Pertemuan 14 — *Attention* dan *Transformer*

Membiarkan setiap kata menengok langsung ke kata lain

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Data berurutan, token, dan *embedding*
2. RNN dan masalah *gradient*
3. LSTM, *encoder-decoder*, dan tiga keterbatasannya
4. Mekanisme *attention*
5. *Self-attention*: *query*, *key*, *value*
6. Contoh terhitung dan *causal mask*
7. *Multi-head attention*
8. *Positional encoding*
9. Blok *encoder* dan *decoder*
10. Mengapa *transformer* mudah diparalelkan
11. Tiga keluarga model *transformer*
12. Praktikum: menelusuri *self-attention* dan membaca makalahnya

Sasaran pertemuan ini

Anda dapat menjelaskan mengapa RNN dan LSTM kesulitan pada urutan panjang, menghitung *self-attention* dengan tangan pada matriks kecil, lalu menguraikan peran *multi-head attention*, *positional encoding*, dan blok *encoder–decoder* di dalam *transformer*.

Data berurutan

Menyambung dari Pertemuan 13

CNN berbagi *weight* **sepanjang ruang**: *filter* yang sama digeser ke seluruh citra. Hari ini datanya berupa **urutan**, dan pertanyaannya bergeser: bagaimana berbagi *weight* sepanjang waktu, dan bagaimana menghubungkan dua posisi yang berjauhan?

Coba baca kalimat ini

"Nasabah itu menutup rekeningnya di bank karena **ia** kecewa dengan **layanannya**."

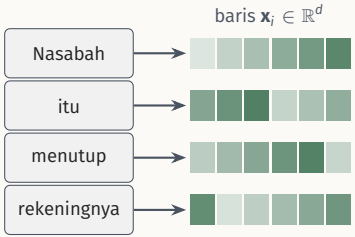
Siapa "ia"? Milik siapa "layanannya"? Anda menjawab tanpa berpikir: "ia" adalah nasabah, dan "layanannya" milik bank. Untuk itu Anda menghubungkan kata yang berjauhan dan menimbang mana yang relevan.

Sekarang bayangkan sebuah program yang membaca kalimat itu kata demi kata dari kiri ke kanan, dengan ingatan berupa satu vektor berukuran tetap. Saat tiba di "ia", informasi tentang "nasabah" sudah tertimpa oleh "rekening", "bank", dan "karena". Itulah kesulitan model urutan generasi awal.

Tiga sifat data berurutan, dan cara teks menjadi matriks

1. **Panjang bervariasi.** Kalimat bisa tiga kata, bisa tiga ratus. MLP menuntut *input* berukuran tetap.
2. **Urutan bermakna.** “Anjing menggigit pria” berbeda dengan “pria menggigit anjing”, padahal kata-katanya sama.
3. **Ketergantungan jarak jauh.** Makna sebuah kata dapat ditentukan kata yang berjarak puluhan posisi, seperti “ia” tadi.

Teks dipecah menjadi ***token***: kata, potongan kata, atau karakter (rinciannya minggu depan). Setiap token punya nomor di ***vocabulary***, lalu nomor itu diubah menjadi vektor berdimensi d lewat tabel ***embedding*** $E \in \mathbb{R}^{|V| \times d}$ yang *weight*-nya dipelajari.



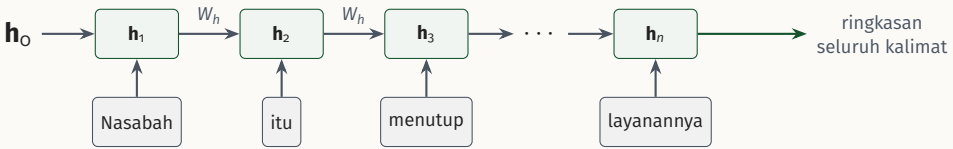
Kalimat n token menjadi matriks $X \in \mathbb{R}^{n \times d}$, satu baris per token. Seluruh pertemuan ini bekerja pada matriks semacam ini.

Recurrent neural network dan LSTM

RNN: ingatan yang diperbarui langkah demi langkah

RNN membaca urutan satu per satu dan membawa vektor ingatan, ***hidden state*** $\mathbf{h}_t$:

$$\mathbf{h}_t = \tanh\left(W_h \mathbf{h}_{t-1} + W_x \mathbf{x}_t + \mathbf{b}\right)$$



- W_h, W_x, **b sama** di semua langkah: *weight sharing* sepanjang waktu, padanan *weight sharing* sepanjang ruang pada CNN. Urutan sepanjang apa pun dibaca dengan parameter tetap.
- Setelah dibuka (*unrolled*), RNN tampak seperti jaringan dalam biasa, satu *layer* per langkah, dan dilatih dengan ***backpropagation through time*** (BPTT).

Vanishing dan *exploding gradient* pada RNN

Gradient loss terhadap *hidden state* awal adalah perkalian banyak matriks Jacobian:

$$\frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_1} = \prod_{t=2}^{n} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \prod_{t=2}^{n} \operatorname{diag}\left(1 - \mathbf{h}_t^2\right) W_h$$

Notasi: $\prod$: perkalian semua suku; $\partial \mathbf{h}_t / \partial \mathbf{h}_{t-1}$: matriks Jacobian (*partial derivative*); $\operatorname{diag}(\cdot)$: matriks diagonal dari sebuah vektor.

Perkalian berulang berperilaku seperti pangkat. Versi skalarnya, setelah 50 langkah:

Faktor/langkah	**Setelah 50 langkah**	
0,9	$0{,}9^{50} \approx 0{,}005$	*vanishing*
1,1	$1{,}1^{50} \approx 117$	*exploding*

Sinyal dari kata yang jauh di belakang nyaris tidak sampai, atau pembaruan *weight* melompat liar. Bengio dkk. (1994) menunjukkan secara formal mengapa ketergantungan jangka panjang sulit dipelajari dengan *gradient descent*.

Gradient clipping

Exploding gradient relatif mudah diatasi: bila *gradient norm* melewati ambang, misalnya 1,0, *gradient* diskalakan turun ke ambang itu. Di PyTorch, sebelum `opt.step()`:

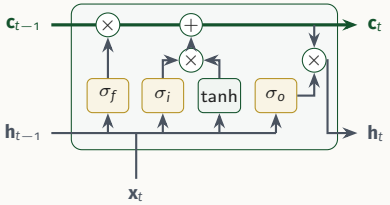
```
nn.utils.clip_grad_norm_(
  model.parameters(), 1.0)
```

Teknik ini tetap dipakai saat melatih *transformer* besar. *Vanishing gradient* lebih sulit, dan LSTM dirancang untuk masalah itu.

LSTM: jalur ingatan kedua yang diperbarui dengan penjumlahan

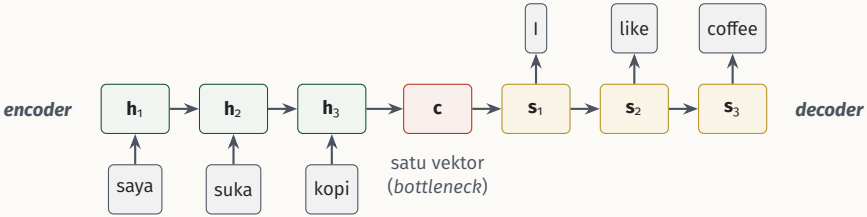
Hochreiter dan Schmidhuber (1997) menambah ***cell state*** $\mathbf{c}_t$ dan tiga ***gate*** bernilai antara 0 dan 1. Ditulis $\mathbf{z}_t = [\mathbf{h}_{t-1}, \mathbf{x}_t]$:

$\mathbf{f}_t = \sigma(W_f \mathbf{z}_t + \mathbf{b}_f)$	*forget gate*
$\mathbf{i}_t = \sigma(W_i \mathbf{z}_t + \mathbf{b}_i)$	*input gate*
$\tilde{\mathbf{c}}_t = \tanh(W_c \mathbf{z}_t + \mathbf{b}_c)$	calon isi baru
$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$	pembaruan
$\mathbf{o}_t = \sigma(W_o \mathbf{z}_t + \mathbf{b}_o)$	*output gate*
$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$	



Jalur atas (hijau) hanya disentuh perkalian dengan $\mathbf{f}_t$ dan penjumlahan isi baru. Bila $\mathbf{f}_t \approx 1$, *cell state* dan *gradient*-nya mengalir hampir utuh, mirip jalan pintas identitas ResNet. GRU adalah varian ringkas dengan dua *gate*.

Encoder-decoder dan tiga keterbatasannya



Untuk terjemahan, Sutskever dkk. dan Cho dkk. (2014) memakai dua RNN: ***encoder*** meringkas kalimat sumber menjadi satu vektor konteks, ***decoder*** membangkitkan kalimat sasaran dari vektor itu (*sequence-to-sequence*).

1. *Bottleneck*

Kalimat 10 kata dan 100 kata dijejalkan ke vektor berukuran sama.

2. Jalur panjang

Informasi dari posisi 1 ke posisi 100 melewati 99 langkah pembaruan.

3. Sekuensial

$\mathbf{h}_t$ menunggu $\mathbf{h}_{t-1}$, sehingga ribuan inti GPU menganggur.

Mekanisme *attention*

Gagasannya: jangan meringkas, tengoklah

Bahdanau, Cho, dan Bengio (2015): biarkan *decoder* menengok **semua** *hidden state encoder* di setiap langkah, lalu memilih yang relevan. Saat menerjemahkan “coffee”, *decoder* seharusnya paling banyak melihat “kopi”.

Pembanding: *dictionary* Python. `d[q]` mencari satu *key* yang persis sama dengan *query* lalu mengembalikan satu *value*. *Attention* adalah versi lunaknya: *query* dibandingkan dengan **semua** *key*, dan hasilnya **campuran dengan *weight*** dari semua *value*, dengan *weight* sebanding kemiripan.

Tiga langkah *attention*

Keadaan *decoder* $\mathbf{s}$ sebagai *query*; *hidden state encoder* $\mathbf{h}_1, \ldots, \mathbf{h}_n$ sebagai *key* sekaligus *value*.

1. Skor: $e_j = \text{skor}(\mathbf{s}, \mathbf{h}_j)$
2. *Weight*: $\alpha_j = \dfrac{\exp(e_j)}{\sum_l \exp(e_l)}$
3. Konteks: $\mathbf{c} = \sum_j \alpha_j \mathbf{h}_j$

Bahdanau memakai skor jaringan kecil, $\mathbf{v}^\top \tanh(W\mathbf{s} + U\mathbf{h}_j)$ (***additive attention***). Luong dkk. memakai $e_j = \mathbf{s}^\top \mathbf{h}_j$ (***dot-product attention***), yang dapat dihitung sebagai perkalian matriks cepat dan kemudian dipakai *transformer*.

Contoh terhitung: *dot-product attention* dengan tiga kata

Hidden state encoder: $\mathbf{h}_1 = (1, 0)$, $\mathbf{h}_2 = (0, 1)$, $\mathbf{h}_3 = (1, 1)$. Keadaan *decoder*: $\mathbf{s} = (1, 2)$.

1. Skor

$$e_1 = 1 \cdot 1 + 2 \cdot 0 = 1, \quad e_2 = 1 \cdot 0 + 2 \cdot 1 = 2, \quad e_3 = 1 + 2 = 3$$

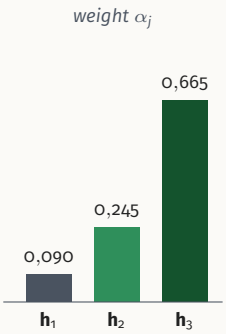
2. *Softmax*

$$e^1 = 2{,}718,\ e^2 = 7{,}389,\ e^3 = 20{,}086;\ \text{jumlah } 30{,}193$$

$$\alpha = (0{,}090;\ 0{,}245;\ 0{,}665)$$

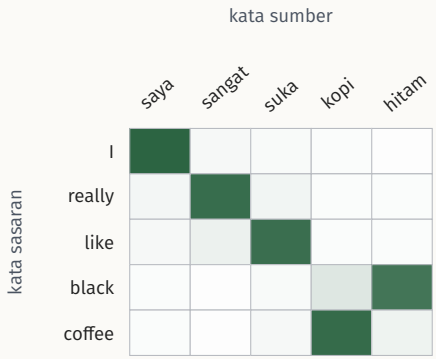
3. Konteks

$$\mathbf{c} = 0{,}090(1, 0) + 0{,}245(0, 1) + 0{,}665(1, 1) = (0{,}755;\ 0{,}910)$$



Kata ketiga paling mirip dengan *query*, sehingga *weight*-nya terbesar, tetapi kata lain tetap ikut menyumbang. Semua langkah dapat diturunkan, sehingga *weight attention* dipelajari bersama bagian model lain dengan *backpropagation*.

Weight attention dapat dibaca



Matriks α untuk terjemahan Indonesia ke Inggris: baris adalah kata sasaran yang sedang dibangkitkan, kolom adalah kata sumber yang ditengok. Makin gelap, makin besar *weight*-nya.

Perhatikan “black coffee” yang menengok “kopi hitam” dengan urutan terbalik. *Attention* menangani perbedaan susunan kata antarbahasa secara alami, tanpa aturan tata bahasa yang ditulis tangan.

Angka di gambar ini ilustratif dan tidak diambil dari model sungguhan. Pada model nyata polanya jarang sebersih ini.

Self-attention: query, key, value

Dari *attention* ke *self-attention*

Pada model Bahdanau, *attention* menghubungkan dua urutan berbeda dan masih menumpang pada RNN. ***Self-attention*** menerapkannya **di dalam satu urutan**: setiap token menengok semua token di kalimat yang sama untuk memperkaya representasinya. “Ia” dapat langsung menengok “nasabah” tanpa melewati kata-kata di antaranya.

Query $\mathbf{q}_i = \mathbf{x}_i W_Q$

“Apa yang sedang saya cari?” Kata “ia” mencari kata benda orang.

Key $\mathbf{k}_j = \mathbf{x}_j W_K$

“Apa yang saya tawarkan untuk dicocokkan?” “Nasabah” menawarkan: saya kata benda orang.

Value $\mathbf{v}_j = \mathbf{x}_j W_V$

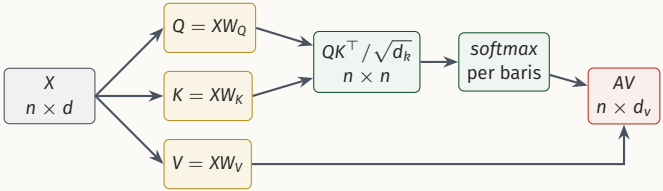
“Informasi apa yang saya serahkan bila terpilih?” “Nasabah” menyerahkan maknanya.

Setiap token memainkan ketiga peran sekaligus, dan setiap peran punya proyeksi linier sendiri yang dipelajari. Memisahkan peran ini penting: apa yang dicari sebuah kata belum tentu sama dengan apa yang ia tawarkan kepada kata lain.

Scaled dot-product attention

Susun $Q = XW_Q$, $K = XW_K$, $V = XW_V$ untuk seluruh kalimat sekaligus. *Output*-nya:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$



Baca dari dalam ke luar:

1. $QK^\top$ berukuran $n \times n$: elemen (i, j) adalah kecocokan *query* token i dengan *key* token j.
2. Dibagi $\sqrt{d_k}$ agar skor tidak terlalu besar.
3. *Softmax* per baris: baris i menjadi distribusi perhatian token i ke semua token.
4. Dikalikan V: *output* token i adalah *weighted average* dari *value* semua token.

Mengapa dibagi $\sqrt{d_k}$

Misalkan komponen **q** dan **k** saling bebas, rata-rata 0, variansi 1. Maka

$$\mathbf{q} \cdot \mathbf{k} = \sum_{l=1}^{d_k} q_l k_l \quad \text{punya } \textit{variance } d_k$$

Untuk $d_k = 64$, simpangan bakunya 8, sehingga skor mudah bernilai puluhan.

Softmax dari skor besar menjadi sangat tajam, hampir satu-panas, dan *gradient*-nya nyaris nol. Membagi dengan $\sqrt{d_k}$ mengembalikan variansi ke sekitar 1.

Dampaknya pada contoh kecil kita

Pada contoh tiga token sebentar lagi ($d_k = 2$), baris ketiga matriks *weight*:

	tok 1	tok 2	tok 3
dibagi $\sqrt{2}$	0,306	0,074	0,620
tanpa pembagi	0,259	0,035	0,705

Tanpa pembagi, *weight* sudah lebih tajam walau d_k hanya 2. Dengan $d_k = 64$ efeknya jauh lebih besar, dan *softmax* menjadi jenuh.

Contoh terhitung: tiga token, $d = d_k = d_v = 2$ (langkah 1–2)

Tiga token dengan *embedding* yang sudah memuat informasi posisi:

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, \quad W_Q = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \quad W_K = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad W_V = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

Langkah 1: proyeksi

$$Q = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 2 & 1 \end{bmatrix}, \; K = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 2 \end{bmatrix}, \; V = \begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 1 & 2 \end{bmatrix}$$

Misalnya baris ketiga Q:
$(1, 1)\, W_Q = (1 \cdot 1 + 1 \cdot 1,\; 1 \cdot 0 + 1 \cdot 1) = (2, 1)$.

Langkah 2: skor, elemen $(i, j) = \mathbf{q}_i \cdot \mathbf{k}_j$

$$QK^\top = \begin{bmatrix} 1 & 0 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 4 \end{bmatrix}$$

$$\frac{QK^\top}{\sqrt{2}} = \begin{bmatrix} 0{,}707 & 0 & 0{,}707 \\ 1{,}414 & 0{,}707 & 2{,}121 \\ 2{,}121 & 0{,}707 & 2{,}828 \end{bmatrix}$$

Contoh: $\mathbf{q}_3 \cdot \mathbf{k}_3 = 2 \cdot 1 + 1 \cdot 2 = 4$, dibagi $1{,}414$ menjadi $2{,}828$.

Contoh terhitung (langkah 3–4): *softmax* dan *weighted average*

Langkah 3: *softmax* per baris

Baris 1: $e^{0,707} = 2,028$, $e^{0} = 1$, $e^{0,707} = 2,028$; jumlah 5,056.

$$\textit{weight}\ \text{baris 1} = (0{,}401;\ 0{,}198;\ 0{,}401)$$

Dengan cara yang sama untuk baris 2 dan 3:

$$A = \begin{bmatrix} 0{,}401 & 0{,}198 & 0{,}401 \\ 0{,}284 & 0{,}140 & 0{,}576 \\ 0{,}306 & 0{,}074 & 0{,}620 \end{bmatrix}$$

Setiap baris berjumlah 1.

Langkah 4: $Z = AV$

Baris 1:

$$0{,}401(1,0) + 0{,}198(0,2) + 0{,}401(1,2)$$
$$= (0{,}802;\ 1{,}198)$$

Lengkapnya:

$$Z = \begin{bmatrix} 0{,}802 & 1{,}198 \\ 0{,}860 & 1{,}432 \\ 0{,}926 & 1{,}389 \end{bmatrix}$$

Cara membaca hasilnya

Setiap baris Z adalah representasi baru token itu yang sudah tercampur konteks. Token ketiga paling banyak memperhatikan dirinya sendiri (0,620); token pertama membagi perhatian sama rata ke token 1 dan 3.

Causal mask: dilarang mengintip masa depan

Pada model yang membangkitkan teks dari kiri ke kanan, token ke-i tidak boleh melihat token sesudahnya, karena saat dipakai token itu belum ada. Sebelum *softmax*, skor di atas diagonal diganti $-\infty$, sehingga *weight*-nya menjadi $e^{-\infty} = 0$.

$$S_{mask} = \begin{bmatrix} 0{,}707 & -\infty & -\infty \\ 1{,}414 & 0{,}707 & -\infty \\ 2{,}121 & 0{,}707 & 2{,}828 \end{bmatrix}$$

Ini ***causal mask***. Nama lainnya *look-ahead mask*.

Pada contoh kita:

- Baris 1 hanya melihat token 1: *weight* $(1;\ 0;\ 0)$.
- Baris 2 melihat token 1 dan 2: $\text{softmax}(1{,}414;\ 0{,}707) = (0{,}670;\ 0{,}330)$.
- Baris 3 tidak berubah.

$$A_{mask} = \begin{bmatrix} 1 & 0 & 0 \\ 0{,}670 & 0{,}330 & 0 \\ 0{,}306 & 0{,}074 & 0{,}620 \end{bmatrix}$$

$$Z_{mask} = \begin{bmatrix} 1 & 0 \\ 0{,}670 & 0{,}660 \\ 0{,}926 & 1{,}389 \end{bmatrix}$$

Self-attention dari nol dengan NumPy

```python
import numpy as np

def softmax(s):
    e = np.exp(s - s.max(axis=-1, keepdims=True))   # cegah overflow
    return e / e.sum(axis=-1, keepdims=True)

def self_attention(X, Wq, Wk, Wv, kausal=False):
    Q, K, V = X @ Wq, X @ Wk, X @ Wv
    S = Q @ K.T / np.sqrt(K.shape[-1])       # skor n x n
    if kausal:
        n = S.shape[0]
        atas = np.triu(np.ones((n, n)), k=1) == 1
        S = np.where(atas, -np.inf, S)
    A = softmax(S)                           # bobot per baris
    return A @ V, A

X  = np.array([[1, 0], [0, 1], [1, 1]], float)
Wq = np.array([[1, 0], [1, 1]], float)
Wk = np.array([[1, 1], [0, 1]], float)
Wv = np.array([[1, 0], [0, 2]], float)
Z, A = self_attention(X, Wq, Wk, Wv)
print(A.round(3), Z.round(3))                    # cocok dengan slide
print(self_attention(X, Wq, Wk, Wv, kausal=True)[1].round(3))
```

Kodenya hampir kata demi kata menyalin rumus: tiga proyeksi, skor dibagi akar d_k, *mask* opsional, *softmax* per baris, kalikan V.

`s - s.max(...)` tidak mengubah hasil *softmax*, karena pembilang dan penyebut sama-sama dikali e^{-m}, tetapi mencegah *overflow* pada skor besar.

Pembanding di PyTorch adalah satu fungsi:

`F.scaled_dot_product_attention(Q, K, V, is_causal=True)`

dengan `F` = `torch.nn.functional`. Fungsi ini memberi angka yang sama dengan contoh tangan kita, dan di GPU memakai kernel yang dioptimalkan.

Kesalahan umum pada *self-attention*

Yang sering salah

- ***Softmax* pada sumbu yang salah.** *Normalization* harus per baris (per *query*), `axis=-1`. Bila per kolom, *weight* tiap token tidak lagi berjumlah 1.
- **Lupa membagi $\sqrt{d_k}$.** Model tetap berjalan, tetapi *training* melambat atau macet karena *softmax* jenuh.
- ***Mask* diisi nol padahal seharusnya $-\infty$.** Mengganti skor dengan 0 tidak melarang apa pun, karena $e^0 = 1$.
- **Lupa *padding mask*.** Dalam satu *batch* berisi kalimat berbeda panjang, token pengisi juga harus di-*mask*.

Cost: kuadratik terhadap panjang

Matriks skor berukuran $n \times n$. Menghitungnya sekitar $n^2 d_k$ operasi, menyimpannya sebanding n^2. Menggandakan panjang konteks melipatempatkan *attention cost*.

Contoh: $n = 4.096$ token memberi 16,8 juta skor per *head* per *layer*; dalam bilangan 16-bit sekitar 33,5 MB, dikali jumlah *head* dan *layer*.

Inilah alasan model punya batas panjang konteks. FlashAttention (2022) menghitung *attention* eksak dengan membagi matriks menjadi blok agar hemat akses memori GPU.

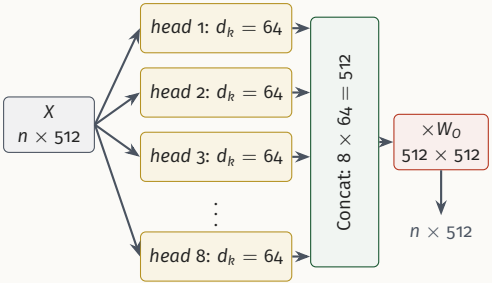
Multi-head attention

Beberapa pola perhatian sekaligus

Sebuah kata sering perlu memperhatikan beberapa hal sekaligus: subjeknya, objeknya, kata yang merujuk kepadanya, kata di sebelahnya. ***Multi-head attention*** menjalankan *h attention* paralel, masing-masing dengan proyeksinya sendiri berdimensi $d_k = d_{\text{model}}/h$:

$$\text{head}_i = \text{Attention}(XW_Q^{(i)}, XW_K^{(i)}, XW_V^{(i)})$$
$$\text{MH}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W_O$$

Karena tiap *head* berdimensi d_{model}/h, total komputasinya kira-kira sama dengan satu *head* berdimensi penuh.



Configuration transformer dasar: $d_{\text{model}} = 512$, $h = 8$. Studi pada model terlatih menunjukkan *head* berbeda cenderung mempelajari pola berbeda, misalnya kata sebelumnya atau hubungan sintaksis, walaupun tidak semua *head* mudah ditafsirkan.

Menghitung parameter, lalu memeriksanya dengan PyTorch

Hitungan tangan, $d_{\text{model}} = 512$, $h = 8$

- Tiap *head*: tiga matriks 512×64, jadi $8 \times 3 \times 512 \times 64 = 786.432$ *weight*, sama dengan tiga matriks 512×512.
- W_O: $512 \times 512 = 262.144$.
- Total *weight* $4 \times 512^2 = 1.048.576$, ditambah $4 \times 512 = 2.048$ bias.

Jumlahnya 1.050.624, dan tidak bergantung pada panjang kalimat n.

Pada `attn_mask` PyTorch, `True` berarti **dilarang** diperhatikan. Arahnya berlawanan dengan intuisi banyak orang, jadi periksa selalu dengan mencetak *weight* baris pertama.

```python
import torch
from torch import nn

n, d_model, h = 6, 512, 8
x = torch.randn(2, n, d_model)          # batch 2, 6 token
mha = nn.MultiheadAttention(d_model, h, batch_first=True)
print(sum(p.numel() for p in mha.parameters()))  # 1050624

# causal mask: True berarti DILARANG diperhatikan
mask = torch.triu(torch.ones(n, n, dtype=torch.bool),
                  diagonal=1)
z, bobot = mha(x, x, x, attn_mask=mask)
print(z.shape)       # torch.Size([2, 6, 512])
print(bobot.shape)   # [2, 6, 6], rata-rata semua head
print(bobot[0, 0])   # token 1 hanya melihat dirinya
```

Untuk *self-attention*, *query*, *key*, dan *value* diisi *tensor* yang sama: `mha(x, x, x)`. Untuk *cross-attention*, *key* dan *value* datang dari *encoder*.

Positional encoding

Self-attention tidak peduli urutan

Tukar baris-baris X, dan baris *output* ikut tertukar dengan cara yang sama tanpa berubah nilai. Sifat ini disebut ***permutation equivariance***.

Periksa dengan contoh kita. Urutan token diubah menjadi (3, 1, 2):

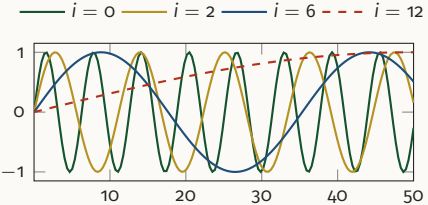
$$Z = \begin{bmatrix} 0{,}802 & 1{,}198 \\ 0{,}860 & 1{,}432 \\ 0{,}926 & 1{,}389 \end{bmatrix} \longrightarrow Z' = \begin{bmatrix} 0{,}926 & 1{,}389 \\ 0{,}802 & 1{,}198 \\ 0{,}860 & 1{,}432 \end{bmatrix}$$

Baris yang sama, hanya urutannya berpindah. Bagi *self-attention* murni, “anjing menggigit pria” dan “pria menggigit anjing” adalah himpunan kata yang sama. Posisi harus disuntikkan.

Sinusoidal positional encoding

Tambahkan vektor posisi ke *embedding*: $\mathbf{x}_{pos} \leftarrow \mathbf{x}_{pos} + \mathrm{PE}_{pos}$,

$$\mathrm{PE}_{(pos,2i)} = \sin \frac{pos}{10000^{2i/d}}, \quad \mathrm{PE}_{(pos,2i+1)} = \cos \frac{pos}{10000^{2i/d}}$$



Komponen sinus untuk $d = 64$. Makin besar i, makin lambat gelombangnya.

Menghitung *positional encoding*, dan alternatifnya

Contoh terhitung, $d = 4$

Dua pasangan dimensi: $i = 0$ dengan pembagi $10000^0 = 1$, dan $i = 1$ dengan pembagi $10000^{2/4} = 100$. Sudut dalam radian.

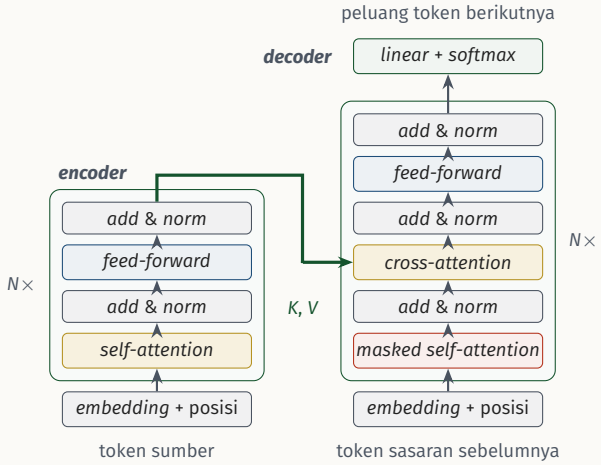
pos	$\sin(pos)$	$\cos(pos)$	$\sin(pos/100)$	$\cos(pos/100)$
0	0	1	0	1
1	0,841	0,540	0,010	1,000
2	0,909	−0,416	0,020	1,000

Dua dimensi pertama berubah drastis dari posisi ke posisi; dua terakhir hampir diam. Seperti jarum detik, menit, dan jam: dimensi cepat membedakan posisi bertetangga, dimensi lambat membedakan posisi yang berjauhan. Untuk setiap geseran tetap k, PE_{pos+k} adalah rotasi dari PE_{pos}, sehingga posisi relatif mudah dipelajari.

Cara	**Ringkasan**	**Catatan**
Sinusoidal	Rumus tetap, ditambahkan ke *embedding*	Tanpa parameter; dipakai *transformer* asli
Learned	Tabel vektor posisi yang dilatih, seperti *embedding* token	Dipakai BERT dan GPT-2; tidak bisa melampaui panjang maksimum latih
RoPE	Memutar *query* dan *key* sesuai posisi	Skor bergantung posisi relatif; umum pada LLM terbuka

Arsitektur *transformer*

Transformer: hanya *attention*, FFN, dan *normalization*



Vaswani dkk. (2017), "*Attention Is All You Need*": model *encoder-decoder* tanpa perulangan dan tanpa konvolusi.

- Setiap sub-lapis dibungkus *residual connection* dan *layer normalization*.
- *Output* akhir *encoder* menjadi *key* dan *value* bagi *cross-attention* di setiap blok *decoder*.
- *Configuration* dasar: $N = 6$ blok *encoder* dan 6 blok *decoder*, $d_{\text{model}} = 512$, $h = 8$, dimensi FFN 2.048.
- Pada terjemahan Inggris–Jerman WMT 2014, versi besarnya mencapai BLEU 28,4, melampaui hasil terbaik sebelumnya, dengan *training cost* jauh lebih kecil (sekitar 3,5 hari pada delapan GPU).

Blok *encoder*

Dua sub-lapis:

1. ***Multi-head self-attention***: setiap token mengumpulkan informasi dari semua token.
2. ***Position-wise feed-forward network*** (FFN): MLP dua *layer* yang diterapkan ke setiap token terpisah dengan *weight* sama, $\mathrm{FFN}(\mathbf{x}) = \max(0, \mathbf{x}W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$. Dimensi dalamnya biasanya $4 \times d_{\text{model}}$.

Pembagian kerjanya jelas: *attention* **mencampur informasi antartoken**, FFN **mengolah informasi di dalam setiap token**.

$$\mathbf{y} = \mathrm{LayerNorm}\left(\mathbf{x} + \mathrm{SubLapis}(\mathbf{x})\right)$$

Layer normalization

Menormalkan setiap vektor token terhadap rata-rata dan simpangan baku komponennya sendiri. *Batch normalization* dari Pertemuan 11 menormalkan per *feature* di sepanjang *batch*; LayerNorm tidak bergantung ukuran *batch* dan cocok untuk urutan berpanjang berbeda.

Post-LN dan Pre-LN

Makalah asli meletakkan LayerNorm sesudah penjumlahan residual (Post-LN), seperti rumus di kiri. Kebanyakan model sesudahnya meletakkannya sebelum sub-lapis, $\mathbf{y} = \mathbf{x} + \mathrm{SubLapis}(\mathrm{LayerNorm}(\mathbf{x}))$ (Pre-LN), karena *training* jaringan dalam lebih stabil. Banyak model juga mengganti ReLU di FFN dengan GELU atau varian sejenis.

Blok *decoder*, *training*, dan inferensi

Tiga sub-lapis:

1. ***Masked self-attention*** dengan *causal mask*: posisi t hanya melihat token sasaran sebelumnya.
2. ***Cross-attention***: *query* dari *decoder*, *key* dan *value* dari *output* akhir *encoder*. Inilah *attention* gaya Bahdanau, kini tanpa RNN.
3. **FFN**, sama seperti di *encoder*.

Output decoder diproyeksikan ke ukuran kosakata lalu dilewatkan *softmax*: peluang token berikutnya.

Saat *training*: *teacher forcing*

Kalimat sasaran sudah diketahui. *Decoder* menerima kalimat sasaran yang digeser satu posisi (diawali token mulai), dan berkat *causal mask* *prediction* untuk **semua** posisi dihitung dalam satu *forward pass*. *Loss*-nya *cross-entropy* di setiap posisi.

Saat inferensi: *autoregressive*

Jawaban belum ada. Bangkitkan satu token, tambahkan ke *input*, bangkitkan berikutnya, sampai token akhir muncul. Agar *key* dan *value* token sebelumnya tidak dihitung ulang setiap langkah, keduanya disimpan dalam ***KV cache***. Minggu depan kita melihatnya lagi pada LLM.

Satu blok *encoder* (Pre-LN) dengan PyTorch

```python
import torch
from torch import nn
class BlokEncoder(nn.Module):
    def __init__(self, d=512, h=8, d_ff=2048, p=0.1):
        super().__init__()
        self.ln1, self.ln2 = nn.LayerNorm(d), nn.LayerNorm(d)
        self.attn = nn.MultiheadAttention(d, h, dropout=p,
                                          batch_first=True)
        self.ffn = nn.Sequential(nn.Linear(d, d_ff), nn.GELU(),
                                 nn.Linear(d_ff, d), nn.Dropout(p))
    def forward(self, x, pad_mask=None):
        a = self.ln1(x)
        x = x + self.attn(a, a, a, key_padding_mask=pad_mask)[0]
        return x + self.ffn(self.ln2(x))       # dua residual

def positional_encoding(n, d):
    pos = torch.arange(n).unsqueeze(1)             # (n, 1)
    div = 10000 ** (-torch.arange(0, d, 2) / d)    # 1/10000^(2i/d)
    pe = torch.zeros(n, d)
    pe[:, 0::2], pe[:, 1::2] = torch.sin(pos * div), torch.cos(pos * div)
    return pe

x = torch.randn(2, 10, 512) + positional_encoding(10, 512)
print(BlokEncoder()(x).shape)    # torch.Size([2, 10, 512])
```

Bentuk *input* dan *output* blok sama, $2 \times 10 \times 512$. Karena itu blok dapat ditumpuk N kali tanpa penyesuaian apa pun.

`div` adalah $1/10000^{2i/d}$ untuk $i = 0, 1, \ldots, d/2 - 1$. Cetak `positional_encoding(3, 4)` dan cocokkan dengan tabel $d = 4$ dua slide lalu.

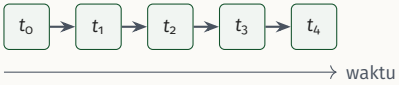
PyTorch juga menyediakan `nn.TransformerEncoderLayer` yang setara; opsi `norm_first=True` memilih bentuk Pre-LN.

Coba sendiri: hitung parameter satu blok dengan $d = 768$, $h = 12$, $d_{\text{ff}} = 3.072$. Jawabannya 7.087.872; periksa dengan `sum(p.numel() ...)`.

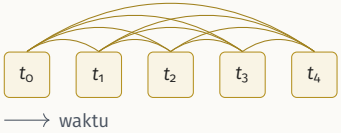
Mengapa *transformer* mudah diparalelkan

Satu langkah lawan n langkah

RNN: n langkah berurutan



Self-attention: satu langkah



Layer	***Cost/layer***	**Op. berurutan**	**Jalur maks.**
Self-attention	$O(n^2 d)$	$O(1)$	$O(1)$
Rekuren	$O(nd^2)$	$O(n)$	$O(n)$
Konvolusi	$O(knd^2)$	$O(1)$	$O(\log_k n)$

Diringkas dari Tabel 1 makalah *transformer*. n panjang urutan, d dimensi, k ukuran *kernel*.

RNN harus menghitung $\mathbf{h}_1$, lalu $\mathbf{h}_2$, dan seterusnya. *Self-attention* menghitung semua token sekaligus sebagai beberapa perkalian matriks besar, operasi yang paling disukai GPU. Jalur pendek juga berarti sinyal dan *gradient* antara dua kata berjauhan tidak melewati banyak langkah.

Harga yang dibayar: memori kuadratik

Contoh terhitung, $d = 512$

n	*self-attention* $n^2 d$	RNN nd^2
100	$5{,}1 \times 10^6$	$2{,}6 \times 10^7$
4.096	$8{,}6 \times 10^9$	$1{,}1 \times 10^9$

Untuk kalimat pendek ($n < d$), *self-attention* bahkan lebih murah daripada RNN. Untuk dokumen panjang, suku n^2 menyusul dan mendominasi.

Transformer menukar urutan dengan memori

Ia paralel dan berjalur pendek, tetapi *attention cost* tumbuh kuadratik terhadap panjang urutan. Itulah sebabnya panjang konteks LLM selalu terbatas, dan mengapa riset seperti FlashAttention dan *attention* jarang (*sparse*) terus berjalan.

Paralelisme yang sama memungkinkan *training* model raksasa di banyak GPU, topik mata kuliah Sistem Terdistribusi.

Tiga keluarga model *transformer*

Encoder-only, decoder-only, encoder-decoder

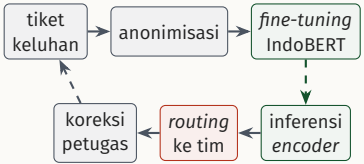
	Encoder-only	***Decoder-only***	***Encoder-decoder***
Contoh	BERT, IndoBERT	keluarga GPT, Llama	T5
Arah perhatian	Dua arah, tanpa *causal mask*	Kiri ke kanan (*causal*)	Dua arah di *encoder*, *causal* di *decoder*
Tujuan pralatih	Menebak token yang ditutupi (*masked language modeling*)	Menebak token berikutnya	Merekonstruksi atau mengubah teks
Cocok untuk	*Classification*, NER, *embedding* untuk pencarian	Pembangkitan teks, *chatbot*, LLM	Terjemahan, peringkasan

Untuk bahasa Indonesia tersedia IndoBERT beserta tolok ukur IndoNLU. *Decoder-only* menjadi dasar hampir semua LLM saat ini, dan itulah bahan Pertemuan 15.

Memilih yang tepat guna: *classification* keluhan nasabah

Sebuah bank digital ingin mengelompokkan keluhan nasabah (transfer gagal, kartu terblokir, penipuan, lainnya) lalu meneruskannya ke tim yang tepat.

Model paling tepat guna di sini adalah *encoder-only* berbahasa Indonesia yang di-*fine-tune*. Kategorinya tetap, sehingga LLM raksasa tidak diperlukan: model ratusan juta parameter cukup, dapat dijalankan di CPU setelah diekspor ke ONNX dan dikuantisasi, dan jauh lebih murah per permintaan.



Data nasabah

Teks keluhan memuat nama, nomor rekening, dan telepon. Anonimisasi sebelum data masuk jalur *training*, batasi akses, dan pastikan dasar pemrosesannya sesuai UU Pelindungan Data Pribadi.

Praktikum, rekap, dan tugas

Praktikum: menelusuri implementasi *self-attention*

Kerjakan dalam satu *notebook*, langkah demi langkah:

1. Jalankan `self_attention` NumPy pada contoh tiga token. Cocokkan A, Z, dan A_{mask} dengan hitungan tangan di slide.
2. Bandingkan dengan `F.scaled_dot_product_attention`, dengan dan tanpa `is_causal=True`.
3. Hapus pembagian $\sqrt{d_k}$. Ulangi dengan $d_k = 64$ dan vektor acak; ukur seberapa sering satu *weight* melebihi $0{,}9$.
4. Tukar urutan baris X dan tunjukkan bahwa *output* hanya bertukar urutan. Tambahkan *positional encoding*, lalu ulangi.
5. Ukur waktu dan memori `nn.MultiheadAttention` untuk $n = 128, 256, 512, 1024$. Apakah pertumbuhannya mendekati kuadratik?

Catatan bacaan: *Attention Is All You Need*

Satu sampai dua halaman, berisi:

- Masalah yang ingin diselesaikan dan mengapa RNN tidak cukup
- Gagasan inti dalam kalimat Anda sendiri (Bagian 3.2)
- Arsitektur: blok, *multi-head*, *positional encoding* (Bagian 3.1–3.5)
- Bukti: Tabel 1 (kompleksitas) dan hasil terjemahan beserta *training cost*-nya
- Satu keterbatasan yang Anda lihat, dan satu pertanyaan untuk kelas

Dinilai bersama kuis singkat di awal Pertemuan 15.

Kuis singkat: periksa pemahaman Anda

1. Sebuah RNN skalar punya faktor *gradient* 0,8 per langkah. Berapa *gradient* yang tersisa setelah 10, 30, dan 100 langkah? $0{,}107$; $0{,}0012$; 2×10^{-10}
2. Ulangi contoh *dot-product attention* dengan $\mathbf{s} = (2, 0)$. Kata mana yang mendapat *weight* terbesar, dan mengapa?
3. Mengapa *mask* memakai $-\infty$ dan tidak memakai 0?
4. Mengapa *decoder-only* memerlukan *causal mask*, sedangkan *encoder-only* tidak? Apa akibatnya bila *mask* dilupakan saat *training*?
5. Berapa parameter `nn.MultiheadAttention(768, 12)`, termasuk bias?
6. Mengapa *transformer* tanpa *positional encoding* tidak dapat membedakan “anjing menggigit pria” dari “pria menggigit anjing”?

Petunjuk

Soal 2: hitung skor $\mathbf{s} \cdot \mathbf{h}_j$ lebih dahulu; dua kata akan seri. Soal 5: rumusnya $4d^2 + 4d$.

Rekap pertemuan ini

- RNN membawa *hidden state* dari langkah ke langkah; perkalian Jacobian berulang menimbulkan *vanishing* dan *exploding gradient*.
- LSTM memperpanjang ingatan lewat *cell state*, tetapi tetap sekuensial dan, dalam *encoder-decoder*, terhambat *bottleneck*.
- *Attention*: skor, *softmax*, *weighted average*.
- *Self-attention*: $\mathrm{softmax}(QK^\top/\sqrt{d_k})V$; *causal mask* melarang melihat masa depan.
- *Multi-head* menangkap beberapa jenis hubungan dengan *cost* hampir sama.
- *Positional encoding* menyuntikkan urutan: sinusoidal, *learned*, atau RoPE.
- Blok *transformer*: *attention* + FFN, masing-masing dengan residual dan LayerNorm.
- Paralel dan berjalur pendek, dengan harga $O(n^2)$ terhadap panjang urutan.
- Tiga keluarga: memahami (BERT), membangkitkan (GPT), mengubah (T5).

Tugas untuk pertemuan berikutnya

1. Baca Prince, *Understanding Deep Learning*, Bab 12, lalu makalah Vaswani dkk. (2017) secara utuh.
2. Selesaikan praktikum *self-attention* dan catatan bacaan makalah.
3. Latihan tangan: hitung ulang A dan Z contoh tiga token tanpa membagi $\sqrt{d_k}$, lalu hitung *positional encoding* untuk $pos = 3$, $d = 4$.
4. Tugas besar UAS: siapkan presentasi proyek kelompok untuk Pertemuan 15 sesuai format yang akan diumumkan di awal kelas.

Pertemuan depan

Tumpuk banyak blok *decoder*, latih dengan menebak token berikutnya pada teks berskala sangat besar, lalu selaraskan dengan preferensi manusia: itulah *large language model*. Kita membahas *tokenization*, *pretraining*, *fine-tuning*, RLHF, *prompting*, RAG, keterbatasan, dan etikanya, ditutup dengan presentasi proyek kelompok.

Terima kasih

Pertanyaan?