\frac{\partial \ell}{\partial y_i}$. Untuk seluruh matriks $\mathbf{X}$, gradien dapat ditulis sebagai:

$$\frac{\partial \ell}{\partial \mathbf{X}} = \begin{pmatrix} \frac{\partial \ell}{\partial x_{11}} & \frac{\partial \ell}{\partial x_{12}} & \cdots & \frac{\partial \ell}{\partial x_{1n}} \\ \frac{\partial \ell}{\partial x_{21}} & \frac{\partial \ell}{\partial x_{22}} & \cdots & \frac{\partial \ell}{\partial x_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial x_{m1}} & \frac{\partial \ell}{\partial x_{m2}} & \cdots & \frac{\partial \ell}{\partial x_{mn}} \end{pmatrix} \tag{3.17}$$

Dari derivasi sebelumnya, kita tahu bahwa $\frac{\partial \ell}{\partial x_{ij}} = \frac{\partial \ell}{\partial y_i}$ untuk semua j. Dengan mensubstitusi hasil ini:

$$\frac{\partial \ell}{\partial \mathbf{X}} = \begin{pmatrix} \frac{\partial \ell}{\partial y_1} & \frac{\partial \ell}{\partial y_1} & \cdots & \frac{\partial \ell}{\partial y_1} \\ \frac{\partial \ell}{\partial y_2} & \frac{\partial \ell}{\partial y_2} & \cdots & \frac{\partial \ell}{\partial y_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial y_m} & \frac{\partial \ell}{\partial y_m} & \cdots & \frac{\partial \ell}{\partial y_m} \end{pmatrix} \tag{3.18}$$

Contoh: Ilustrasi propagasi gradien

Misalkan $\mathbf{X} \in \mathbb{R}^{2\times 3}$ adalah masukan dan $\mathbf{y} \in \mathbb{R}^2$ adalah hasil penjumlahan total dari $\mathbf{X}$ pada dimensi baris. Pada laluan maju dengan

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix}, \tag{3.19}$$

maka

$$\mathbf{y} = \begin{pmatrix} x_{11} + x_{12} + x_{13} \\ x_{21} + x_{22} + x_{23} \end{pmatrix}. \tag{3.20}$$

Pada laluan mundur, misalkan gradien *upstream* $\frac{\partial \ell}{\partial \mathbf{y}} = \begin{pmatrix} g_1 \\ g_2 \end{pmatrix}$. Karena dampak *broadcasting*, gradien g_1 "disebarkan" ke semua elemen baris pertama, dan g_2 ke semua elemen baris kedua.

$$\frac{\partial \ell}{\partial \mathbf{X}} = \begin{pmatrix} g_1 & g_1 & g_1 \\ g_2 & g_2 & g_2 \end{pmatrix} \tag{3.21}$$

Ketika `keepdims=False` (bawaan kebanyakan pustaka), dimensi yang direduksi akan hilang dari luaran. Misalnya, masukan $\mathbf{X} \in \mathbb{R}^{m\times n}$ akan menjadi luaran $\mathbf{y} \in \mathbb{R}^m$ (bukan $\mathbb{R}^{m\times 1}$). Saat propagasi balik, kita harus merekonstruksi dimensi yang hilang sebelum melakukan *broadcasting*.

```
def tensor_sum(
    x: Tensor, dim: int | tuple[int, ...] | None = None, keepdims: bool = False
):
    # Normalkan argumen dimensi
    if dim is None:
        dim = tuple(range(x.data.ndim))
    elif isinstance(dim, int):
        dim = (dim % x.data.ndim,)
    else:
        dim = tuple(d % x.data.ndim for d in dim)

    def backward_fn(upstream_grad: NDArray):
        reduced_axes = dim if not keepdims else ()
        if reduced_axes:
            shape = list(upstream_grad.shape)
            for axis in sorted(reduced_axes):
                shape.insert(axis, 1)
            upstream_grad = upstream_grad.reshape(shape)
        x_grad = np.broadcast_to(upstream_grad, x.shape) if x.requires_grad else None
        return [x_grad]

    result = Tensor(
        x.data.sum(axis=dim, keepdims=keepdims), requires_grad=x.requires_grad
    )
    result.backward_fn = backward_fn
```

```
    result.inputs = [x]
    return result
```

3.5.2. Rataan (*Mean*)

Rataan adalah penjumlahan total dibagi jumlah elemen: $\text{mean}(x) = \frac{1}{N}\sum_{i=1}^{N} x_i$ Turunannya adalah

$$\begin{aligned}\frac{\partial}{\partial x_i}\,\text{mean}(x) &= \frac{\partial\left(\frac{1}{N}\sum_{i=1}^{N} x_i\right)}{\partial x_i} \\ &= \frac{1}{N}\end{aligned} \tag{3.22}$$

Ini berarti gradien untuk mean adalah gradien sum dibagi dengan jumlah elemen yang dicari rataannya. Perhatikan bahwa N bergantung pada dimensi reduksi yang ditentukan:

```
X = [[1, 2, 3],
     [4, 5, 6]]  # Shape: (2, 3)

mean(X, dim=0) = [2.5, 3.5, 4.5]  # N=2 untuk setiap kolom
mean(X, dim=1) = [2, 5]           # N=3 untuk setiap baris
mean(X, dim=None) = 3.5           # N=6 untuk semua elemen
```

Untuk `mean(X, dim=1)` dengan upstream_grad = [1, 1]:

- Setiap elemen di baris pertama mendapat gradien 1/3
- Setiap elemen di baris kedua mendapat gradien 1/3

Implementasinya mirip dengan sum, tetapi dengan faktor skala $\frac{1}{N}$.

```
def tensor_mean(
    x: Tensor, dim: int | tuple[int, ...] | None = None, keepdims: bool = False
):
    if dim is None:
        dim = tuple(range(x.data.ndim))
    elif isinstance(dim, int):
        # Normalize single negative dimension
        dim = (dim % x.data.ndim,)
    else:
        # Normalize tuple of dimensions (including negative ones)
        dim = tuple(d % x.data.ndim for d in dim)

    result = Tensor(
        x.data.mean(axis=dim, keepdims=keepdims), requires_grad=x.requires_grad
    )

    def backward_fn(upstream_grad: NDArray):
        # Calculate how many elements were averaged
        n_elements = 1
        for axis in dim:
            n_elements *= x.shape[axis]
```

```
        reduced_axes = dim if not keepdims else ()
        if reduced_axes:
            shape = list(upstream_grad.shape)
            for axis in sorted(reduced_axes):
                shape.insert(axis, 1)
            upstream_grad = upstream_grad.reshape(shape)

        upstream_grad = upstream_grad / n_elements
        x_grad = np.broadcast_to(upstream_grad, x.shape) if x.requires_grad else None
        return [x_grad]

    result.backward_fn = backward_fn
    result.inputs = [x]
    return result
```

3.5.3. Maksimum (*Max*) dan Minimum (*Min*)

Max:

```
def tensor_max(
    x: Tensor, dim: int | tuple[int, ...] | None = None, keepdims: bool = False
):
    if dim is None:
        dim = tuple(range(x.data.ndim))
    elif isinstance(dim, int):
        # Normalize single negative dimension
        dim = (dim % x.data.ndim,)
    else:
        # Normalize tuple of dimensions (including negative ones)
        dim = tuple(d % x.data.ndim for d in dim)

    result_data = x.data.max(axis=dim, keepdims=keepdims)
    result = Tensor(result_data, requires_grad=x.requires_grad)

    def backward_fn(upstream_grad: NDArray):
        # Create mask where max values are
        expanded_result = result_data
        reduced_axes = dim if not keepdims else ()
        if reduced_axes:
            shape = list(expanded_result.shape)
            for axis in sorted(reduced_axes):
                shape.insert(axis, 1)
            expanded_result = expanded_result.reshape(shape)

        # Mask is 1 where x equals the max value
        mask = (x.data == expanded_result).astype(x.data.dtype)

        # Count how many times each max appears (for tie-breaking)
        normalizer = mask.sum(axis=dim, keepdims=True)
        mask = mask / normalizer
```

```
        # Expand upstream gradient
        if reduced_axes:
            shape = list(upstream_grad.shape)
            for axis in sorted(reduced_axes):
                shape.insert(axis, 1)
            upstream_grad = upstream_grad.reshape(shape)

        x_grad = mask * upstream_grad if x.requires_grad else None
        return [x_grad]

    result.backward_fn = backward_fn
    result.inputs = [x]
    return result
```

Min:

```
def tensor_min(
    x: Tensor, dim: int | tuple[int, ...] | None = None, keepdims: bool = False
):
    if dim is None:
        dim = tuple(range(x.data.ndim))
    elif isinstance(dim, int):
        # Normalize single negative dimension
        dim = (dim % x.data.ndim,)
    else:
        # Normalize tuple of dimensions (including negative ones)
        dim = tuple(d % x.data.ndim for d in dim)

    result_data = x.data.min(axis=dim, keepdims=keepdims)
    result = Tensor(result_data, requires_grad=x.requires_grad)

    def backward_fn(upstream_grad: NDArray):
        # Create mask where min values are
        expanded_result = result_data
        reduced_axes = dim if not keepdims else ()
        if reduced_axes:
            shape = list(expanded_result.shape)
            for axis in sorted(reduced_axes):
                shape.insert(axis, 1)
            expanded_result = expanded_result.reshape(shape)

        # Mask is 1 where x equals the min value
        mask = (x.data == expanded_result).astype(x.data.dtype)

        # Count how many times each min appears (for tie-breaking)
        normalizer = mask.sum(axis=dim, keepdims=True)
        mask = mask / normalizer

        # Expand upstream gradient
```

```
        if reduced_axes:
            shape = list(upstream_grad.shape)
            for axis in sorted(reduced_axes):
                shape.insert(axis, 1)
            upstream_grad = upstream_grad.reshape(shape)

        x_grad = mask * upstream_grad if x.requires_grad else None
        return [x_grad]

    result.backward_fn = backward_fn
    result.inputs = [x]
    return result
```

3.6. Implementasi Beberapa Fungsi Aktivasi

3.6.1. ReLU

```
def relu(x: Tensor) -> Tensor:
    def backward_fn(upstream_grad: NDArray):
        x_grad = upstream_grad * (x.data > 0) if x.requires_grad else None
        return [x_grad]

    result = Tensor(np.maximum(x.data, 0), requires_grad=x.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [x]
    return result
```

3.6.2. Sigmoid

```
def sigmoid(x: Tensor) -> Tensor:
    # sigmoid(x) = 1 / (1 + exp(-x))
    sig_data = 1 / (1 + np.exp(-x.data))

    def backward_fn(upstream_grad: NDArray):
        # d/dx sigmoid(x) = sigmoid(x) * (1 - sigmoid(x))
        x_grad = upstream_grad * sig_data * (1 - sig_data) if x.requires_grad else
None
        return [x_grad]

    result = Tensor(sig_data, requires_grad=x.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [x]
    return result
```

3.6.3. Tanh

```
def tanh(x: Tensor) -> Tensor:
    tanh_x = np.tanh(x.data)
    def backward_fn(upstream_grad: NDArray) -> list[NDArray | None]:
        # d/dx tanh(x) = 1 - tanh(x)^2
```

```
        x_grad = upstream_grad * (1 - tanh_x**2) if x.requires_grad else None
        return [x_grad]

    result = Tensor(tanh_x, requires_grad=x.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [x]

    return result
```

3.7. Softmax

Softmax biasanya digunakan di lapisan terakhir model *neural network* untuk memperoleh luaran ternormalisasi yang jumlah totalnya 1, sehingga bisa diinterpretasikan sebagai distribusi probabilitas. Beberapa arsitektur modern seperti transformer juga menggunakannya untuk mekanisme *attention*. Misal kita memiliki rangkap (*tuple*) $\mathbf{z} = (z_1, ..., z_K) \in \mathbb{R}^K$, misalnya, *logits* kotor atau luaran lapisan terakhir suatu model *neural network*. Softmax dihitung dengan

$$\text{softmax}(\mathbf{z}) = \frac{e^{z_i}}{\sum_{i=1}^{K} e^{z_i}} \tag{3.23}$$

Dengan operasi-operasi yang sudah kita implementasikan sebelumnya, mudah saja kita implementasikan operasi softmax.

```
def softmax(x, dim=-1):
    numerator = exp(x)
    return numerator / tensor_sum(numerator, dim, keepdims=True)
```

Namun implementasi di atas memiliki kelemahan, yaitu instabilitas numerik. Saat nilai masukan (misal, dari logit) memiliki nilai yang sangat besar atau sangat kecil. Saat masukan bernilai sangat besar, mendekati tak hingga, akan terjadi *overflow*, di mana fungsi eksponensial akan mengembalikan `np.inf`.

Solusinya, kita bisa mengimplementasikan versi stabil dari *softmax*.

```
def softmax(x: Tensor, dim: int = -1) -> Tensor:
    # Normalisasi dimensi
    dim = dim % x.data.ndim

    # Kurangkan nilai terbesar maksimal untuk stabilitas numerik
    x_max = tensor_max(x, dim=dim, keepdims=True)
    x_shifted = x - x_max

    # Softmax
    exp_x = exp(x_shifted)
    sum_exp = tensor_sum(exp_x, dim=dim, keepdims=True)
    return exp_x / sum_exp
```

Pendalaman

Walau stabil, softmax masih menyisakan kelemahan, yaitu *overconfidence*. Nilai masukan yang tinggi cenderung mendominasi distribusi probabilitas, walaupun kenyataannya model bersifat *uncertain*. Untuk menanggulanginya, kita bisa menggunakan pengembangannya, yaitu softmax dalam bentuk logaritmik, atau log-softmax.

```
def log_softmax(x: Tensor, dim: int = -1) -> Tensor:
  dim = dim % x.data.ndim

  # stabilitas numerik
  x_max = tensor_max(x, dim=dim, keepdims=True)
  x_shifted = x - x_max

  # log(softmax(x)) = x - max(x) - log(sum(exp(x - max(x))))
  exp_shifted = exp(x_shifted)
  sum_exp = tensor_sum(exp_shifted, dim=dim, keepdims=True)
  log_sum_exp = log(sum_exp)

  return x_shifted - log_sum_exp
```

3.8. Implementasi Operasi Perkalian Matriks

Perkalian matriks (tensor ranking 2) mungkin adalah salah satu operasi paling penting dalam *deep learning*. Laluan maju operasi ini mudah saja dan sudah didukung oleh NumPy. Ekspresi penghitungan gradien pada laluan mundurnya juga sederhana, hanya melibatkan sedikit perkalian matriks dan transposisi yang melibatkan gradien *upstream*:

```
def matmul(a: Tensor, b: Tensor) -> Tensor:
    def backward_fn(upstream_grad: NDArray):
        a_grad = upstream_grad @ b.data.T if a.requires_grad else None
        b_grad = a.data.T @ upstream_grad if b.requires_grad else None
        return [a_grad, b_grad]

    result = Tensor(a.data @ b.data, requires_grad=a.requires_grad or b.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [a, b]
    return result
```

Kendati mudah dalam kode, perlu kita cermati pemahaman konseptual di balik implementasinya.

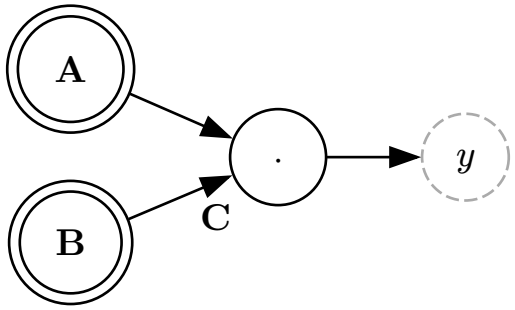
Ingat kembali dasar perkalian matriks. Untuk matriks $\mathbf{A} \in \mathbb{R}^{m \times n}$ dan $\mathbf{B} \in \mathbb{R}^{n \times p}$, hasil perkalian $\mathbf{C} = \mathbf{AB}$ adalah matriks berukuran $m \times p$. Setiap elemen $c_{ij} \in \mathbf{C}$ dihitung sebagai

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{3.24}$$

Dengan kata lain, elemen di posisi (i, j) dari hasil adalah darab bintik (*dot product*) dari baris ke-i matriks $\mathbf{A}$ dengan kolom ke-j matriks $\mathbf{B}$.

Misal, kita punya nilai skalar *loss* y yang bergantung pada $\mathbf{C} = \mathbf{AB}$ (diilustrasikan sebagai graf komputasi oleh Gambar 7) dan kita ingin menghitung $\frac{\partial y}{\partial a_{ik}}$ dan $\frac{\partial y}{\partial b_{kj}}$. Pertama, perhatikan bahwa setiap elemen a_{ik} berkontribusi ke semua elemen di baris ke-i dari $\mathbf{C}$:

$$\begin{pmatrix} c_{i1} \\ c_{i2} \\ \vdots \\ c_{ip} \end{pmatrix} = \begin{pmatrix} a_{i1}b_{11} + \cdots + a_{ik}b_{k1} + \cdots + a_{in}b_{n1} \\ a_{i1}b_{12} + \cdots + a_{ik}b_{k2} + \cdots + a_{in}b_{n2} \\ \vdots \\ a_{i1}b_{1p} + \cdots + a_{ik}b_{kp} + \cdots + a_{in}b_{np} \end{pmatrix} \tag{3.25}$$



Gambar 7: Representasi graf perkalian matriks $\mathbf{C} = \mathbf{AB}$ yang diikuti dengan suatu fungsi sebarang dengan luaran skalar y (yang diperoleh dari, misalnya, dari operasi penjumlahan total atau rataan)

Dengan menggunakan aturan rantai, gradien y terhadap elemen-elemen baris ke-i dari $\mathbf{A}$ diperoleh dengan

$$\begin{aligned}
\begin{pmatrix} \frac{\partial y}{\partial a_{i1}} \\ \frac{\partial y}{\partial a_{i2}} \\ \vdots \\ \frac{\partial y}{\partial a_{in}} \end{pmatrix} &= \begin{pmatrix} \sum_j \frac{\partial y}{\partial c_{ij}} \frac{\partial c_{ij}}{\partial a_{i1}} \\ \sum_j \frac{\partial y}{\partial c_{ij}} \frac{\partial c_{ij}}{\partial a_{i2}} \\ \vdots \\ \sum_j \frac{\partial y}{\partial c_{ij}} \frac{\partial c_{ij}}{\partial a_{in}} \end{pmatrix} \\
&= \begin{pmatrix} \sum_j \frac{\partial y}{\partial c_{ij}} b_{1j} \\ \sum_j \frac{\partial y}{\partial c_{ij}} b_{2j} \\ \vdots \\ \sum_j \frac{\partial y}{\partial c_{ij}} b_{nj} \end{pmatrix} && \text{(Ingat bahwa } \tfrac{\partial c_{ij}}{\partial a_{ik}} = b_{kj} \text{ karena } c_{ij} = \textstyle\sum_k a_{ik}b_{kj}\text{)} \\
&= \begin{pmatrix} \frac{\partial y}{\partial c_{i1}} \\ \frac{\partial y}{\partial c_{i2}} \\ \cdots \\ \frac{\partial y}{\partial c_{ip}} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}^{\mathsf{T}}, && \text{(Hasil dari langkah sebelumnya adalah darab bintik dan dapat ditulis sebagai perkalian matriks)} \\
\begin{pmatrix} \frac{\partial y}{\partial a_{i1}} \\ \frac{\partial y}{\partial a_{i2}} \\ \vdots \\ \frac{\partial y}{\partial a_{in}} \end{pmatrix} &= \begin{pmatrix} \frac{\partial y}{\partial c_{i1}} \\ \frac{\partial y}{\partial c_{i2}} \\ \vdots \\ \frac{\partial y}{\partial c_{ip}} \end{pmatrix} \mathbf{B}^{\mathsf{T}}.
\end{aligned} \tag{3.26}$$

Ekspresi di atas juga bisa diterjemahkan sebagai kontribusi elemen-elemen baris ke-i dari $\mathbf{A}$. Untuk kemudahan, dengan sedikit "*notation abuse*", kita tuliskan $\frac{\partial y}{\partial \mathbf{A}}$ sebagai matriks yang elemen (i, j)-nya adalah $\frac{\partial y}{\partial a_{ij}}$. Maka, gradien keseluruhan dari semua baris pada $\mathbf{A}$ jika disusun dalam bentuk matriks menjadi

$$\begin{pmatrix} \frac{\partial l}{\partial a_{11}} & \frac{\partial l}{\partial a_{12}} & \cdots & \frac{\partial l}{\partial a_{1n}} \\ \frac{\partial l}{\partial a_{21}} & \frac{\partial l}{\partial a_{22}} & \cdots & \frac{\partial l}{\partial a_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial a_{m1}} & \frac{\partial l}{\partial a_{m2}} & \cdots & \frac{\partial l}{\partial a_{mn}} \end{pmatrix} = \begin{pmatrix} \frac{\partial y}{\partial c_{11}} & \frac{\partial y}{\partial c_{12}} & \cdots & \frac{\partial y}{\partial c_{1p}} \\ \frac{\partial y}{\partial c_{21}} & \frac{\partial y}{\partial c_{22}} & \cdots & \frac{\partial y}{\partial c_{2p}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial c_{m1}} & \frac{\partial y}{\partial c_{m2}} & \cdots & \frac{\partial y}{\partial c_{mp}} \end{pmatrix} \mathbf{B}^\mathsf{T} \tag{3.27}$$

$$\frac{\partial y}{\partial \mathbf{A}} = \frac{\partial y}{\partial \mathbf{C}} \mathbf{B}^\mathsf{T}$$

Dengan logika serupa untuk matriks $\mathbf{B}$:

$$\frac{\partial y}{\partial \mathbf{B}} = \mathbf{A}^\mathsf{T} \frac{\partial y}{\partial \mathbf{C}} \tag{3.28}$$

Pada kode operasi `matmul`, Atribut `a.grad` dan `b.grad` berkorespondensi dengan $\frac{\partial y}{\partial \mathbf{A}}$ dan $\frac{\partial y}{\partial \mathbf{B}}$, sedangkan `upstream_grad` berkorespondensi dengan $\frac{\partial y}{\partial \mathbf{C}}$. Sekarang makna dari baris kode pada fungsi laluan balik `matmul` mulai jelas:

```
def matmul(a: Tensor, b: Tensor) -> Tensor:
    ...

    def backward_fn(upstream_grad: NDArray):
        a_grad = upstream_grad @ b.data.T if a.requires_grad else None
        b_grad = a.data.T @ upstream_grad if b.requires_grad else None
        return [a_grad, b_grad]
    ...
```

3.8.1. Kasus Khusus: *Batch Matrix Multiplication*

Operator `@` pada NumPy sudah mendukung *batch matrix multiplication*. Dua dimensi terakhir diperlakukan sebagai matriks, dimensi sisanya sebagai *batch*.

Operand Kiri	Operand Kanan	Hasil
(m, n)	(n, p)	(m, p)
(b, m, n)	(b, n, p)	(b, m, p)
(b, m, n)	(n, p)	(b, m, p)

Tabel 4: Perilaku operator `@` untuk berbagai dimensi tensor

Untuk tensor $\mathbf{A} \in \mathbb{R}^{b \times m \times n}$ dan $\mathbf{B} \in \mathbb{R}^{b \times n \times p}$, operasi `@` menghasilkan $\mathbf{C} \in \mathbb{R}^{b \times m \times p}$ dengan $\mathbf{C}_i = \mathbf{A}_i \mathbf{B}_i$ untuk setiap $i \in \{1, ..., b\}$. Laluan maju implementasi kita otomatis mendukung ini. Namun, ada masalah pada laluan mundur: metode `.T` membalik **seluruh** dimensi, bukan hanya dimensi matriks. Perbaiki dengan `swapaxes(-2, -1)`:

```
>>> x = np.zeros((2, 3, 4))
>>> x.T.shape  # keliru untuk dimensi lebih dari 2
(4, 3, 2)
>>> x.swapaxes(-2, -1).shape  # lebih tepat, hanya membalik dua dimensi terakhir
(2, 4, 3)
```

Fungsi `backward_fn` kita perbarui sebagai berikut:

```
def matmul(a: Tensor, b: Tensor) -> Tensor:
    def backward_fn(upstream_grad: NDArray):
        b_T = np.swapaxes(b.data, -2, -1)
        a_T = np.swapaxes(a.data, -2, -1)

        a_grad = upstream_grad @ b_T if a.requires_grad else None
        b_grad = a_T @ upstream_grad if b.requires_grad else None
        return [a_grad, b_grad]

    result = Tensor(a.data @ b.data, requires_grad=a.requires_grad or b.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [a, b]
    return result
```

Sekarang implementasi kita mendukung tensor dengan dimensi sembarang.

3.9. *Indexing* dan *Slicing*

Kita juga perlu mendukung operasi *indexing* dan *slicing* pada tensor. Operasi ini penting dalam deep *deep learning*, mulai dari mengambil *batch* data dengan indeks tertentu, memilih fitur spesifik, hingga implementasi operasi yang lebih kompleks seperti *embedding lookup* dan pemilihan sampel pada langkah waktu tertentu untuk arsitektur rekuren.

Yang menarik dari *indexing* adalah bagaimana kita menghitung gradiennya. Berbeda dengan operasi matematika seperti perkalian atau penjumlahan yang memiliki notasi turunan formal ($\frac{\partial y}{\partial x}$), operasi *indexing* tidak memiliki notasi matematis standar untuk turunannya. Ini karena *indexing* bukan fungsi matematis dalam pengertian "tradisional". Ia hanyalah operasi "pemilihan" atau "penyalinan" elemen.

Meski begitu, aturan gradiennya sederhana: ketika kita mengambil sebagian elemen dari tensor, gradien yang mengalir balik harus diletakkan tepat pada posisi yang sama dengan elemen yang diambil, sementara posisi lainnya mendapat gradien nol. Ini sesuai dengan intuisi bahwa hanya elemen yang dipilih yang berkontribusi pada tensor luaran, sehingga hanya mereka yang menerima gradien.

```
def index_select(x: Tensor, indices) -> Tensor:
    def backward_fn(upstream_grad: NDArray) -> list[NDArray | None]:
        if not x.requires_grad:
            return [None]

        # Persiapkan penampung gradien dengan ukuran yang sama
        # dengan masukan
        x_grad = np.zeros_like(x.data)
```

```
        # Letakkan upstream_grad tepat pada lokasi terindeks
        x_grad[indices] = upstream_grad

        return [x_grad]

    indexed_data = x.data[indices]
    result = Tensor(indexed_data, requires_grad=x.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [x]

    return result
```

Agar bisa menggunakannya dengan nyaman seperti *indexing* normal (`x[i]`, `x[:, 1:100]`, dst.), tambahkan metode *dunder* pada kelas `Tensor`.

```
class Tensor:

    ...

    def __getitem__(self, indices):
        return index_select(self, indices)

    ...
```

3.10. Penggabungan dan Pemecahan Tensor

3.10.1. *Stack*

```
def stack(tensors: list[Tensor], dim: int = 0) -> Tensor:
    """Tumpuk daftar tensor sepanjang dimensi tertentu"""
    # Tangani dimensi negatif
    data_list = [t.data for t in tensors]
    ndim = data_list[0].ndim
    if dim < 0:
        dim = ndim + 1 + dim

    requires_grad = any(t.requires_grad for t in tensors)

    def backward_fn(upstream_grad: NDArray) -> list[NDArray | None]:
        grads = []
        for i, t in enumerate(tensors):
            if t.requires_grad:
                # Extract the gradient for this tensor
                indices: list[slice | int] = [slice(None)] * upstream_grad.ndim
                indices[dim] = i
                grads.append(upstream_grad[tuple(indices)])
            else:
                grads.append(None)
        return grads
```

```
    stacked_data = np.stack(data_list, axis=dim)
    result = Tensor(stacked_data, requires_grad=requires_grad)
    result.backward_fn = backward_fn
    result.inputs = tensors

    return result
```

3.10.2. *Split*

```
def split(x: Tensor, num_splits: int, dim: int = -1) -> list[Tensor]:
    """
    Potong tensor menjadi bagian-bagian berukuran sama sepanjang dimensi tertentu

    Tidak ada backward_fn karena operasi ini turunan dari index_select yang sudah
    mengimplementasikan backward_fn
    """
    # Tangani dimensi negatif
    if dim < 0:
        dim = x.data.ndim + dim

    # Hitung ukuran tiap potongan
    dim_size = x.shape[dim]
    if dim_size % num_splits != 0:
        raise ValueError(f"Dimension size {dim_size} not divisible by {num_splits}")

    split_size = dim_size // num_splits

    splits = []
    for i in range(num_splits):
        indices = [slice(None)] * x.data.ndim
        indices[dim] = slice(i * split_size, (i + 1) * split_size)
        splits.append(x[tuple(indices)])

    return splits
```

3.11. Manipulasi Bentuk

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere.

3.11.1. Transposisi

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere.

```
TODO
```

3.11.2. *Reshape*

```
def reshape(x: Tensor, *shape: int) -> Tensor:
    x = _ensure_tensor(x)
    reshaped_data = x.data.reshape(shape)

    def backward_fn(upstream_grad: NDArray):
        # Kembalikan upstream_grad ke bentuk awal x
        x_grad = upstream_grad.reshape(x.data.shape) if x.requires_grad else None
        return [x_grad]

    result = Tensor(reshaped_data, requires_grad=x.requires_grad)
    result.backward_fn = backward_fn
    result.inputs = [x]
    return result
```

Modifikasi kelas `Tensor`

```
class Tensor:
    ...

    def reshape(self, *shape: int):
        return reshape(self, *shape)
```

3.12. Pemanasan: Regresi Linier

Sebelum membangun neural network yang kompleks, mari kita uji implementasi dengan model paling sederhana: regresi linier. Regresi linier memodelkan hubungan antara masukan $\mathbf{x} \in \mathbb{R}^d$ (tiap sampel) dan luaran $y \in \mathbb{R}$ sebagai:

$$y = \mathbf{x}^\mathsf{T}\mathbf{w} + b \tag{3.29}$$

di mana $\mathbf{w} \in \mathbb{R}^d$ adalah vektor bobot (*weight*) dan $b \in \mathbb{R}$ adalah bias. Untuk dataset dengan n sampel, kita peroleh $\mathbf{X} \in \mathbb{R}^{n \times d}$ dan semua prediksi direpresentasikan dengan notasi matriks sebagai berikut:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b \tag{3.30}$$

Kita gunakan Mean Squared Error (MSE) sebagai fungsi loss:

$$L_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.31}$$

Mari implementasikan MSE dengan operasi tensor kita:

```
def mean_squared_error(y: Tensor, y_pred: Tensor) -> Tensor:
    err = y - y_pred
    return (err * err).mean()
```

3.12.1. Dataset

Untuk latihan ini, kita menggunakan dataset *California Housing* yang disediakan oleh pustaka `scikit-learn`. Fungsi `fetch_california_housing` akan mengembalikan larin NumPy. Kita standarisasi matriks fitur dan pastikan `y` berukuran $(n, 1)$. Setelah itu konversi `x` dan `y` dalam `Tensor`.

```
...
from sklearn.datasets import fetch_california_housing

x, y = fetch_california_housing(return_X_y=True)

x = (x - x.mean(axis=0)) / x.std(axis=0)  # standarisasi
y = y.reshape(-1, 1)

# Konversikan ke Tensor
x = Tensor(x)
y = Tensor(y)

# Variable pembantu, jumlah feature
n_feat = x.shape[1]
```

3.12.2. Gradient Descent

Untuk mengoptimasi parameter, kita gunakan aturan pembaruan *gradient descent*:

$$\begin{aligned} w &:= w - \alpha \frac{\partial}{\partial w} L_{\text{MSE}} \\ b &:= b - \alpha \frac{\partial}{\partial b} L_{\text{MSE}} \end{aligned} \tag{3.32}$$

di mana α adalah *learning rate*. Perhatikan bagaimana dengan mudahnya kita bisa memperoleh gradien seluruh parameter dengan memanggil `loss.backward()`.

```
# Parameter dan bias.
# Perhatikan bahwa w dan b melacak gradien karena kita akan memperbarui mereka.
w = Tensor(np.random.randn(n_feat, 1), requires_grad=True)
b = Tensor(0.0, requires_grad=True)

learning_rate = 0.1
max_iter = 1000
for i in range(max_iter):
    y_pred = x @ w + b
    loss = mean_squared_error(y, y_pred)
    loss.backward()

    # Pembaruan parameter sesuai gradient descent
    w.data = w.data - learning_rate * w.grad  # type:ignore
    b.data = b.data - learning_rate * b.grad  # type:ignore

    # Nol-kan kembali gradien
    w.zero_grad()
```

```
    b.zero_grad()

    if i % 10 == 0:
        print(f"loss at {i}: {float(loss.data):.2f}")

# Prediksi akhir
y_pred = x @ w + b
```

3.12.3. Kode Selengkapnya

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import fetch_california_housing
from tqdm import tqdm

# Penulis menamai pustaka ini dengan "Lantern". Tidak sebesar "Torch",
# namun cukup sebagai penerang ;)
from lantern import Tensor


def mean_squared_error(y: Tensor, y_pred: Tensor) -> Tensor:
    err = y - y_pred
    return (err * err).mean()


# Kita tidak memecah dataset menjadi bagian train-test sekarang
# karena kita hanya ingin melihat seberapa baik model kita melakukan fitting
x, y = fetch_california_housing(return_X_y=True)

x = (x - x.mean(axis=0)) / x.std(axis=0)  # standarisasi
y = y.reshape(-1, 1)

x = Tensor(x)
y = Tensor(y)

n_feat = x.shape[1]

# Parameter dan bias.
# Perhatikan bahwa w dan b melacak gradien karena kita akan memperbarui mereka.
w = Tensor(np.random.randn(n_feat, 1), requires_grad=True)
b = Tensor(0.0, requires_grad=True)

learning_rate = 0.1
max_iter = 1000

for i in range(max_iter):
    y_pred = x @ w + b
    loss = mean_squared_error(y, y_pred)
    loss.backward()
```

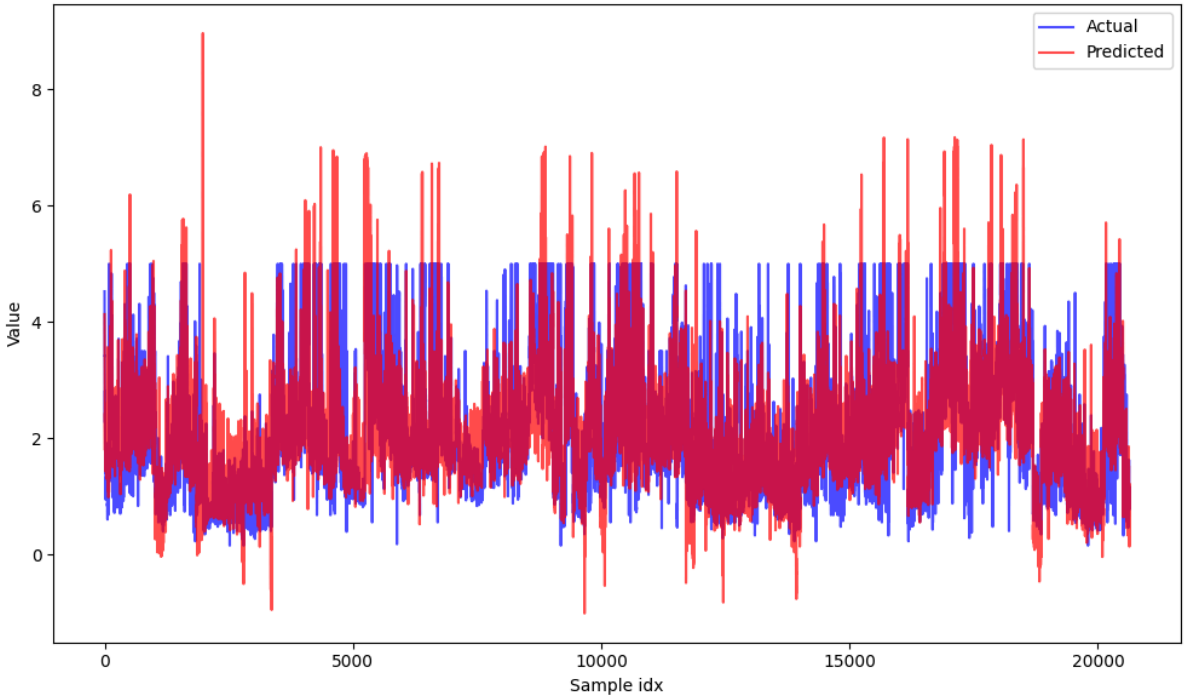
```
    # Parameter update
    w.data = w.data - learning_rate * w.grad
    b.data = b.data - learning_rate * b.grad

    # Reset parameter gradients
    w.zero_grad()
    b.zero_grad()

    if i % 10 == 0:
        print(f"loss: {float(loss.data):.2f}")

# Prediksi akhir
y_pred = x @ w + b

# Plot prediction vs actual
plt.figure(figsize=(10, 6))
plt.plot(y.data, "b-", label="Actual", alpha=0.7)
plt.plot(y_pred.data, "r-", label="Predicted", alpha=0.7)
plt.xlabel("Sample idx")
plt.ylabel("Value")
plt.legend()
plt.show()
```

Luaran

```
loss at 0: 11.42
loss at 10: 0.75
loss at 20: 0.65
loss at 30: 0.62
...
loss at 970: 0.52
loss at 980: 0.52
loss at 990: 0.52
```

Jika implementasi kita benar, error seharusnya mendekati nol (dalam batas presisi numerik).

Dari visualisasi pada Gambar 8, kita dapat melihat bahwa model berhasil menangkap tren umum harga rumah. Prediksi dapat mengikuti pola nilai aktual dengan cukup baik. Dengan fondasi ini, kita siap membangun model yang lebih kompleks seperti *multilayer perceptron*.



Gambar 8: Perbandingan nilai aktual (biru) dan prediksi (merah) pada dataset *California Housing*. Model regresi linier berhasil menangkap tren umum meskipun tidak sempurna untuk data yang kompleks.

Bab 4. Antarmuka Pemrograman Aplikasi (API) *Deep learning*

Pada bab sebelumnya, kita telah membangun sistem automatic differentiation yang fungsional. Namun, menulis model yang kompleks dengan API level rendah akan menjadi sangat merepotkan. Melacak puluhan parameter secara manual bukan hal mudah. Belum lagi mengatur mode training/evaluation, atau membangun arsitektur berlapis. Di sinilah kita memerlukan abstraksi yang lebih tinggi.

PyTorch memperkenalkan konsep modul (`Module`) sebagai unsur pokok untuk membangun *neural network*. Desain ini elegan karena memungkinkan kita untuk membuat komposisi hierarkis yang alami untuk *deep learning*, di mana suatu modul dapat berisi beberapa modul lain. Tentu kita juga akan membuat versi kita sendiri.

4.1. Kelas *Module*

Kelas `Module` adalah jantung dari API tingkat tinggi kita. Setiap komponen *neural network* dari lapisan sederhana hingga model kompleks akan mewarisi kelas ini.

4.1.1. Anatomi Modul

Kita mulai sketsa kelas `Module` dengan beberapa atribut.

```
class Module:
    def __init__(self):
        self._parameters: dict[str, Tensor] = {}
        self._modules: dict[str, Module] = {}
        self.training = True
```

Dictionary `_parameters` menyimpan semua tensor yang perlu dioptimasi (misal, bobot dan bias dari lapisan linier). Dictionary `_modules` dapat menyimpan sub-modul yang memungkinkan kita untuk membangun arsitektur berlapis. Penanda `training` menentukan perilaku module saat laluan maju. Penanda ini penting untuk beberapa lapisan seperti *dropout* dan *batch normalization* yang berperilaku berbeda saat mode lati dan mode inferensi.

4.1.2. Forward Pass Abstrak

```
class Module:
  ...
  def forward(self, *args, **kwargs) -> Tensor:
      raise NotImplementedError()
```

Method `forward` sengaja dibuat abstrak. Setiap module konkret harus mendefinisikan logika forward pass-nya sendiri. Ini memaksa desain yang eksplisit—tidak ada perilaku default yang mungkin menyesatkan. Signature yang fleksibel dengan `*args` dan `**kwargs` mengakomodasi berbagai jenis module, dari lapisan sederhana yang menerima satu tensor hingga model kompleks dengan multiple inputs.

4.1.3. Mode Training dan Evaluasi

Metode `train` mengatur penanda `training` secara rekursif.

```
def train(self, mode: bool = True):
    self.training = mode
    for module in self._modules.values():
        module.train(mode)
```

Ketika kita memanggil `model.train()` pada model tingkat teratas, semua sub-modul juga akan masuk mode latih. Perhatikan bahwa kita tidak perlu melakukan apapun terhadap `parameters` di sini. Parameters tidak memiliki "mode" karena mereka hanya data. Yang berubah adalah bagaimana perilaku modul menggunakan parameter-parameter tersebut.

4.1.4. Iterasi Parameter

```
def parameters(self):
    for param in self._parameters.values():
        yield param
    for module in self._modules.values():
        yield from module.parameters()
```

Metode dengan kembalian generator ini mengumpulkan semua parameter secara rekursif. Pertama, *yield* semua parameter lokal, kemudian secara rekursif *yield* juga parameter dari sub-modul. Penggunaan generator membuat metode ini efisien dari sisi konsumsi memori karena kita tidak perlu membuat larik besar berisi semua parameter sekaligus.

Penggunakan generator alih-alih `list` tentu memiliki konsekuensi (*trade-off*). Generator lebih efisien untuk model besar, tapi kita tidak bisa mengindeks hasilnya atau mengetahui jumlah parameter tanpa iterasi penuh.

Catatan

PyTorch juga memiliki metode `parameters()` yang mengembalikan *generator* karena keuntungan efisiensi memori lebih penting untuk model modern yang bisa memiliki miliaran parameter. Metode `parameters()` ini biasanya akan diakses oleh pengoptimal (*optimizer*). Pembaca yang akrab dengan pustaka PyTorch mungkin mengenali pola ini:

```
...
optimizer = Adam(model.parameters(), lr=0.001)
...
```

4.1.5. Metode *Magic* untuk Ergonomi

Kita akan menggunakan pola yang mirip dengan PyTorch dari segi ergonomi dengan mengimplementasikan beberapa metode *magic* seperti `__call__` dan `__setattr__`.

```
def __call__(self, *args, **kwargs) -> Tensor:
    return self.forward(*args, **kwargs)
```

Dengan mendefinisikan `__call__`, kita bisa menggunakan module seperti fungsi: `output = model(input)` alih-alih `output = model.forward(input)`. Ini membuat kode lebih natural dan konsisten dengan konvensi Python di mana callable objects umum digunakan.

Penimpaan metode *magic* `__setattr__` berguna untuk registrasi modul atau parameter secara otomatis.

```
def __setattr__(self, name, value):
    if isinstance(value, Tensor):
        self._parameters[name] = value
    elif isinstance(value, Module):
        self._modules[name] = value
    super().__setattr__(name, value)
```

Setiap kali kita melakukan penetapan tensor atau module sebagai atribut objek, mereka otomatis terdaftar di dictionary internal yang sesuai. Kita akan dimungkinkan untuk menulis sintaks secara natural seperti ini dengan efek samping tambahan:

```
self.weight = Tensor(np.random.randn(10, 5))  # Otomatis terdaftar sebagai parameter
self.linear = Linear(5, 3)                    # Otomatis terdaftar sebagai sub-module
```

Tanpa ini, kita harus secara manual melakukan penetapan setiap parameter dan modul, dan ini menambah peluang terjadinya galat.

4.1.6. Implementasi Lengkap

```
class Module:
    def __init__(self):
        self._parameters: dict[str, Tensor] = {}
        self._modules: dict[str, Module] = {}
        self.training = True

    def forward(self, *args, **kwargs) -> Tensor:
        raise NotImplementedError()

    def train(self, mode: bool = True):
        self.training = mode
        for module in self._modules.values():
            module.train(mode)

    def parameters(self):
        for param in self._parameters.values():
            yield param
        for module in self._modules.values():
            yield from module.parameters()

    def __call__(self, *args, **kwargs) -> Tensor:
        return self.forward(*args, **kwargs)

    def __setattr__(self, name, value):
```

```
        if isinstance(value, Tensor):
            self._parameters[name] = value
        elif isinstance(value, Module):
            self._modules[name] = value
        super().__setattr__(name, value)
```

4.2. Beberapa Jenis Lapisan (*Layer*) Dasar

4.2.1. Lapisan Linier

Lapisan linier bertugas melakukan operasi $y = \mathbf{W}\mathbf{x} + \mathbf{b}$. Pada lapisan ini, terjadi proses penjumlahan terbobot (*weighted sum*) pada semua fitur masukan dengan matriks $\mathbf{W}$. Vektor bias $\mathbf{b}$ memberikan nilai *offset* untuk menunjang ekspresivitas model. Berikut implementasinya:

```
class Linear(Module):
    def __init__(self, in_features: int, out_features: int, bias: bool = True):
        super().__init__()

        self.in_features = in_features
        self.out_features = out_features
        self.bias = bias

        self.w = Tensor(
            np.random.randn(self.in_features, self.out_features), requires_grad=True
        )
        self.b = (
            Tensor(np.zeros((self.out_features,)), requires_grad=True)
            if self.bias
            else None
        )

    def forward(self, x: Tensor) -> Tensor:
        out = x @ self.w
        # Tambah bias jika diperlukan
        if self.b:
            out = out + self.b
        return out
```

Penulis mengikuti API PyTorch yang mengizinkan bias bersifat opsional. Perilaku bawaan inisialisasi bias adalah menetapkan seluruh elemennya dengan nilai 0.

Pendalaman

Teknik lanjutan seperti Xavier [4] dan He [1] digunakan pustaka *deep learning* modern untuk menginisialisasi matriks bobot `self.w` alih-alih menggunakan sebarang nilai acak. Misalkan suatu model *deep learning* memiliki L lapisan dengan $l = 1, \ldots, L$, matriks bobot untuk tiap lapisan l yang dinotasikan dengan $\mathbf{W}^{(l)}$, dan jumlah neuron pada lapisan ke-l yang dinotasikan dengan n_l. Inisialisasi Xavier menentukan tiap entri $w \in \mathbf{W}^{(l)}$ untuk mengikuti distribusi seragam

$$w \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{l-1}+n_l}}, \sqrt{\frac{6}{n_{l-1}+n_l}},\right). \tag{4.1}$$

Varian normalnya mengikuti distribusi

$$w \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{l-1}+n_l}}\right) \tag{4.2}$$

Dalam terminologi Xavier, variabel n_{l-1} disebut *fan-in* dan n_l disebut *fan-out*. Sayangnya, performa teknik Xavier disinyalir tidak begitu bagus dengan fungsi aktivasi ReLU seperti dilaporkan oleh Kumar *et al.* [5]. He *et al.* dalam [1] kemudian menawarkan inisialisasi berdasarkan distribusi seragam

$$w \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{l-1}}}, \sqrt{\frac{6}{n_l}}\right) \tag{4.3}$$

dan varian normalnya berdasarkan

$$w \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{l-1}}}\right). \tag{4.4}$$

Berikut ini implementasi keempatnya:

```
import numpy as np

def xavier_uniform(n_in, n_out, gain=1.0):
    """Xavier/Glorot uniform initialization"""
    bound = gain * np.sqrt(6.0 / (n_in + n_out))
    return np.random.uniform(-bound, bound, (n_in, n_out))

def xavier_normal(n_in, n_out, gain=1.0):
    """Xavier/Glorot normal initialization"""
    std = gain * np.sqrt(2.0 / (n_in + n_out))
    return np.random.normal(0, std, (n_in, n_out))

def he_uniform(n_in, n_out, gain=1.0):
    """He uniform initialization"""
    bound = gain * np.sqrt(6.0 / n_in)
    return np.random.uniform(-bound, bound, (n_in, n_out))

def he_normal(n_in, n_out, gain=1.0):
    """He normal initialization"""
    std = gain * np.sqrt(2.0 / n_in)
    return np.random.normal(0, std, (n_in, n_out))
```

Kita bisa menggunakan salah satu fungsi di atas untuk menggantikan `np.random.rand(self.in_features, self.out_features)` sesuai kebutuhan.

4.2.2. Fungsi Aktivasi Sebagai Modul

Pola umum yang bisa ditemui ... fungsi aplikasi digunakan sebagai modul.

Hal ini juga dapat ditemui pada pustaka PyTorch.

```
class ReLU(Module):
    def forward(self, x: Tensor) -> Tensor:
        return relu(x)


class Sigmoid(Module):
    def forward(self, x: Tensor) -> Tensor:
        return sigmoid(x)


class Tanh(Module):
    def forward(self, x: Tensor) -> Tensor:
        return tanh(x)
```

4.2.3. Lapisan *Drop-Out*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

4.2.4. Lapisan *Embedding*

Lapisan *embedding* mengubah indeks diskrit (bilangan bulat) menjadi vektor kontinu berdimensi tetap. Bayangkan kita punya kamus dengan 10.000 kata, dan setiap kata diwakili oleh indeks unik (0 sampai 9.999). Lapisan embedding memetakan setiap indeks ini ke vektor berukuran tetap, misalnya 128 dimensi.

Secara matematis, lapisan *embedding* dapat dipandang sebagai matriks $\mathbf{E} \in \mathbb{R}^{v \times d}$, dengan V adalah ukuran kosa-kata dan d adalah dimensi *embedding*. Ketika kita memberikan indeks i, lapisan ini mengembalikan baris ke-i dari matriks $\mathbf{E}$. Operasi ini sebenarnya sama dengan melakukan perkalian matriks antara vektor *one-hot* dengan matriks *embedding*. Namun, implementasi langsung dengan *indexing* jauh lebih efisien karena kita tidak perlu membuat vektor *one-hot* yang sangat jarang (*sparse*).

```
class Embedding(Module):
    def __init__(self, num_embeddings: int, embedding_dim: int):
        super().__init__()
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim
```

```
        # Inisialisasi matriks embedding dengan distribusi normal
        self.weight = Tensor(
            np.random.normal(0, 0.02, (num_embeddings, embedding_dim)),
            requires_grad=True
        )

    def forward(self, input_ids: Tensor) -> Tensor:
        # Gunakan indexing yang sudah kita implementasikan
        return self.weight[input_ids.data.astype(int)]
```

Gradien untuk lapisan *embedding* dihitung melalui operasi indexing yang telah kita implementasikan sebelumnya. Ketika backpropagation terjadi, hanya baris-baris yang digunakan (yang terindeks) yang akan menerima gradien, sementara baris lainnya tetap tidak berubah.

Lapisan embedding menjadi fondasi model-model bahasa modern seperti BERT dan GPT. Pada model-model tersebut, setiap *token* (kata atau subkata) dipetakan ke vektor *embedding* yang kemudian diproses oleh lapisan-lapisan *transformer* (yang akan kita implementasikan juga di bagian selanjutnya).

Contoh: Contoh penggunaan lapisan *embedding*

Mari kita lihat bagaimana lapisan embedding bekerja dalam praktiknya:

```
# Kosa-kata sederhana: indeks untuk kata-kata
vocab = {"<pad>": 0, "halo": 1, "dunia": 2, "deep": 3, "learning": 4}
vocab_size = len(vocab)
embedding_dim = 4

# Buat lapisan embedding
embed_layer = Embedding(vocab_size, embedding_dim)

# Kalimat sebagai deretan indeks
# "halo halo learning" -> [1, 1, 4]
sentence_indices = Tensor(np.array([1, 1, 4]))

# Laluan maju
embeddings = embed_layer(sentence_indices)
embeddings.backward()

print("Embedding weight")
print(embed_layer.weight)
print("Embedding weight.grad")
print(embed_layer.weight.grad)
print("Embedding result")
print(embeddings)
print(f"Shape: {embeddings.shape}")  # (3, 8) - 3 kata, masing-masing 4
dimensi
```

Luaran

```
Embedding weight
array([[-0.01665979,  0.01320985,  0.00940399,  0.02108722],
       [-0.01865394,  0.0582143 , -0.01788336,  0.00022806],
       [-0.01375527, -0.00489541, -0.00520801, -0.00751396],
       [ 0.02269423, -0.00837957,  0.02449486, -0.01828196],
       [-0.00226322,  0.01171692, -0.01726989, -0.00774343]])
Embedding weight.grad
[[0. 0. 0. 0.]
 [2. 2. 2. 2.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [1. 1. 1. 1.]]
Embedding result
array([[-0.01865394,  0.0582143 , -0.01788336,  0.00022806],
       [-0.01865394,  0.0582143 , -0.01788336,  0.00022806],
       [-0.00226322,  0.01171692, -0.01726989, -0.00774343]])
Shape: (3, 4)
```

Perhatikan bahwa baris pertama dan kedua hasil *embedding* sama karena kita mengulang indeks kata "halo" (1) dua kali. Karena itu pula baris kedua pada `weight.grad` bernilai dua, karena data pada baris ini berkontribusi sebanyak dua kali bagi luaran.

4.3. Lapisan Sekuensial

Lapisan sekuensial menerima modul-modul pada konstruktor dengan mempertahankan urutannya. Saat `forward()` dipanggil, lapisan ini akan menerima masukan yang kompatibel dengan modul pertama pada sekuens. Modul pertama akan memanggil `forward()`, kemudian luarannya menjadi masukan modul selanjutnya, kemudian luarannya jadi masukan modul selanjutnya, dan seterusnya, secara berurutan.

```
class Sequential(Module):
    def __init__(self, *modules):
        super().__init__()
        for i, module in enumerate(modules):
            # Use string indices so they're stored in _modules
            setattr(self, str(i), module)

    def forward(self, x: Tensor) -> Tensor:
        for module in self._modules.values():
            x = module(x)
        return x
```

4.4. Fungsi *Loss* (*Loss Function*)

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

4.4.1. MSE

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

4.4.2. Cross-Entropy

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

```
class CrossEntropyLoss(Module):
    def __init__(self, dim: int = -1):
        super().__init__()
        self.dim = dim

    def forward(self, logits: Tensor, targets: Tensor) -> Tensor:
        """Cross-entropy loss for one-hot targets and logits"""
        # Apply log-softmax for numerical stability
        log_probs = log_softmax(logits, dim=self.dim)

        # Coss-entropy: -sum(targets * log_probs)
        loss_per_sample = -(targets * log_probs).sum(dim=self.dim)

        # Mean loss
        return tensor_mean(loss_per_sample)
```

4.5. Pengoptimal (*Optimizer*)

Pengoptimal adalah komponen dalam ekosistem *deep learning* yang bertanggung jawab mengatur pembaruan parameter model berdasarkan gradien yang dihitung selama propagasi balik. Kelas ini memberi abstraksi terhadap pembaruan parameter secara manual. Pengoptimal mengimplementasikan berbagai varian dari ide dasar *gradient descent*. Namun, *gradient descent* murni memiliki beberapa kelemahan fundamental yang membuatnya kurang praktis untuk model modern. Mari kita bahas evolusi dari pengoptimal sederhana hingga yang lebih mutakhir.

4.5.1. Kelas Basis

Semua pengoptimal dalam pustaka kita akan mewarisi kelas basis yang mendefinisikan antarmuka umum.

```
from typing import Iterable
import numpy as np

from .tensor import Tensor
```

```
class Optimizer:
    def __init__(self, parameters: Iterable[Tensor]):
        self.parameters = list(parameters)

    def step(self):
        """Update parameters based on their gradients"""
        pass

    def zero_grad(self):
        """Zero out all parameter gradients"""
        for param in self.parameters:
            param.zero_grad()
```

Metode `step()` adalah bagian utama dari pengoptimal, di mana pembaruan parameter terjadi. Setiap pengoptimal konkret akan mengimplementasikan logika pembaruannya sendiri di metode ini. Metode `zero_grad()` juga penting karena sistem autodiff kita mengakumulasi gradien. Tanpa me-nol-kan gradien setelah setiap iterasi, gradien dari iterasi sebelumnya akan terus terakumulasi, menghasilkan pembaruan yang keliru.

4.5.2. *Stochastic Gradient descent* (SGD)

SGD adalah pengoptimal paling fundamental dalam deep learning. Meskipun sederhana, SGD masih banyak digunakan karena sifatnya yang dapat diprediksi dan terbukti efektif untuk berbagai masalah. Aturan pembaruannya mengikuti konsep gradient descent yang pernah kita bahas pada Persamaan (3.32).

```
class SGD(Optimizer):
    def __init__(self, parameters: Iterable[Tensor], lr: float = 0.01):
        super().__init__(parameters)
        self.lr = lr

    def step(self):
        for param in self.parameters:
            if param.grad is not None:
                param.data -= self.lr * param.grad
```

Kesederhanaan SGD adalah kekuatan sekaligus kelemahannya. Tidak ada mekanisme adaptif, tidak ada momentum, hanya langkah konstan ke arah berlawanan gradien. Ini membuat SGD mudah dipahami dan diawakutu, tapi juga membuatnya lambat konvergen dan sensitif terhadap pemilihan *learning rate*.

Catatan

Implementasi SGD pada pustaka PyTorch sudah mengadopsi momentum dan Nesterov

4.5.3. RMSprop

RMSprop (*root mean square propagation*) adalah pengoptimal pertama yang memperkenalkan kon-

sep *learning rate* adaptif. Diperkenalkan pada *Lecture 6* Coursera oleh Hinton[5], RMSprop mengatasi masalah learning rate yang terlalu besar atau kecil dengan menyesuaikannya berdasarkan magnitudo riwayat gradien. Ide utamanya adalah parameter dengan gradien besar secara konsisten seharusnya mendapat *learning rate* efektif yang lebih kecil, sementara parameter dengan gradien kecil seharusnya mendapat *learning rate* yang lebih besar. Aturan pembaruan RMSprop adalah sebagai berikut:

$$
\begin{aligned}
v_t &= \beta v_{t-1} + (1-\beta) g_t^2 \\
\theta_t &= \theta_{t-1} - \frac{\alpha}{\sqrt{v_t + \varepsilon}} g_t
\end{aligned}
\tag{4.5}
$$

Variabel v_t di langkah waktu t berkorespondensi dengan masing-masing skalar parameter model, $\theta \in \Theta$.

```python
class RMSprop(Optimizer):
    def __init__(
        self,
        parameters: Iterable[Tensor],
        lr: float = 0.01,
        alpha: float = 0.99,
        eps: float = 1e-8
    ):
        super().__init__(parameters)
        self.lr = lr
        self.alpha = alpha
        self.eps = eps

        # Running average of squared gradients
        self.v = [np.zeros_like(param.data) for param in self.parameters]

    def step(self):
        for i, param in enumerate(self.parameters):
            if param.grad is None:
                continue

            grad = param.grad

            # Pembaruan rata-rata bergerak dari kuadrat gradien
            self.v[i] = self.alpha * self.v[i] + (1 - self.alpha) * (grad ** 2)

            # Pembaruan parameter dengan learning rate adaptif
            param.data -= self.lr * grad / (np.sqrt(self.v[i]) + self.eps)
```

4.5.4. Adam

Adam (*Adaptive Moment Estimation*) [6] menggabungkan ide momentum dan *learning rate* adaptif per parameter. Adam cenderung menjadi pilihan *default* untuk banyak praktisi karena performanya yang konsisten untuk beragam masalah. Adam menyimpan dua momen statistik untuk setiap parameter, yaitu, 1) momen pertama m sebagai estimasi rata-rata bergerak dari gradien; dan 2) momen

[5] https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

kedua v: Estimasi rata-rata bergerak dari kuadrat gradien. Aturan pembaruan Adam adalah sebagai berikut:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_b) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_t &= \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}
\end{aligned}
\tag{4.6}
$$

dengan *hyperparameter* β_1 dan β_2 untuk mengontrol seberapa cepat estimasi momen "melupakan" nilai-nilai lama, dan *hyperparameter* ε untuk mencegah pembagian dengan nol. Variabel m_t dan v_t di langkah waktu t berkorespondensi dengan masing-masing skalar parameter model, $\theta_t \in \Theta$. Berikut implementasinya dengan Python.

```
class Adam(Optimizer):
    def __init__(
        self,
        parameters: Iterable[Tensor],
        lr: float = 0.001,
        beta1: float = 0.9,
        beta2: float = 0.999,
        eps: float = 1e-8,
    ):
        super().__init__(parameters)
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.eps = eps
        self.t = 0  # time step, akan bertambah tiap `.step()` dieksekusi

        # Inisialisasi penampung momentum
        self.m = [np.zeros_like(param.data) for param in self.parameters]
        self.v = [np.zeros_like(param.data) for param in self.parameters]

    def step(self):
        self.t += 1

        for i, param in enumerate(self.parameters):
            if param.grad is None:
                continue

            grad = param.grad

            # Perbarui estimasi bias momen pertama
            self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * grad
```

```
            # Perbarui estimasi bias momen kedua
            self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * (grad**2)

            # Koreksi bias untuk momen pertama
            m_hat = self.m[i] / (1 - self.beta1**self.t)
            # Koreksi bias untuk momen kedua
            v_hat = self.v[i] / (1 - self.beta2**self.t)

            # Perbarui parameter
            param.data -= self.lr * m_hat / (np.sqrt(v_hat) + self.eps)
```

Momen pertama membantu melewati area datar dan mengurangi osilasi. Setiap parameter memiliki *learning rate* efektif sendiri berdasarkan riwayat gradiennya. Selain itu, koreksi bias memastikan estimasi akurat di awal pelatihan.

Catatan

Dalam praktiknya, tidak ada pengoptimal yang "terbaik" untuk semua kasus. Pilihan pengoptimal bergantung pada:

- Jenis arsitektur (CNN vs RNN vs Transformer)
- Ukuran dataset
- Anggaran komputasi
- Kebutuhan generalisasi vs kecepatan konvergensi

Prinsip dasar: mulai dengan Adam untuk purwarupa cepat, lalu coba SGD+Momentum untuk generalisasi lebih baik jika ada waktu untuk melakukan *tuning*. Biasanya Adam saja sudah cukup.

4.5.5. AdamW

AdamW adalah varian Adam yang memisahkan efek *weight decay* dari gradien. Hal ini ternyata menghasilkan regularisasi yang lebih efektif dan berdampak baik pada generalisasi model. Model populer seperti Transformer dilatih dengan pengoptimal ini.

```
class AdamW(Optimizer):
    def __init__(
        self,
        parameters: Iterable[Tensor],
        lr: float = 0.001,
        betas: tuple[float, float] = (0.9, 0.999),
        eps: float = 1e-8,
        weight_decay: float = 0.01
    ):
        super().__init__(parameters)
        self.lr = lr
        self.beta1, self.beta2 = betas
        self.eps = eps
        self.weight_decay = weight_decay
```

```
        self.t = 0  # time step, akan bertambah tiap `.step()` dieksekusi

        self.m = [np.zeros_like(param.data) for param in self.parameters]
        self.v = [np.zeros_like(param.data) for param in self.parameters]

    def step(self):
        self.t += 1

        for i, param in enumerate(self.parameters):
            if param.grad is None:
                continue

            grad = param.grad

            self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * grad
            self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * (grad ** 2)

            # Koreksi bias
            m_hat = self.m[i] / (1 - self.beta1 ** self.t)
            v_hat = self.v[i] / (1 - self.beta2 ** self.t)

            # Pembaruan dengan weight decay terpisah (a.l., bukan di gradien)
            param.data -= self.lr * (
                m_hat / (np.sqrt(v_hat) + self.eps) +
                self.weight_decay * param.data
            )
```

4.6. *Multi-Layer Perceptron (MLP)* dengan API baru

Dengan semua komponen yang telah kita bangun, mari kita mengujinya dengan API tingkat tinggi kita dengan membangun dan melatih MLP untuk klasifikasi digit tulisan tangan.

4.6.1. Pewarisan kelas *Module*

Pendekatan pertama adalah membuat kelas MLP yang mewarisi Module. Ini memberikan kontrol penuh atas arsitektur dan alur data:

```
import lantern
from lantern.nn import CrossEntropyLoss, Linear, Module, ReLU
from sklearn.datasets import load_digits
from sklearn.preprocessing import OneHotEncoder

from lantern.optim import Adam
from lantern.tensor import Tensor

x, y = load_digits(return_X_y=True)
x = x / x.max()
y = y.reshape(-1, 1)  # type:ignore
y_one_hot = OneHotEncoder(sparse_output=False).fit_transform(y)

x = Tensor(x)
```

```
y_one_hot = Tensor(y_one_hot)


class MLP(Module):
    def __init__(self):
        super().__init__()

        self.l1 = Linear(64, 256)
        self.l2 = Linear(256, 256)
        self.l3 = Linear(256, 10)
        self.act1 = ReLU()
        self.act2 = ReLU()

    def forward(self, x: Tensor) -> Tensor:
        x = self.act1(self.l1(x))
        x = self.act2(self.l2(x))
        x = self.l3(x)
        return x


model = MLP()
loss_fn = CrossEntropyLoss()
optim = Adam(model.parameters())

for i in range(500):
    optim.zero_grad()

    logits = model(x)
    loss = loss_fn(logits, y_one_hot)
    loss.backward()
    print(loss)

    optim.step()
```

Pendekatan ini memberikan fleksibilitas maksimal bagi pengguna. Kita bisa menambahkan logika khusus di metode `forward`, seperti *dropout* kondisional atau koneksi lompatan (*skip connections*).

4.6.2. API Lapisan Sekuensial

Untuk arsitektur yang lebih sederhana dan linear, kita bisa menggunakan `Sequential`:

```
model = Sequential(
    Linear(64, 256),
    ReLU(),
    Linear(256, 256),
    ReLU(),
    Linear(256, 10),
)

loss_fn = CrossEntropyLoss()
optim = Adam(model.parameters())
```

```
for i in range(500):
    optim.zero_grad()

    logits = model(x)
    loss = loss_fn(logits, y_one_hot)
    loss.backward()
    print(loss)

    optim.step()
```

API Sequential lebih ringkas untuk arsitektur sederhana, tapi kurang fleksibel. Pilihan antara keduanya tergantung pada kompleksitas model yang dibutuhkan. keduanya juga bisa dikombinasikan satu sama lain. Misalnya, kita bisa membuat kelompok modul, membungkusnya dalam lapisan sekuensial, dan menggunakannya secara berulang dalam modul lain.

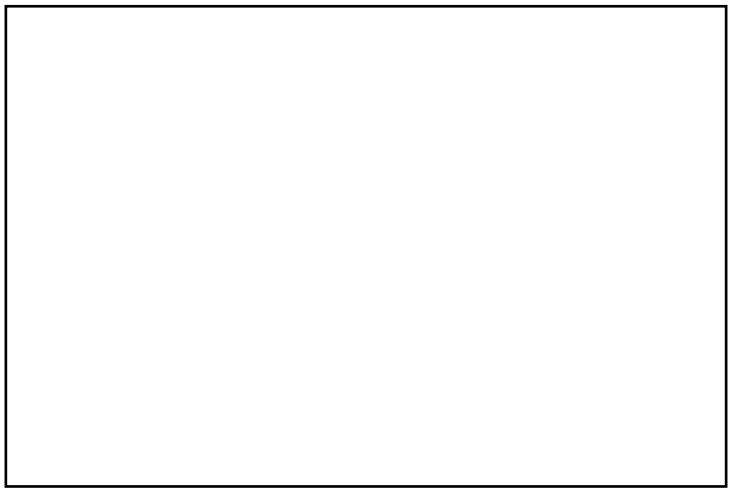
Bab 5. Lapisan Lanjutan

5.1. Lapisan Konvolusional Dua Dimensi

Lapisan konvolusional adalah komponen penting dalam pada *deep learning*, terutama untuk pemrosesan citra dan data yang memiliki struktur spasial. Berbeda dengan lapisan linier yang memperlakukan setiap piksel secara independen, lapisan konvolusional mempelajari hubungan spasial antar piksel dengan menggunakan filter (*kernel*) yang bergeser melintasi citra masukan.

5.1.1. Intuisi dan Motivasi

Dalam pengolahan citra tradisional, filter konvolusi telah lama digunakan untuk deteksi tepian, efek buram, penajaman, dan berbagai operasi pemrosesan citra lainnya. *Deep learning* mengadopsi konsep ini. Bedanya, parameter filter dipelajari secara otomatis melalui proses pelatihan, bukan dirancang manual.



Gambar 9: Contoh penggunaan *gaussian filter* untuk menghasilkan citra buram

Secara formal, operasi konvolusi untuk citra multikanal (*multi-channel image*) didefinisikan oleh

$$g_{c_{\text{out}},i,j} = \sum_{c_{\text{in}}=0}^{C_{\text{in}}-1} \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} x_{c_{\text{in}},i-u,j-v} \cdot w_{c_{\text{out}},c_{\text{in}},u,v} + b_{c_{\text{out}}} \tag{5.1}$$

dengan keterangan variabel-variabel berikut ini

Variabel	Keterangan
g	luaran konvolusi
x	citra masukan konvolusi
w	filter konvolusi
b	vektor bias

C_{in}	jumlah kanal citra masukan
C_{out}	jumlah kanal citra luaran
k_h	tinggi filter konvolusi
k_w	lebar filter konvolusi
i, j	posisi spasial pada luaran
$c_{\text{in}}, c_{\text{out}}$	indeks kanal
u, v	indeks posisi filter

5.1.2. Konvolusi Sebagai Perkalian Matriks

Meskipun konvolusi tampak sebagai operasi yang kompleks, pada implementasinya sering diubah menjadi perkalian matriks untuk efisiensi komputasi. Transformasi ini melibatkan tiga langkah utama:

1. Im2Col: Mengubah patches dari citra masukan menjadi kolom-kolom dalam matriks
2. Perkalian matriks: Melakukan perkalian matriks biasa
3. *Reshape*: Mengatur ulang hasil perkalian menjadi peta fitur berupa tensor ranking 4 (bukan 3 karena kita akan memproses dalam mode *batch*)

Pendekatan ini memungkinkan pemanfaatan pustaka aljabar linier yang sudah dioptimalkan seperti BLAS.

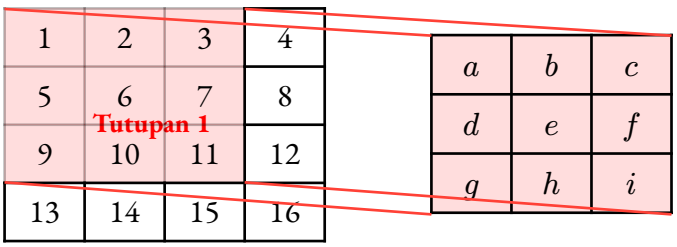
5.1.3. Im2Col

Algoritma Im2Col adalah teknik untuk mengubah operasi konvolusi menjadi perkalian matriks. Ide dasarnya adalah "menghamparkan" tutupan-tutupan (*patches*) dari citra masukan menjadi kolom atau baris dalam matriks.

Catatan

Secara historis, pada implementasi awal pustaka *deep learning* im2col akan menghamparkan tutupan memanjang sebagai kolom, dan karena itulah prosedur ini disebut im2col, *image patches to columns*. Beberapa pustaka *deep learning* dan implementasi pada buku ini akan menghamparkan tutupan-tutupan citra sebagai baris.

Untuk setiap posisi jendela geser pada citra masukan, lakukan ekstraksi tutupan berukuran filter dari citra. Ratakan tutupan menjadi vektor kolom. Kemudian susun semua kolom secara berdampingan Sebagai contoh, misalkan kita memiliki citra skala keabuan (1 kanal) 4×4 dan filter 3×3:



Gambar 1: Contoh konvolusi pada citra abu berukuran 4×4 dengan filter berukuran 3×3. Filter akan bergeser satu langkah ke kanan dan satu langkah ke bawah

Im2Col akan mengekstraksi 4 tutupan (2×2 posisi luaran) seperti ditunjukkan pada Gambar 2.

Tutupan 1

1	2	3
5	6	7
9	10	11

Tutupan 2

2	3	4
6	7	8
10	11	12

Tutupan 3

5	6	7
9	10	11
13	14	15

Tutupan 4

6	7	8
10	11	12
14	15	16

Gambar 2: Tutupan-tutupan berdasarkan oleh jendela filter

Setelah tutupan-tutupan citra dihamparkan menjadi tumpukan baris (matriks berukuran 4×9) dan filter dihamparkan menjadi kolom, bisa dilakukan perkalian matriks biasa, ditunjukkan pada Gambar 1.

1	2	3	5	6	7	9	10	11
2	3	4	6	7	8	10	11	12
5	6	7	9	10	11	13	14	15
6	7	8	10	11	12	14	15	16

$*$

a
b
c
d
e
f
g
h
i

Gambar 1: Perkalian matriks antara citra dan filter yang dihamparkan. Pada contoh ini, hanya ada satu kanal warna pada citra (derajat keabuan) dan hanya satu filter luaran (respons filter juga memiliki satu kanal)

Contoh di atas hanya untuk satu kanal citra masukan dan luaran. Jika kita memiliki masukan dan luaran dengan beberapa kanal, ukuran matriks tentu akan berbeda. Untuk masukan dengan, misalnya, 3 kanal (citra RGB) berukuran 4×4 dengan filter 3×3 untuk kanal luaran 64:

- Citra masukan berbentuk $1 \times 3 \times 4 \times 4$
- Filter berbentuk $64 \times 3 \times 3 \times 3$

Im2Col akan menghasilkan tumpukan tutupan berupa matriks dengan bentuk (1×2×2, 3×3×3) = (4, 27). Setiap baris berisi 27 elemen: 3 kanal × 3×3 kernel = 27 nilai untuk tiap tutupan. Akan ada 4 baris untuk 4 posisi luaran berbentuk 2×2 (seperti pada Gambar 2). Untuk luaran multikanal, filter dihamparkan dan disusun hingga berbentuk $(3 \times 3 \times 3, 64) = (27, 64)$.

Berikut implementasi im2col:

```
def im2col(
    input_data: NDArray,
    filter_h: int,
    filter_w: int,
    stride_h: int = 1,
    stride_w: int = 1,
    pad_h: int = 0,
    pad_w: int = 0,
```

```
) -> NDArray:
    N, C, H, W = input_data.shape

    # Hitung dimensi luaran
    out_h = (H + 2 * pad_h - filter_h) // stride_h + 1
    out_w = (W + 2 * pad_w - filter_w) // stride_w + 1

    # Persiapkan padding untuk masukan, namun pertahankan dimensi batch dan
    # dimensi kanal
    img = np.pad(
        input_data, [(0, 0), (0, 0), (pad_h, pad_h), (pad_w, pad_w)], "constant"
    )

    # Inisialisasi larik multidimensi untuk tutupan-tutupan dari citra:
    col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))

    # Ekstraksi tutupan-tutupan dengan jendela geser
    for y in range(filter_h):
        y_max = y + stride_h * out_h
        for x in range(filter_w):
            x_max = x + stride_w * out_w
            col[:, :, y, x, :, :] = img[:, :, y:y_max:stride_h, x:x_max:stride_w]

    # Transpose: (N, C, filter_h, filter_w, out_h, out_w) menjadi
    #            (N, out_h, out_w, C, filter_h, filter_w)
    # Reshape: Gabungkan posisi spasial (N×out_h×out_w) dan ratakan dimensi filter
    #          (C×filter_h×filter_w)
    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N * out_h * out_w, -1)
    return col
```

5.1.4. Col2Im

Operasi col2im merupakan transformasi kebalikan dari im2col yang berperan dalam propagasi balik lapisan konvolusi. Sementara im2col mengekstrak dan menyusun tutupan-tutupan citra menjadi matriks untuk efisiensi komputasi, col2im melakukan rekonstruksi sebaliknya: mengembalikan representasi matriks kolom ke format tensor citra multidimensi aslinya.

Pada konteks konvolusi beberapa piksel muncul di beberapa tutupan karena adanya *overlap* antar jendela filter yang bergeser. Fenomena *overlap* ini memerlukan strategi khusus dalam rekonstruksi, yaitu dengan mengakumulasi kontribusi dari setiap tutupan yang mengandung piksel tersebut. Proses akumulasi ini menggambarkan bagaimana gradien dari berbagai posisi luaran konvolusi berkontribusi terhadap gradien di posisi masukan tertentu.

```
def col2im(
    col: NDArray,
    input_shape: tuple[int, ...],
    filter_h: int,
    filter_w: int,
    stride_h: int = 1,
    stride_w: int = 1,
```

```
    pad_h: int = 0,
    pad_w: int = 0,
) -> NDArray:
    N, C, H, W = input_shape
    out_h = (H + 2 * pad_h - filter_h) // stride_h + 1
    out_w = (W + 2 * pad_w - filter_w) // stride_w + 1

    col = col.reshape(N, out_h, out_w, C, filter_h, filter_w).transpose(
        0, 3, 4, 5, 1, 2
    )

    img = np.zeros((N, C, H + 2 * pad_h + stride_h - 1, W + 2 * pad_w + stride_w - 1))
    for y in range(filter_h):
        y_max = y + stride_h * out_h
        for x in range(filter_w):
            x_max = x + stride_w * out_w
            img[:, :, y:y_max:stride_h, x:x_max:stride_w] += col[:, :, y, x, :, :]

    return img[:, :, pad_h : H + pad_h, pad_w : W + pad_w]
```

Verifikasi kebenaran implementasi col2im dapat dilakukan dengan menguji sifat inversnya terhadap im2col. Untuk kasus stride=1 dan padding yang mempertahankan dimensi, aplikasi berurutan im2col diikuti col2im seharusnya menghasilkan tensor identik dengan masukan awal. Namun, perlu dicatat bahwa untuk konfigurasi stride lebih dari 1, properti invers sempurna ini tidak selalu terpenuhi karena adanya *subsampling* yang menyebabkan hilangnya informasi.

5.1.5. Fungsi Konvolusi

Laluan maju dalam conv2d memanfaatkan transformasi im2col untuk mengubah operasi konvolusi menjadi perkalian matriks biasa. Filter di-*reshape* menjadi matriks dengan cara yang kompatibel dengan luaran im2col. Hal ini memungkinkan dilakukannya operasi GEMM (*general matrix multiply*) dengan sangat cepat dan efisien oleh pustaka BLAS (atau cuBLAS pada *platform* CUDA) untuk menghitung seluruh luaran. Setelah perkalian matriks, hasil di-*reshape* dan ditransposisi untuk mendapatkan format tensor luaran dengan ranking 4 sesuai harapan.

Gradien terhadap masukan dihitung dengan menerapkan konvolusi tertransposisi menggunakan col2im. Gradien terhadap filter dihitung melalui perkalian matriks antara masukan yang telah ditransformasi (`col_input`) dengan gradien *upstream*. Untuk bias, gradien diperoleh dengan menjumlahkan gradien *upstream* sepanjang dimensi *batch* dan spasial. Ini menggambarkan bias uang dibagikan sepanjang seluruh lokasi spasial.

```
def conv2d(
    input: Tensor,
    weight: Tensor,
    bias: Tensor | None = None,
    stride: int | tuple[int, int] = 1,
    padding: int | tuple[int, int] | str = 0,
) -> Tensor:
    if isinstance(stride, int):
```

```
        stride_h = stride_w = stride
    else:
        stride_h, stride_w = stride

    N, C_in, H, W = input.shape
    C_out, C_in_w, kernel_h, kernel_w = weight.shape

    # Penanganan padding
    if isinstance(padding, str):
        if padding == "valid":
            pad_h = pad_w = 0
        elif padding == "same":
            pad_h = ((H - 1) * stride_h + kernel_h - H) // 2
            pad_w = ((W - 1) * stride_w + kernel_w - W) // 2
        else:
            raise ValueError(f"Unknown padding mode: {padding}")
    elif isinstance(padding, int):
        pad_h = pad_w = padding
    else:
        pad_h, pad_w = padding

    # Hitung ukuran luaran
    H_out = (H + 2 * pad_h - kernel_h) // stride_h + 1
    W_out = (W + 2 * pad_w - kernel_w) // stride_w + 1

    # Forward pass using im2col
    col_input = im2col(input.data, kernel_h, kernel_w, stride_h, stride_w, pad_h,
pad_w)
    col_W = weight.data.reshape(C_out, -1).T

    out = col_input @ col_W
    out = out.reshape(N, H_out, W_out, C_out).transpose(0, 3, 1, 2)

    # Tambahkan bias jika tersedia
    if bias is not None:
        out = out + bias.data.reshape(1, -1, 1, 1)

    requires_grad = (
        input.requires_grad
        or weight.requires_grad
        or (bias is not None and bias.requires_grad)
    )
    result = Tensor(out, requires_grad=requires_grad)

    def backward_fn(upstream_grad: NDArray):
        # Gradien terhadap input
        input_grad = None
        if input.requires_grad:
            dout_reshaped = upstream_grad.transpose(0, 2, 3, 1).reshape(-1, C_out)
            dcol = dout_reshaped @ col_W.T
```

```
            input_grad = col2im(
                dcol, input.shape, kernel_h, kernel_w, stride_h, stride_w, pad_h,
pad_w
            )

        # Gradien terhadap filter
        weight_grad = None
        if weight.requires_grad:
            dout_reshaped = upstream_grad.transpose(0, 2, 3, 1).reshape(-1, C_out)
            weight_grad = col_input.T @ dout_reshaped
            weight_grad = weight_grad.T.reshape(C_out, C_in_w, kernel_h, kernel_w)

        # Gradien terhadap bias
        bias_grad = None
        if bias is not None and bias.requires_grad:
            bias_grad = upstream_grad.sum(axis=(0, 2, 3))

        return [input_grad, weight_grad, bias_grad]

    result.backward_fn = backward_fn
    result.inputs = [input, weight] if bias is None else [input, weight, bias]

    return result
```

5.1.6. Modul Conv2d

```
class Conv2d(Module):
    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        kernel_size: int | tuple[int, int],
        stride: int | tuple[int, int] = 1,
        padding: int | tuple[int, int] | str = 0,
        bias: bool = True,
        padding_mode: str = "zeros",
    ):
        super().__init__()

        if padding_mode != "zeros":
            raise NotImplementedError(f"padding_mode={padding_mode} is not
implemented")

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = (
            kernel_size
            if isinstance(kernel_size, tuple)
            else (kernel_size, kernel_size)
        )
```

```
        self.stride = stride if isinstance(stride, tuple) else (stride, stride)
        self.padding = padding

        # Inisialisasi Kaiming He
        kernel_h, kernel_w = self.kernel_size
        fan_in = in_channels * kernel_h * kernel_w
        std = np.sqrt(2.0 / fan_in)

        self.weight = Tensor(
            np.random.normal(0, std, (out_channels, in_channels, kernel_h, kernel_w)),
            requires_grad=True,
        )

        if bias:
            self.bias = Tensor(np.zeros(out_channels), requires_grad=True)
        else:
            self.bias = None

    def forward(self, x: Tensor) -> Tensor:
        return conv2d(
            x,
            self.weight,
            self.bias,
            self.stride,
            self.padding,
        )
```

Inisialisasi parameter dalam Conv2d menggunakan metode inisialisasi Kaiming He *initialization* [1]. Simpangan baku untuk inisialisasi filter dihitung berdasarkan *fan-in*, yaitu jumlah koneksi masukan ke setiap neuron luaran. Untuk konvolusi, *fan-in* adalah perkalian dari jumlah kanal masukan dengan ukuran filter. Pendekatan ini membantu menjaga variansi aktivasi agar tetap stabil sepanjang lapisan dan menghindari masalah hilangnya gradien (*vanishing gradients*) atau ledakan gradien (*gradient explision*).

Catatan

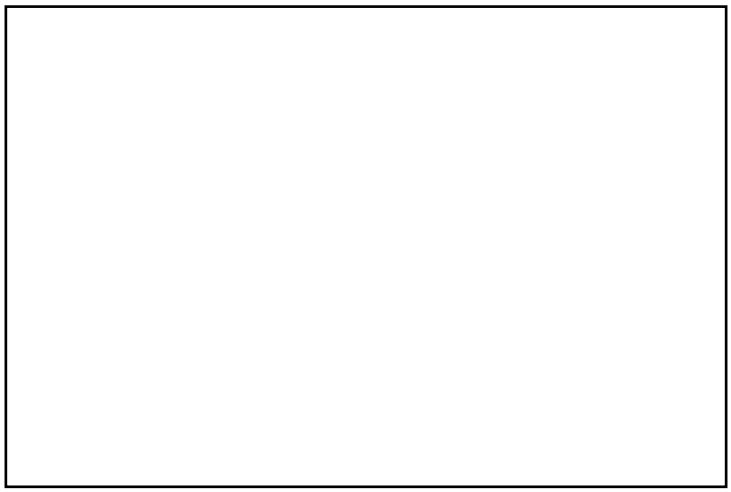
Ingat kembali kita pernah membahas inisialisasi ini sebelumnya pada implementasi lapisan linier. Bedanya, pada lapisan linier *fan-in* dan *fan-out* merujuk pada jumlah neuron. Pada konvolusi, keduanya merujuk pada jumlah kanal masukan dan luaran.

5.2. Lapisan *Pooling*

Lapisan berfungsi untuk mengurangi dimensi spasial peta fitur sembari mempertahankan informasi yang penting. Dalam hal ini, informasi yang dianggap penting adalah nilai maksimal dari pada jendela geser. Berbeda dengan konvolusi yang mempelajari filter melalui pelatihan, pooling menggunakan operasi deterministik tanpa parameter yang dapat dipelajari.

Operasi pooling memiliki beberapa manfaat utama, antara lain, mengurangi jumlah parameter dan komputasi dalam jaringan, membantu model menjadi lebih robust terhadap pergeseran kecil

dalam masukan, dan mengambil fitur yang paling menonjol dari setiap wilayah pada citra. Operasi pooling yang paling umum adalah *max pooling*, yang mengambil nilai maksimum dari setiap jendela. Untuk citra, ini setara dengan mengambil piksel paling terang dalam setiap wilayah.



Gambar 10: Ilustrasi operasi max pooling 2×2 dengan stride 2 pada peta fitur

5.2.1. Max Pooling dengan Im2Col

Menariknya, kita dapat menggunakan kembali infrastruktur im2col yang sudah kita bangun untuk konvolusi. Idenya sederhana: ekstrak tutupan seperti biasa, kemudian ambil nilai maksimum dari setiap tutupan alih-alih melakukan perkalian matriks.

Untuk max pooling dengan filter berukuran spasial $k_h \times k_w$ dan *stride* (langkah pergeseran) $s_h \times s_w$, ukuran luaran dihitung dengan:

$$h_{\text{out}} = \left\lfloor \frac{h_{\text{in}} + 2p_h - k_h}{s_h} \right\rfloor + 1 \tag{5.2}$$

$$w_{\text{out}} = \left\lfloor \frac{w_{\text{in}} + 2p_w - k_w}{s_w} \right\rfloor + 1 \tag{5.3}$$

dengan h_{in} dan w_{in} adalah tinggi dan lebar citra masukan, k_h dan k_w adalah ukuran kernel pooling, p_h dan p_w adalah *padding* yang ditambahkan, serta h_{out} dan w_{out} adalah dimensi spasial luaran yang dihasilkan.

Berikut implementasi max pooling menggunakan im2col:

```
def max_pool2d(
    input: Tensor,
    kernel_size: int | tuple[int, int],
    stride: int | tuple[int, int] | None = None,
    padding: int | tuple[int, int] = 0,
) -> Tensor:
    if isinstance(kernel_size, int):
        kernel_h = kernel_w = kernel_size
    else:
        kernel_h, kernel_w = kernel_size
```

```
    if stride is None:
        stride_h = stride_w = kernel_h
    elif isinstance(stride, int):
        stride_h = stride_w = stride
    else:
        stride_h, stride_w = stride

    if isinstance(padding, int):
        pad_h = pad_w = padding
    else:
        pad_h, pad_w = padding

    N, C, H, W = input.shape
    H_out = (H + 2 * pad_h - kernel_h) // stride_h + 1
    W_out = (W + 2 * pad_w - kernel_w) // stride_w + 1

    # Ekstraksi tutupan
    col = im2col(input.data, kernel_h, kernel_w, stride_h, stride_w, pad_h, pad_w)
    col = col.reshape(N * H_out * W_out, C, kernel_h * kernel_w)

    # Max pooling
    max_idx = col.argmax(axis=2)
    out = col.max(axis=2)
    out = out.reshape(N, H_out, W_out, C).transpose(0, 3, 1, 2)

    result = Tensor(out, requires_grad=input.requires_grad)

    def backward_fn(upstream_grad: NDArray) -> list[NDArray | None]:
        if not input.requires_grad:
            return [None]

        # Siapkan gradien
        dout = upstream_grad.transpose(0, 2, 3, 1).reshape(-1)

        # Buat matriks gradien jarang (sparse)
        # Kita perlu menempatkan gradien pada posisi yang ditunjukkan oleh max_idx
        dmax = np.zeros((N * H_out * W_out, C * kernel_h * kernel_w))

        # Penyebaran tervektorisasi menggunakan indeks lanjutan
        rows = np.arange(N * H_out * W_out)[:, None]
        cols = np.arange(C)[None, :] * kernel_h * kernel_w + max_idx
        dmax[rows, cols] = dout.reshape(N * H_out * W_out, C)

        # Konversi kembali ke bentuk citra
        dx = col2im(
            dmax, input.shape, kernel_h, kernel_w, stride_h, stride_w, pad_h, pad_w
        )

        return [dx]
```

```
    result.backward_fn = backward_fn
    result.inputs = [input]

    return result
```

5.2.2. Fungsi Laluan Mundur untuk Max Pooling

Laluan mundur untuk max pooling memiliki karakteristik unik: gradien hanya mengalir melalui elemen yang memiliki nilai maksimum dalam setiap jendela pooling. Elemen lain menerima gradien nol.

Catatan

Dalam implementasi di atas, kita menggunakan array `max_idx` untuk melacak posisi nilai maksimum di setiap jendela. Saat propagasi balik, gradien dari lapisan berikutnya diteruskan hanya ke posisi-posisi tersebut.

Pendekatan vektorisasi yang kita gunakan menghindari iterasi eksplisit dengan memanfaatkan *advanced indexing* NumPy:

```
# Penyebaran tervektorisasi menggunakan pengindeksan lanjutan
rows = np.arange(N * H_out * W_out)[:, None]
cols = np.arange(C)[None, :] * kernel_h * kernel_w + max_idx
dmax[rows, cols] = dout.reshape(N * H_out * W_out, C)
```

Teknik ini jauh lebih efisien daripada iterasi manual melalui setiap posisi.

5.2.3. Modul MaxPool2d

Seperti halnya Conv2d, kita juga membuat kelas MaxPool2d yang mengikuti API Module:

```
class MaxPool2d(Module):
    def __init__(
        self,
        kernel_size: int | tuple[int, int],
        stride: int | tuple[int, int] | None = None,
        padding: int | tuple[int, int] = 0,
    ):
        super().__init__()

        self.kernel_size = (
            kernel_size
            if isinstance(kernel_size, tuple)
            else (kernel_size, kernel_size)
        )
        self.stride = stride
        self.padding = padding

    def forward(self, x: Tensor) -> Tensor:
        return max_pool2d(x, self.kernel_size, self.stride, self.padding)
```

Perhatikan bahwa MaxPool2d tidak memiliki parameter yang dapat dipelajari, sehingga kita tidak perlu menginisialisasi bobot atau bias.

5.2.4. Jenis Pooling Lainnya

Meskipun max pooling adalah yang paling populer, ada beberapa variasi pooling lainnya:

1. **Average Pooling**: Mengambil rata-rata nilai dalam setiap jendela
2. **Global Average Pooling**: Mengambil rata-rata seluruh peta fitur menjadi satu nilai per kanal
3. **Adaptive Pooling**: Menyesuaikan ukuran kernel secara otomatis untuk menghasilkan ukuran luaran yang diinginkan

Implementasi average pooling serupa dengan max pooling, hanya mengganti operasi `max` dengan `mean` dan menyesuaikan propagasi mundurnya.

5.3. Lapisan Normalisasi

Lapisan normalisasi dalam arsitektur *deep learning* membantu menstabilkan dan mempercepat proses pelatihan. Ide dasarnya sederhana: normalisasi nilai aktivasi untuk mengurangi pergeseran distribusi internal (*internal covariate shift*) yang terjadi selama pelatihan.

Ketika kita melatih model *deep learning*, distribusi masukan ke setiap lapisan berubah seiring pembaruan parameter lapisan sebelumnya. Fenomena ini memaksa setiap lapisan untuk terus beradaptasi dengan distribusi masukan yang berubah, memperlambat konvergensi dan membuat pelatihan lebih sulit.

5.3.1. Normalisasi *Batch*

Normalisasi *batch* (*batch normalization*) menormalkan aktivasi di sepanjang dimensi *batch*. Untuk memahami operasinya, perhatikan tensor masukan $\mathbf{X} \in \mathbb{R}^{m \times d}$ dengan m sampel dan d fitur. Normalisasi *batch* menghitung statistik untuk setiap fitur di sepanjang dimensi *batch*.

Untuk fitur ke-j (kolom ke-j dari $\mathbf{X}$), dengan elemen-elemen $\{x_{1j}, x_{2j}, ..., x_{mj}\}$:

$$\begin{aligned} \mu_j &= \frac{1}{m} \sum_{i=1}^{m} x_{ij} \\ \sigma_j^2 &= \frac{1}{m} \sum_{i=1}^{m} (x_{ij} - \mu_j)^2 \\ \hat{x}_{ij} &= \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \\ y_{ij} &= \gamma_j \hat{x}_{ij} + \beta_j \end{aligned} \tag{5.4}$$

dengan γ_j dan β_j adalah parameter yang dapat dipelajari untuk fitur ke-j. Operasi ini dilakukan untuk setiap fitur $j = 1, ..., d$.

Implementasi normalisasi *batch* harus menangani dua mode berbeda: mode latih dan mode evaluasi. Saat latih, kita menghitung statistik dari *batch* saat ini. Saat evaluasi, kita menggunakan statistik yang telah diakumulasi selama pelatihan.

```
class BatchNorm1d(Module):
    def __init__(self, num_features: int, eps: float = 1e-5, momentum: float = 0.1):
        super().__init__()
        self.num_features = num_features
        self.eps = eps
        self.momentum = momentum

        # Parameter yang dapat dipelajari
        self.gamma = Tensor(np.ones(num_features), requires_grad=True)
        self.beta = Tensor(np.zeros(num_features), requires_grad=True)

        # Running statistics untuk mode evaluasi
        self.running_mean = np.zeros(num_features)
        self.running_var = np.ones(num_features)

    def forward(self, x: Tensor) -> Tensor:
        if self.training:
            # Hitung statistik batch
            batch_mean = tensor_mean(x, dim=0, keepdims=True)
            batch_var = tensor_mean((x - batch_mean) * (x - batch_mean),
                                    dim=0, keepdims=True)

            # Update running statistics (tanpa gradien)
            self.running_mean = (1 - self.momentum) * self.running_mean + \
                                self.momentum * batch_mean.data.squeeze()
            self.running_var = (1 - self.momentum) * self.running_var + \
                               self.momentum * batch_var.data.squeeze()

            # Normalisasi
            x_norm = (x - batch_mean) / sqrt(batch_var + self.eps)
        else:
            # Gunakan running statistics
            mean = Tensor(self.running_mean.reshape(1, -1))
            var = Tensor(self.running_var.reshape(1, -1))
            x_norm = (x - mean) / sqrt(var + self.eps)

        # Skala dan geser
        return self.gamma * x_norm + self.beta
```

Untuk lapisan konvolusional, kita perlu versi 2D yang menormalkan sepanjang dimensi *batch* dan spasial:

```
class BatchNorm2d(Module):
    def __init__(self, num_features: int, eps: float = 1e-5, momentum: float = 0.1):
        super().__init__()
        self.num_features = num_features
        self.eps = eps
        self.momentum = momentum

        # Parameter per kanal
```

```
        self.gamma = Tensor(np.ones(num_features), requires_grad=True)
        self.beta = Tensor(np.zeros(num_features), requires_grad=True)

        # Running statistics per kanal
        self.running_mean = np.zeros(num_features)
        self.running_var = np.ones(num_features)

    def forward(self, x: Tensor) -> Tensor:
        # x shape: (batch, channels, height, width)
        if self.training:
            # Statistik per kanal: mean dan var sepanjang batch, height, width
            batch_mean = tensor_mean(x, dim=(0, 2, 3), keepdims=True)
            batch_var = tensor_mean((x - batch_mean) * (x - batch_mean),
                                    dim=(0, 2, 3), keepdims=True)

            # Update running statistics
            self.running_mean = (1 - self.momentum) * self.running_mean + \
                                self.momentum * batch_mean.data.squeeze()
            self.running_var = (1 - self.momentum) * self.running_var + \
                               self.momentum * batch_var.data.squeeze()

            x_norm = (x - batch_mean) / sqrt(batch_var + self.eps)
        else:
            # Reshape untuk broadcasting
            mean = Tensor(self.running_mean.reshape(1, -1, 1, 1))
            var = Tensor(self.running_var.reshape(1, -1, 1, 1))
            x_norm = (x - mean) / sqrt(var + self.eps)

        # Reshape gamma dan beta untuk broadcasting
        gamma = self.gamma.reshape(1, -1, 1, 1)
        beta = self.beta.reshape(1, -1, 1, 1)

        return gamma * x_norm + beta
```

Implementasi kita menyimpan `running_mean` dan `running_var` sebagai larik NumPy array, bukan Tensor. Ini karena kedua statistik tersebut tidak perlu gradien dan hanya digunakan saat inferensi.

5.3.2. Normalisasi Lapisan

Normalisasi lapisan (layer normalization) menormalkan aktivasi di sepanjang dimensi fitur alih-alih dimensi *batch*. Ini membuatnya lebih cocok untuk arsitektur rekuren dan *transformer* di mana ukuran batch bisa sangat kecil atau bahkan 1.

Untuk masukan $\mathbf{x} \in \mathbb{R}^d$, normalisasi lapisan menghitung:

$$
\begin{aligned}
\mu &= \frac{1}{d}\sum_{i=1}^{d} x_i \\
\sigma^2 &= \frac{1}{d}\sum_{i=1}^{d} (x_i - \mu)^2 \\
\hat{\mathbf{x}} &= \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \\
\mathbf{y} &= \boldsymbol{\gamma} \odot \hat{\mathbf{x}} + \boldsymbol{\beta}
\end{aligned}
\tag{5.5}
$$

Perbedaan utama dengan batch normalisasi *batch* adalah statistik dihitung per sampel, bukan per *batch*. Ini membuat layer norm tidak bergantung pada ukuran batch dan tidak memerlukan statistik running untuk mode evaluasi.

```python
class LayerNorm(Module):
    def __init__(self, normalized_shape: int | tuple[int, ...], eps: float = 1e-5):
        super().__init__()
        if isinstance(normalized_shape, int):
            normalized_shape = (normalized_shape,)
        self.normalized_shape = normalized_shape
        self.eps = eps

        self.gamma = Tensor(np.ones(normalized_shape), requires_grad=True)
        self.beta = Tensor(np.zeros(normalized_shape), requires_grad=True)

    def forward(self, x: Tensor) -> Tensor:
        ndim = len(self.normalized_shape)
        axes = tuple(range(-ndim, 0))

        mean = x.mean(dim=axes, keepdims=True)
        var = ((x - mean) * (x - mean)).mean(dim=axes, keepdims=True)

        x_norm = (x - mean) / sqrt(var + self.eps)

        return self.gamma * x_norm + self.beta
```

Pendalaman

Ada perdebatan dalam komunitas tentang penempatan normalisasi lapisan: *pre-norm* (sebelum sub-lapisan) vs *post-norm* (setelah sub-lapisan). *Pre-norm* cenderung lebih stabil untuk model yang sangat dalam, sementara post-norm sering memberikan performa akhir yang lebih baik dengan *tuning* yang tepat.

5.4. Lapisan Rekuren

Lapisan rekuren dirancang untuk memproses data sekuensial seperti teks, suara, atau deret waktu. Berbeda dengan lapisan laluan maju biasa yang memproses setiap masukan secara independen (seperti pada lapisan linier dan konvolusi), lapisan rekuren memiliki melibatkan informasi dari langkah waktu sebelumnya untuk melakukan pemrosesan pada langkah waktu saat ini.

5.4.1. *Neural Network* Rekuren (*Recurrent Neural Network*) "Vanila"

RNN vanila merupakan bentuk paling sederhana dari jaringan rekuren. Pada setiap langkah waktu t, RNN menerima masukan $\mathbf{x}_t$ dan *hidden state* sebelumnya $\mathbf{h}_{t-1}$, kemudian menghasilkan hidden state baru $\mathbf{h}_t$. Secara matematis, operasi RNN didefinisikan sebagai:

$$\begin{aligned} \mathbf{h}_t &= \tanh(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h) \\ \mathbf{y}_t &= \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y \end{aligned} \tag{5.6}$$

dengan $\mathbf{W}_{xh}$ adalah matriks bobot *input-to-hidden*, $\mathbf{W}_{hh}$ adalah matriks bobot *hidden-to-hidden*, dan $\mathbf{W}_{hy}$ adalah matriks bobot *hidden-to-output*.

```
class RNNCell(Module):
    def __init__(self, input_size: int, hidden_size: int):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        # Input to hidden
        self.w_ih = Tensor(
            np.random.randn(input_size, hidden_size) * 0.01,
            requires_grad=True
        )
        # Hidden to hidden
        self.w_hh = Tensor(
            np.random.randn(hidden_size, hidden_size) * 0.01,
            requires_grad=True
        )
        # Bias
        self.b_h = Tensor(
            np.zeros(hidden_size),
            requires_grad=True
        )

    def forward(self, x: Tensor, h_prev: Tensor) -> Tensor:
        # h_t = tanh(W_ih @ x_t + W_hh @ h_prev + b_h)
        return tanh(x @ self.w_ih + h_prev @ self.w_hh + self.b_h)
```

```
class RNN(Module):
    def __init__(self, input_size: int, hidden_size: int, num_layers: int = 1):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # Tumpuk RNN cells
        self.cells: list[RNNCell] = []
        for i in range(num_layers):
            layer_input_size = input_size if i == 0 else hidden_size
            cell = RNNCell(layer_input_size, hidden_size)
```

```
            setattr(self, f"cell_{i}", cell)
            self.cells.append(cell)

    def forward(self, x: Tensor, h_0: Tensor | None = None) -> tuple[Tensor, Tensor]:
        batch_size, seq_len, _ = x.shape

        # Inisialisasi hidden state mula-mula
        if h_0 is None:
            h_0 = Tensor(np.zeros((self.num_layers, batch_size, self.hidden_size)))

        # Pemrosesan tiap langkah waktu
        outputs = []
        h_t = h_0

        for t in range(seq_len):
            # Seperti yang sudah kita implementasikan, indexing ini
            # memiliki backward_fn!
            x_t = x[:, t, :]  # (batch, input_size)

            # Pemrosesan tiap lapisan
            h_t_new = []
            for layer in range(self.num_layers):
                cell = self.cells[layer]
                h_prev = h_t[layer, :, :]
                h_curr = cell(x_t if layer == 0 else h_t_new[-1], h_prev)
                h_t_new.append(h_curr)

            h_t = stack(h_t_new)  # (num_layers, batch, hidden_size)
            outputs.append(h_t_new[-1])  # Gunakan luaran dari lapisan sebelumnya

        output = stack(outputs, dim=1)  # (batch, seq_len, hidden_size)
        return output, h_t
```

Catatan

Perhatikan bahwa kita menggunakan `setattr` untuk mendaftarkan *cells* sebagai submodul. Alasannya adalah kita membuat *cells* dalam perulangan dengan jumlah dinamis. Python tidak mengizinkan kita menulis `self.cell_{i} = cell` karena `{i}` bukan sintaks yang valid. Dengan `setattr(self, f'cell_{i}', cell)`, kita bisa membuat nama atribut secara dinamis dalam perulangan. Hasilnya sama, yaitu setiap *cell* tetap terdaftar di modul dan parameternya terdeteksi oleh pengoptimal.

Masalah utama RNN vanila adalah *vanishing gradient* dan *exploding gradient* saat memproses sekuens panjang. Gradien cenderung mengecil atau membesar secara eksponensial saat dipropagasi balik melalui banyak langkah waktu.

5.4.2. *Long Short Term Memory* (LSTM)

Long short term memory (LSTM) dikembangkan untuk menanggulangi permasalahan gradien yang menghilang, yang dimiliki RNN. LSTM memanfaatkan mekanisme gerbang (*gate*) untuk menyaring informasi pada langkah waktu mana yang layak dijaga dan mana yang layak dilupakan. Ada 4 gerbang yang dimiliki oleh LSTM:

LSTM memiliki arsitektur yang lebih kompleks dibandingkan RNN vanila. Arsitektur ini terdiri dari tiga gerbang utama dan satu cell state:

- ***Forget gate***: Menentukan informasi mana dari cell state sebelumnya yang perlu dilupakan
- ***Input gate***: Menentukan nilai-nilai baru mana yang akan disimpan dalam *cell state*
- ***Output gate***: Menentukan bagian mana dari cell state yang akan dijadikan luaran sebagai *hidden state*

Untuk setiap langkah waktu t, LSTM menghitung:

$$
\begin{aligned}
\mathbf{f}_t &= \sigma\big(\mathbf{W}_f \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f\big) && \text{(\textit{Forget gate})} \\
\mathbf{i}_t &= \sigma(W_i \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) && \text{(\textit{Input gate})} \\
\tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}_c \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c) && \text{(Kandidat \textit{cell state})} \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t && \text{(\textit{Cell state} baru)} \\
\mathbf{o}_t &= \sigma(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) && \text{(\textit{Output gate})} \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t) && \text{(\textit{Hidden state} baru)}
\end{aligned}
\tag{5.7}
$$

di mana σ adalah fungsi sigmoid, $\odot$ adalah perkalian antar-elemen, dan $\left[\mathbf{h}_{\{t-1\}}, \mathbf{x}_t\right]$ adalah penggabungan (*concatenation*) dari *hidden state* sebelumnya dan masukan saat ini.

```python
class LSTMCell(Module):
    def __init__(self, input_size: int, hidden_size: int):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        # Gabungkan semua gerbang dalam satu transformasi linear untuk efisiensi
        # Urutan: forget, input, cell, output
        self.i2h = Linear(input_size, 4 * hidden_size)
        self.h2h = Linear(hidden_size, 4 * hidden_size, bias=False)

        self.reset_parameters()

    def reset_parameters(self):
        # Inisialisasi Xavier untuk weight
        std = (2.0 / (self.input_size + self.hidden_size)) ** 0.5
        for param in self.parameters():
            param.data = np.random.normal(0, std, param.data.shape)

        # Inisialisasi bias forget gate ke 1 (standar praktik)
        forget_bias_start = self.hidden_size
        forget_bias_end = 2 * self.hidden_size
```

```
        self.i2h.bias.data[forget_bias_start:forget_bias_end] = 1.0

    def forward(self, x: Tensor, hidden: tuple[Tensor, Tensor]) -> tuple[Tensor,
Tensor]:
        h_prev, c_prev = hidden

        # Transformasi linear untuk masukan dan hidden state
        gi = self.i2h(x)  # gate masukan
        gh = self.h2h(h_prev)  # gate hidden
        i_f, i_i, i_g, i_o = split(gi, 4, dim=1)
        h_f, h_i, h_g, h_o = split(gh, 4, dim=1)

        # Hitung gerbang-gerbang
        forget_gate = sigmoid(i_f + h_f)
        input_gate = sigmoid(i_i + h_i)
        cell_gate = tanh(i_g + h_g)
        output_gate = sigmoid(i_o + h_o)

        # Update cell state dan hidden state
        c_new = forget_gate * c_prev + input_gate * cell_gate
        h_new = output_gate * tanh(c_new)

        return h_new, c_new
```

Perhatikan bahwa alih-alih membuat empat transformasi linier terpisah untuk setiap gerbang seperti ini

```
self.forget_gate = Linear(input_size, hidden_size)
self.input_gate = Linear(input_size, hidden_size)
self.cell_gate = Linear(input_size, hidden_size)
self.output_gate = Linear(input_size, hidden_size)
```

Kita menggabungkan semuanya dalam dua transformasi besar dengan hasil yang identik:

```
self.i2h = Linear(input_size, 4 * hidden_size)     # input-to-hidden
self.h2h = Linear(hidden_size, 4 * hidden_size, bias=False)  # hidden-to-hidden
```

Dengan begini, kita bisa mengurangi operasi matriks, yaitu empat perkalian matriks kecil diganti dengan dua perkalian matriks besar. Ini menguntungkan kita bila menggunakan BLAS di mana ia bekerja lebih efisien untuk matriks besar.

Kemudian kita implementasikan pada kode python:

```
class LSTM(Module):
   def __init__(
       self,
       input_size: int,
       hidden_size: int,
       num_layers: int = 1
   ):
```

```
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # Buat LSTM cells untuk setiap layer
        self.cells = []
        for layer in range(num_layers):
            layer_input_size = input_size if layer == 0 else hidden_size
            cell = LSTMCell(layer_input_size, hidden_size)
            self.cells.append(cell)
            setattr(self, f'cell_{layer}', cell)

    def forward(
        self,
        x: Tensor,
        hidden: tuple[Tensor, Tensor] | None = None
    ) -> tuple[Tensor, tuple[Tensor, Tensor]]:
        batch_size, seq_len, _ = x.shape

        # Inisialisasi hidden state jika tidak diberikan
        if hidden is None:
            h_t = zeros((self.num_layers, batch_size, self.hidden_size))
            c_t = zeros((self.num_layers, batch_size, self.hidden_size))
        else:
            h_t, c_t = hidden


        # Proses setiap langkah waktu
        outputs = []
        for t in range(seq_len):
            x_t = x[:, t, :]  # (batch, input_size)

            # Proses tiap layer
            h_t_new = []
            c_t_new = []
            for layer in range(self.num_layers):
                cell = self.cells[layer]
                h_prev = h_t[layer, :, :]
                c_prev = c_t[layer, :, :]

                # Masukan ke layer ini adalah luaran layer sebelumnya atau masukan asli
                layer_input = x_t if layer == 0 else h_t_new[-1]

                h_curr, c_curr = cell(layer_input, (h_prev, c_prev))
                h_t_new.append(h_curr)
                c_t_new.append(c_curr)

            h_t = stack(h_t_new)  # (num_layers, batch, hidden_size)
            c_t = stack(c_t_new)  # (num_layers, batch, hidden_size)
```

```
            outputs.append(h_t_new[-1])  # Luaran dari layer terakhir

        output = stack(outputs, dim=1)  # (batch, seq_len, hidden_size)

        return output, (h_t, c_t)
```

5.4.3. *Gated Recurrent Unit* (GRU)

Gated recurrent unit (GRU) adalah varian arsitektur rekuren yang diperkenalkan oleh Cho et al. [7] sebagai alternatif yang lebih sederhana dari LSTM. GRU mempertahankan kemampuan untuk menangani dependensi jangka panjang namun dengan arsitektur yang lebih ringkas. Perbedaan utamanya adalah GRU menggabungkan *forget gate* dan *input gate* menjadi satu *update gate*, serta menggabungkan *cell state* dan *hidden state*.

Arsitektur GRU hanya memiliki dua gerbang:

- ***Update gate*** (z_t): Menentukan seberapa banyak informasi dari langkah waktu sebelumnya yang dipertahankan
- ***Reset gate*** (r_t): Mengontrol seberapa banyak informasi masa lalu yang diabaikan saat menghitung kandidat hidden state

Secara matematis, operasi GRU pada langkah waktu t adalah:

$$\begin{aligned}
\mathbf{r}_t &= \sigma(\mathbf{W}_r \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_r) && \text{(\textit{Reset gate})} \\
\mathbf{z}_t &= \sigma(\mathbf{W}_z \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_z) && \text{(\textit{Update gate})} \\
\tilde{\mathbf{h}}_t &= \tanh(\mathbf{W}_h \cdot [\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_h) && \text{(Kandidat \textit{hidden state})} \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t && \text{(\textit{Hidden state} baru)}
\end{aligned} \tag{5.8}$$

Perhatikan bagaimana *update gate* $\mathbf{z}_t$ berfungsi sebagai interpolasi linier antara *hidden state* sebelumnya dan kandidat baru. Ketika $\mathbf{z}_t$ mendekati 0, GRU mempertahankan informasi lama sepenuhnya. Sebaliknya, ketika $\mathbf{z}_t$ mendekati 1, GRU mengganti *hidden state* dengan informasi baru.

```
class GRUCell(Module):
    def __init__(self, input_size: int, hidden_size: int):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        # Gabungkan reset dan update gates untuk efisiensi
        # Urutan: reset, update
        self.i2h = Linear(input_size, 2 * hidden_size)
        self.h2h = Linear(hidden_size, 2 * hidden_size, bias=False)

        # Kandidat hidden state
        self.i2n = Linear(input_size, hidden_size)
        self.h2n = Linear(hidden_size, hidden_size, bias=False)

        self.reset_parameters()
```

```
    def reset_parameters(self):
        # Inisialisasi Xavier
        std = (2.0 / (self.input_size + self.hidden_size)) ** 0.5
        for param in self.parameters():
            param.data = np.random.normal(0, std, param.data.shape)

    def forward(self, x: Tensor, h_prev: Tensor) -> Tensor:
        # Hitung reset dan update gates
        gi = self.i2h(x)
        gh = self.h2h(h_prev)
        i_r, i_z = split(gi, 2, dim=1)
        h_r, h_z = split(gh, 2, dim=1)

        reset_gate = sigmoid(i_r + h_r)
        update_gate = sigmoid(i_z + h_z)

        # Kandidat hidden state dengan reset gate
        n_t = tanh(self.i2n(x) + self.h2n(reset_gate * h_prev))

        # Interpolasi antara hidden state lama dan kandidat baru
        h_new = (1 - update_gate) * h_prev + update_gate * n_t

        return h_new
```

Implementasi GRUCell menggunakan strategi yang sama dengan LSTM untuk efisiensi, yaitu menggabungkan transformasi linier untuk kedua gerbang. Perhatikan juga penggunaan *reset gate* pada perhitungan kandidat *hidden state*. Operasi `reset_gate * h_prev` memungkinkan model untuk "melupakan" bagian tertentu dari hidden state sebelumnya saat menghitung kandidat baru.

Berikut implementasi modul utama GRU:

```
class GRU(Module):
    def __init__(self, input_size: int, hidden_size: int, num_layers: int = 1):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # Buat GRU cells untuk setiap lapisan
        self.cells = []
        for layer in range(num_layers):
            layer_input_size = input_size if layer == 0 else hidden_size
            cell = GRUCell(layer_input_size, hidden_size)
            self.cells.append(cell)
            setattr(self, f'cell_{layer}', cell)

    def forward(
        self,
        x: Tensor,
        h_0: Tensor | None = None
```

```
    ) -> tuple[Tensor, Tensor]:
        batch_size, seq_len, _ = x.shape

        # Inisialisasi hidden state jika tidak diberikan
        if h_0 is None:
            h_t = Tensor(np.zeros((self.num_layers, batch_size, self.hidden_size)))
        else:
            h_t = h_0

        # Proses setiap langkah waktu
        outputs = []
        for t in range(seq_len):
            x_t = x[:, t, :]  # (batch, input_size)

            # Proses tiap lapisan
            h_t_new = []
            for layer in range(self.num_layers):
                cell = self.cells[layer]
                h_prev = h_t[layer, :, :]

                # Masukan ke lapisan ini adalah luaran lapisan sebelumnya
                # atau masukan awal
                layer_input = x_t if layer == 0 else h_t_new[-1]

                h_curr = cell(layer_input, h_prev)
                h_t_new.append(h_curr)

            h_t = stack(h_t_new)  # (num_layers, batch, hidden_size)
            outputs.append(h_t_new[-1])  # Luaran dari layer terakhir

        output = stack(outputs, dim=1)  # (batch, seq_len, hidden_size)

        return output, h_t
```

Kesederhanaan GRU dibandingkan LSTM membuatnya lebih cepat untuk dilatih dan membutuhkan parameter yang lebih sedikit. Dalam praktiknya, performa GRU sering kali sebanding dengan LSTM untuk berbagai tugas pemrosesan sekuens [8]. Pilihan antara LSTM dan GRU juga bergantung pada dataset spesifik dan kebutuhan komputasi. Untuk dataset yang lebih kecil atau ketika kecepatan pelatihan menjadi prioritas, GRU sering menjadi pilihan yang lebih baik.

Catatan

Meskipun GRU memiliki parameter yang lebih sedikit dibandingkan LSTM, mekanisme gerbangnya tetap efektif dalam menangani masalah gradien yang menghilang. Reset gate memungkinkan model untuk membuang informasi yang tidak relevan, sementara update gate memungkinkan propagasi gradien jangka panjang dengan mengontrol seberapa banyak informasi yang dipertahankan dari langkah waktu sebelumnya.

5.5. Lapisan *Attention*

Mekanisme *attention* merupakan salah satu inovasi terpenting dalam arsitektur *deep learning* modern. Konsep ini pertama kali diperkenalkan oleh Bahdanau *et al.* [9] untuk mengatasi keterbatasan arsitektur *encoder-decoder* tradisional dalam menerjemahkan kalimat panjang. Ide dasarnya sederhana namun powerful: alih-alih memaksa model untuk mengompresi seluruh informasi masukan ke dalam satu vektor *hidden state* tetap, *attention* memungkinkan model untuk "memperhatikan" bagian-bagian relevan dari masukan secara dinamis.

5.5.1. Motivasi dan Intuisi

Bayangkan kita sedang menerjemahkan kalimat "*The agreement on the European Economic Area was signed in August 1992*" ke bahasa Indonesia. Saat menerjemahkan kata "ditandatangani", Anda secara alami akan fokus pada kata "*signed*" dalam kalimat sumber. Inilah esensi dari mekanisme *attention*: memberikan bobot perhatian yang berbeda pada bagian-bagian masukan berdasarkan relevansinya terhadap langkah pemrosesan saat ini.

Pada arsitektur *seq2seq* tradisional dengan LSTM, seluruh kalimat sumber harus dikodekan menjadi satu vektor *context* berukuran tetap. Ini menjadi hambatan informasi, terutama untuk sekuens panjang. *Attention* mengatasi masalah ini dengan mempertahankan akses ke semua *hidden state* dari *encoder* dan menghitung kombinasi tertimbang yang relevan untuk setiap langkah dekoding.

5.5.2. *Attention* Dasar: Bahdanau *Attention*

Mekanisme *attention* Bahdanau menghitung skor relevansi antara *hidden state decoder* saat ini dengan setiap *hidden state encoder*. Skor-skor ini kemudian dinormalisasi menggunakan *softmax* untuk menghasilkan bobot *attention*.

Secara formal, untuk *hidden state decoder* $\mathbf{s}_t$ pada langkah waktu t dan *hidden state encoder* $\mathbf{h}_i$ untuk posisi masukan i, kita hitung:

$$\begin{aligned} e_{ti} &= f_{\text{att}(\mathbf{s}_{t-1}, \mathbf{h}_i)} && \text{(Skor } \textit{attention}\text{)} \\ \alpha_{ti} &= \frac{\exp(e_{ti})}{\sum_{j=1}^{T} \exp(e_{tj})} && \text{(Bobot } \textit{attention}\text{)} \\ \mathbf{c}_t &= \sum_{i=1}^{T} \alpha_{ti} \mathbf{h}_i && \text{(Vektor } \textit{context}\text{)} \end{aligned} \tag{5.9}$$

dengan f_{att} adalah fungsi skor yang dapat diimplementasikan dengan berbagai cara. Bahdanau menggunakan *feed-forward network* satu lapis:

$$f_{\text{att}(\mathbf{s}_{t-1}, \mathbf{h}_i)} = \mathbf{v}^\top \tanh(\mathbf{W}_s \mathbf{s}_{t-1} + \mathbf{W}_h \mathbf{h}_i) \tag{5.10}$$

```
class BahdanauAttention(Module):
    def __init__(self, hidden_size: int, attention_size: int):
        super().__init__()
        self.hidden_size = hidden_size
        self.attention_size = attention_size

        # Proyeksi untuk hidden state decoder
```

```
        self.W_s = Linear(hidden_size, attention_size, bias=False)
        # Proyeksi untuk hidden state encoder
        self.W_h = Linear(hidden_size, attention_size, bias=False)
        # Vektor untuk menghitung skor akhir
        self.v = Linear(attention_size, 1, bias=False)

    def forward(
        self,
        decoder_hidden: Tensor,  # (batch, hidden_size)
        encoder_outputs: Tensor  # (batch, seq_len, hidden_size)
    ) -> tuple[Tensor, Tensor]:
        batch_size, seq_len, _ = encoder_outputs.shape

        # Proyeksikan decoder hidden state
        # (batch, attention_size) -> (batch, 1, attention_size)
        s_proj = self.W_s(decoder_hidden).unsqueeze(1)

        # Proyeksikan semua encoder outputs
        # (batch, seq_len, hidden_size) -> (batch, seq_len, attention_size)
        h_proj = self.W_h(encoder_outputs)

        # Hitung skor attention
        # (batch, seq_len, attention_size)
        combined = tanh(s_proj + h_proj)
        # (batch, seq_len, 1) -> (batch, seq_len)
        scores = self.v(combined).squeeze(-1)

        # Normalisasi dengan softmax
        attention_weights = softmax(scores, dim=1)

        # Hitung context vector sebagai weighted sum
        # (batch, seq_len) -> (batch, seq_len, 1)
        attention_weights_expanded = attention_weights.unsqueeze(-1)
        # (batch, seq_len, hidden_size) * (batch, seq_len, 1) -> sum -> (batch,
hidden_size)
        context = (encoder_outputs * attention_weights_expanded).sum(dim=1)

        return context, attention_weights
```

Implementasi di atas menunjukkan bagaimana mekanisme *attention* menghitung relevansi antara setiap posisi *encoder* dengan keadaan *decoder* saat ini. Metode `unsqueeze` dan operasi *broadcasting* memungkinkan kita menghitung semua skor secara paralel, tanpa perulangan eksplisit.

5.5.3. Luong *Attention*: Penyederhanaan dan Variasinya

Luong *et al.*[10] mengusulkan beberapa penyederhanaan dan variasi dari mekanisme *attention* Bahdanau. Perbedaan utamanya adalah Luong *attention* menggunakan *hidden state* saat ini ($\mathbf{s}_t$) alih-alih *hidden state* sebelumnya ($\mathbf{s}_{t-1}$), dan menawarkan beberapa fungsi skor alternatif:

1. ***Dot product***: $f_{\text{att}(\mathbf{s}_t, \mathbf{h}_i)} = \mathbf{s}_t^\top \mathbf{h}_i$
2. ***General***: $f_{\text{att}(\mathbf{s}_t, \mathbf{h}_i)} = \mathbf{s}_t^\top \mathbf{W} \mathbf{h}_i$

3. ***Concat***: $f_{\text{att}(\mathbf{s}_t, \mathbf{h}_i)} = \mathbf{v}^\top \tanh(\mathbf{W}[\mathbf{s}_t; \mathbf{h}_i])$

```
class LuongAttention(Module):
    def __init__(self, hidden_size: int, method: str = "dot"):
        super().__init__()
        self.hidden_size = hidden_size
        self.method = method

        if method == "general":
            self.W = Linear(hidden_size, hidden_size, bias=False)
        elif method == "concat":
            self.W = Linear(hidden_size * 2, hidden_size, bias=False)
            self.v = Linear(hidden_size, 1, bias=False)
        elif method ≠ "dot":
            raise ValueError(f"Unknown attention method: {method}")

    def forward(
        self,
        decoder_hidden: Tensor,  # (batch, hidden_size)
        encoder_outputs: Tensor  # (batch, seq_len, hidden_size)
    ) -> tuple[Tensor, Tensor]:
        batch_size, seq_len, _ = encoder_outputs.shape

        if self.method == "dot":
            # Simple dot product
            # (batch, hidden_size) @ (batch, hidden_size, seq_len) -> (batch, seq_len)
            scores = (decoder_hidden.unsqueeze(1) @ encoder_outputs.transpose(-2,
-1)).squeeze(1)

        elif self.method == "general":
            # Weighted dot product
            # (batch, hidden_size) -> (batch, 1, hidden_size)
            decoder_hidden_proj = self.W(decoder_hidden).unsqueeze(1)
            # (batch, 1, hidden_size) @ (batch, hidden_size, seq_len) -> (batch,
1, seq_len)
            scores = (decoder_hidden_proj @ encoder_outputs.transpose(-2,
-1)).squeeze(1)

        elif self.method == "concat":
            # Concatenation-based
            # Expand decoder hidden untuk setiap posisi encoder
            decoder_hidden_expanded = decoder_hidden.unsqueeze(1).expand(-1, seq_len,
-1)
            # Concat: (batch, seq_len, hidden_size * 2)
            concat = torch.cat([decoder_hidden_expanded, encoder_outputs], dim=-1)
            # Score: (batch, seq_len)
            scores = self.v(tanh(self.W(concat))).squeeze(-1)

        # Normalisasi dan hitung context
        attention_weights = softmax(scores, dim=1)
```

```
        context = (encoder_outputs * attention_weights.unsqueeze(-1)).sum(dim=1)

        return context, attention_weights
```

Metode *dot product* adalah yang paling sederhana dan efisien secara komputasi, namun mensyaratkan *encoder* dan *decoder* memiliki dimensi *hidden state* yang sama. Metode *general* menambahkan matriks bobot yang dapat dipelajari untuk transformasi linier, memberikan fleksibilitas lebih. Metode *concat* paling ekspresif namun juga paling mahal dari sisi penggunaan memori dan komputasi.

Catatan

Meskipun Bahdanau dan Luong attention mengatasi masalah hambatan informasi, arsitektur tetap bergantung pada RNN yang harus memproses sekuens secara berurutan. Hal ini membatasi paralelisasi dan memperlambat training untuk sekuens panjang.

5.5.4. *Self-Attention*: Fondasi *Transformer*

Dari penelitian populer bertajuk "*Attention is All You Need*", Vaswani *et al.* [11] memperkenalkan *self-attention*. Versi ini adalah varian khusus di mana mekanisme *attention* dapat "memperhatikan" semua posisi lain dalam sekuens yang sama. Ini menjadi komponen fundamental arsitektur *Transformer* yang akan kita bahas di bagian selanjutnya.

Pada *self-attention*, kita memproyeksikan masukan yang sama menjadi tiga representasi berbeda:

- ***Query*** $(\mathbf{Q})$: Apa yang dicari oleh posisi saat ini
- ***Key*** $(\mathbf{K})$: Apa yang ditawarkan oleh setiap posisi
- ***Value*** $(\mathbf{V})$: Informasi aktual yang akan diagregasi

$$\begin{aligned} \mathbf{Q} &= \mathbf{X}\mathbf{W}_Q \\ \mathbf{K} &= \mathbf{X}\mathbf{W}_K \\ \mathbf{V} &= \mathbf{X}\mathbf{W}_V \\ \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) &= \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V} \end{aligned} \tag{5.11}$$

dengan d_k adalah dimensi *key*, dan pembagian dengan $\sqrt{d_k}$ untuk menjaga stabilitas numerik saat dimensi besar.

Perhatikan bahwa pada persamaan di atas, sama sekali tidak ada komponen yang melibatkan langkah waktu sebelumnya $(t-1)$ yang diproses secara berulang. Ini menunjukkan bahwa *self-attention* berbeda secara fundamental dari kedua versi *attention* sebelumnya, karena *self-attention* tidak menggunakan arsitektur rekuren **sama sekali**. Semua *embedding* dalam sekuens diproses **secara sekaligus**.

Tabel 6 menunjukkan beberapa masalah RNN yang dipecahkan oleh *self-attention*, sekaligus titik awal RNN mulai ditinggalkan.

	RNN	***Self-Attention***

Paralelisasi	Posisi t harus menunggu posisi $t-1$ selesai diproses	Semua posisi dihitung dalam satu operasi matriks seperti ditunjukkan pada oleh fungsi $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$
Akses jarak jauh	Informasi dari posisi ke-1 ke posisi ke-100 harus melewati 99 langkah	Posisi ke-100 dapat langsung "melihat" posisi 1
Alur gradien	Gradien dari posisi ke-100 ke posisi ke-1 harus melewati 99 perkalian matriks	Gradien dapat mengalir langsung melalui bobot *attention*

Tabel 6: Permasalahan RNN yang diselesaikan oleh *self-attention*

Pengembangan lanjutan dari *self-attention* adalah *multi-head attention*, yang memungkinkan model untuk secara serentak memperhatikan informasi dari representasi subruang yang berbeda. Setiap *head* dapat belajar untuk menangkap jenis hubungan yang berbeda dalam data. Misalnya, satu *head* mungkin fokus pada hubungan sintaksis, sementara *head* lain menangkap hubungan semantik.

Berikut implementasinya dalam kode Python.

```
class SelfAttention(Module):
    def __init__(self, embed_dim: int, num_heads: int = 1):
        super().__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        assert self.head_dim * num_heads == embed_dim, \
            "embed_dim must be divisible by num_heads"

        # Proyeksi untuk Q, K, V
        self.W_q = Linear(embed_dim, embed_dim)
        self.W_k = Linear(embed_dim, embed_dim)
        self.W_v = Linear(embed_dim, embed_dim)

        # Proyeksi output
        self.W_o = Linear(embed_dim, embed_dim)

        self.scale = self.head_dim ** -0.5

    def forward(self, x: Tensor, mask: Tensor | None = None) -> Tensor:
        batch_size, seq_len, embed_dim = x.shape

        # Hitung Q, K, V
        Q = self.W_q(x)  # (batch, seq_len, embed_dim)
        K = self.W_k(x)
        V = self.W_v(x)

        # Reshape untuk multi-head attention
        # (batch, seq_len, num_heads, head_dim) -> (batch, num_heads, seq_len,
head_dim)
        Q = Q.reshape(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1,
2)
```

```
        K = K.reshape(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1,
2)
        V = V.reshape(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1,
2)

        # Hitung attention scores
        # (batch, num_heads, seq_len, head_dim) @ (batch, num_heads, head_dim,
seq_len)
        # -> (batch, num_heads, seq_len, seq_len)
        scores = (Q @ K.transpose(-2, -1)) * self.scale

        # Apply mask jika ada (untuk padding atau causal attention)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # Softmax untuk mendapatkan attention weights
        attention_weights = softmax(scores, dim=-1)

        # Apply attention ke values
        # (batch, num_heads, seq_len, seq_len) @ (batch, num_heads, seq_len, head_dim)
        # -> (batch, num_heads, seq_len, head_dim)
        attended = attention_weights @ V

        # Reshape kembali
        # (batch, num_heads, seq_len, head_dim) -> (batch, seq_len, embed_dim)
        attended = attended.transpose(1, 2).reshape(batch_size, seq_len, embed_dim)

        # Final projection
        output = self.W_o(attended)

        return output
```

Catatan

Konsep *query*, *key*, dan *value* dalam *self-attention* terinspirasi dari sistem temu balik informasi (*information retrieval*). Bayangkan sebuah basis data di mana setiap entri memiliki *key* (identitas) dan *value* (konten). Untuk mengakses informasi, kita menggunakan *query* yang dicocokkan dengan semua *key* untuk menemukan *value* yang relevan. Dalam konteks *attention*, "pencocokan" dilakukan melalui *dot product* dan "relevansi" dihitung melalui *softmax*.

Pendalaman

Kompleksitas komputasi *self-attention* adalah $O(n^2 d)$ dengan n adalah panjang sekuens dan d adalah dimensi model. Ini menjadi *bottleneck* untuk sekuens yang sangat panjang. Berbagai varian efisien telah diusulkan, seperti *Linformer* [12] yang mengurangi kompleksitas menjadi $O(nd)$ dengan memproyeksikan matriks *attention* ke dimensi yang lebih rendah, atau

Reformer [13] yang menggunakan *locality-sensitive hashing* untuk mengurangi kompleksitas menjadi $O(n \log n)$.

5.5.4.1. *Multi-Head Attention*

5.5.5. *Masked Attention* dan *Causal Attention*

Dalam beberapa aplikasi, kita perlu membatasi posisi mana yang dapat "diperhatikan" oleh posisi tertentu. Dua jenis pembatasan yang umum adalah:

1. ***Padding mask***: Mengabaikan posisi *padding* dalam sekuens yang panjangnya bervariasi
2. ***Causal mask***: Mencegah posisi untuk melihat posisi "masa depan" (untuk model autoregresif)

Padding mask diperlukan karena kita sering memproses sekuens dalam *batch* dengan panjang yang berbeda. Sekuens yang lebih pendek diisi dengan token *padding*, dan kita tidak ingin model memperhatikan posisi-posisi ini.

Causal mask penting untuk model generatif seperti GPT, di mana prediksi token ke-i hanya boleh bergantung pada token 1 hingga $i - 1$. Ini memastikan model dapat digunakan untuk generasi autoregresif tanpa *information leakage*.

Implementasi *masking* dilakukan dengan menambahkan nilai negatif yang sangat besar (misalnya -10^9) pada skor *attention* untuk posisi yang di-*mask*. Setelah *softmax*, posisi-posisi ini akan memiliki bobot mendekati nol.

5.5.6. Visualisasi dan Interpretasi *Attention*

Salah satu keunggulan mekanisme *attention* adalah interpretabilitasnya. Bobot *attention* dapat divisualisasikan sebagai *heatmap* yang menunjukkan posisi mana yang "diperhatikan" model saat memproses posisi tertentu. Ini memberikan wawasan tentang apa yang dipelajari model dan dapat membantu dalam *debugging* atau analisis kesalahan.

Dalam tugas penerjemahan, visualisasi *attention* sering menunjukkan pola diagonal yang menandakan korespondensi kata-per-kata, dengan deviasi menarik untuk konstruksi gramatikal yang berbeda antar bahasa. Untuk tugas *question answering*, *attention* weights dapat mengungkapkan bagian mana dari konteks yang dianggap relevan untuk menjawab pertanyaan.

Catatan

Meskipun visualisasi *attention* memberikan intuisi yang berguna, interpretasi harus dilakukan dengan hati-hati. Penelitian terbaru menunjukkan bahwa bobot *attention* tidak selalu berkorelasi langsung dengan kepentingan fitur, dan model mungkin menggunakan mekanisme lain bersamaan dengan *attention* untuk membuat keputusan.

Bab 6. Transformer

6.1. Komponen Penyusun

6.1.1. *Positional Encoding*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

```
class PositionalEncoding(Module):
    def __init__(self, d_model: int, max_len: int = 5000):
        super().__init__()

        pe = np.zeros((max_len, d_model))
        pos = np.arange(max_len)[:, np.newaxis]  # (max_len, 1)

        div_term = np.exp(
            np.arange(0, d_model, 2) * (-np.log(10000.0) / d_model)
        )  # (d_model/2,)

        pe[:, 0::2] = np.sin(pos * div_term)  # even indices
        pe[:, 1::2] = np.cos(pos * div_term)  # odd indices

        self.pe = pe  # NumPy array, will not registered as parameter

    def forward(self, x: Tensor) -> Tensor:
        # x: (batch, seq_len, d_model)
        seq_len = x.shape[1]
        return x + Tensor(self.pe[:seq_len])
```

6.1.2. *Multi-head Attention*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime

placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

6.1.3. *Feed-Forward Network*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

6.1.4. *Residual Connection*

6.2. Blok *Transformer*

6.2.1. Blok *Transformer* Sebagai Modul

```
class TransformerBlock(Module):
    def __init__(self, d_model: int, num_heads: int, d_ff: int, dropout: float = 0.0):
        super().__init__()

        self.attention = SelfAttention(d_model, num_heads)
        self.norm1 = LayerNorm(d_model)
        self.norm2 = LayerNorm(d_model)

        self.ffn = Sequential(
            Linear(d_model, d_ff),
            ReLU(),
            Linear(d_ff, d_model),
        )

    def forward(self, x: Tensor, mask: Tensor | None = None) -> Tensor:
        # Self-attention + residual + norm
        attn_out = self.attention(x, mask)
        x = self.norm1(x + attn_out)

        # FFN + residual + norm
        ffn_out = self.ffn(x)
        x = self.norm2(x + ffn_out)

        return x
```

6.2.2. *Transformer Encoder*

```
class TransformerEncoder(Module):
    def __init__(
        self,
        num_layers: int,
```

```
        d_model: int,
        num_heads: int,
        d_ff: int,
        vocab_size: int,
        max_len: int = 512,
    ):
        super().__init__()

        self.embedding = Embedding(vocab_size, d_model)
        self.pos_encoding = PositionalEncoding(d_model, max_len)

        self.blocks = []
        for i in range(num_layers):
            block = TransformerBlock(d_model, num_heads, d_ff)
            setattr(self, f"block_{i}", block)
            self.blocks.append(block)

    def forward(self, x: Tensor, mask: Tensor | None = None) -> Tensor:
        x = self.embedding(x)
        x = self.pos_encoding(x)

        for block in self.blocks:
            x = block(x, mask)

        return x
```

6.2.3. Penggunaan *Transformer Encoder* untuk Klasifikasi

```
class TransformerClassifier(Module):
    def __init__(
        self,
        num_layers: int,
        d_model: int,
        num_heads: int,
        d_ff: int,
        vocab_size: int,
        num_classes: int,
        max_len: int = 512,
    ):
        super().__init__()

        self.encoder = TransformerEncoder(
            num_layers, d_model, num_heads, d_ff, vocab_size, max_len
        )
        self.classifier = Linear(d_model, num_classes)

    def forward(self, x: Tensor, mask: Tensor | None = None) -> Tensor:
        encoded = self.encoder(x, mask)  # (batch, seq_len, d_model)
        pooled = encoded[:, 0, :]  # CLS token (first position)
        return self.classifier(pooled)
```

6.3.

Daftar Pustaka

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[3] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, and others, "Improving language understanding by generative pre-training," 2018.

[4] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[5] S. K. Kumar, "On weight initialization in deep neural networks," *arXiv preprint arXiv:1704.08863*, 2017.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[7] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[8] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[9] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[10] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[11] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[12] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *arXiv preprint arXiv:2006.04768*, 2020.

[13] N. Kitaev, Ł. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *arXiv preprint arXiv:2001.04451*, 2020.

Daftar Gambar

Daftar Tabel