Sieci RNN/LSTM/GRU i Transformer

Od RNN do Transformera — ewolucja sieci sekwencyjnych

1.1 Sieci rekurencyjne (RNN)

Sieci rekurencyjne (**Recurrent Neural Networks, RNN**) to pierwsze podejście do modelowania sekwencji. Kluczowa idea: **stan ukryty h_t** przenosi informację z poprzednich kroków czasowych. W każdym kroku model oblicza:

```
h_t = tanh( W_h · h_{t-1} + W_x · x_t + b )
```

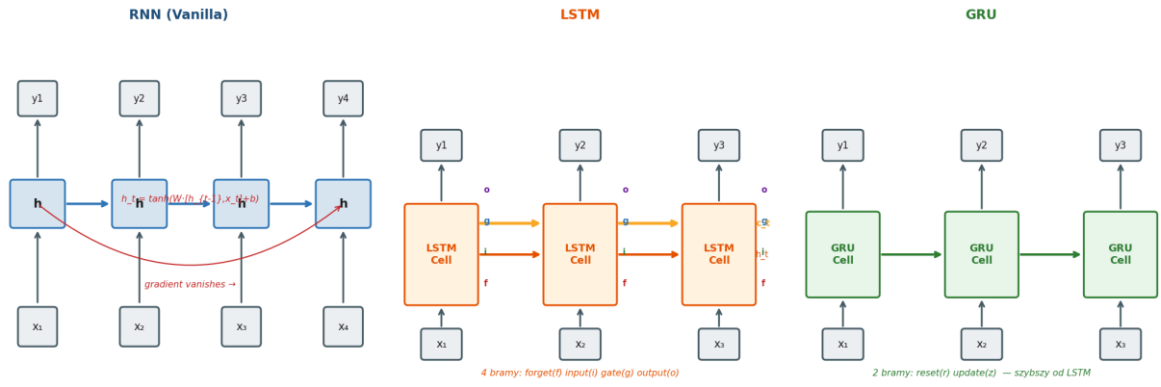
gdzie **h_{t-1}** to poprzedni stan, **x_t** to bieżące wejście, a **W_h, W_x** to macierze wag dzielone między wszystkimi krokami.

Kluczowy problem: znikający gradient (Vanishing Gradient)

Podczas propagacji wstecznej (backpropagation through time, BPTT) gradienty są mnożone przez tę samą macierz W_h w każdym kroku. Przy długich sekwencjach prowadzi to do:

- Znikania gradientu (|value| < 1): model traci pamięć dalekiej przeszłości
- Eksplozji gradientu (|value| > 1): niestabilne uczenie

Efekt praktyczny: RNN nie potrafi zapamiętać zależności odległych o więcej niż ~20 kroków.



Rysunek 1. Porównanie architektur RNN, LSTM i GRU. Strzałki czerwone symbolizują problem znikającego gradientu w vanilla RNN.

1.2 LSTM — Long Short-Term Memory

LSTM (Hochreiter & Schmidhuber, 1997) rozwiązuje problem znikającego gradientu przez wprowadzenie **pamięci komórkowej c_t** oraz czterech bramek (gates), które uczą się co zapamiętać, co zapomnieć i co przekazać:

Bramka	Symbol	Rola	Formuła (skrócona)
Forget Gate	f_t	Ile z c_{t-1} zachować?	σ(W_f·[h_{t-1},x_t]+b_f)
Input Gate	i_t	Co nowego dodać?	σ(W_i·[h_{t-1},x_t]+b_i)
Gate Gate	g_t	Kandydaci nowej pamięci	tanh(W_g·[h_{t-1},x_t]+b_g)
Output Gate	o_t	Co wypuścić jako h_t?	σ(W_o·[h_{t-1},x_t]+b_o)

Aktualizacja pamięci komórkowej:

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$
$$h_t = o_t \odot \tanh(c_t)$$

Symbol ⊙ oznacza mnożenie elementarne. Kluczowe: ścieżka c_t przechodzi **bez mnożenia przez macierze wag** — gradienty mogą przepływać wstecz bez zanikania (**highway for gradients**).

1.3 GRU — Gated Recurrent Unit

GRU (Cho et al., 2014) to uproszczona wersja LSTM — łączy bramki forget i input w jedną **bramkę aktualizacji z_t** oraz wprowadza **bramkę reset r_t**. Brak osobnej pamięci komórkowej:

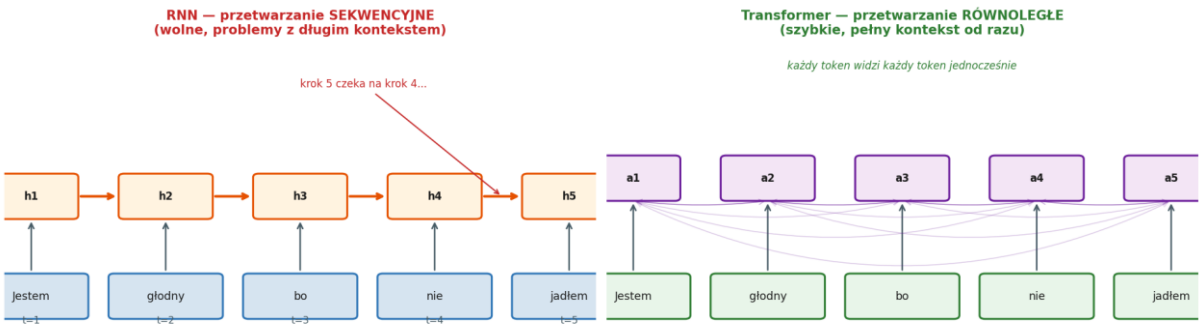
$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \quad \text{(update gate)}$$
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \quad \text{(reset gate)}$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t])$$

GRU ma **mniej parametrów niż LSTM** (2 bramki zamiast 4), przez co jest szybszy w treningu. W praktyce obie architektury osiągają porównywalną jakość.

1.4 Ograniczenia architektur rekurencyjnych

Dlaczego RNN/LSTM/GRU nie wystarczyły?

- **Sekwencyjność:** kroki muszą być przetwarzane jeden po drugim → niemożliwe pełne zrównoleglenie na GPU
- **Długi kontekst:** nawet LSTM traci ważne informacje przy setkach kroków
- **Wąskie gardło:** cały kontekst kompresowany do wektora h_t o stałym rozmiarze
- **Skalowanie:** trudne do efektywnego treningu na miliardach tokenów



Rysunek 2. RNN przetwarza tokeny sekwencyjnie (t=1,2,3,...), podczas gdy Transformer przetwarza wszystkie tokeny równolegle — każdy widzi każdy od razu.

Właśnie te ograniczenia stały się motywacją do opracowania Transformera. Zamiast **rekurencji** — zastosowano **mechanizm uwagi (attention)**, który pozwala każdemu tokenowi bezpośrednio "zapytać" o dowolny inny token w sekwencji, niezależnie od odległości.

2. Embedding — reprezentacja wektorowa tokenów

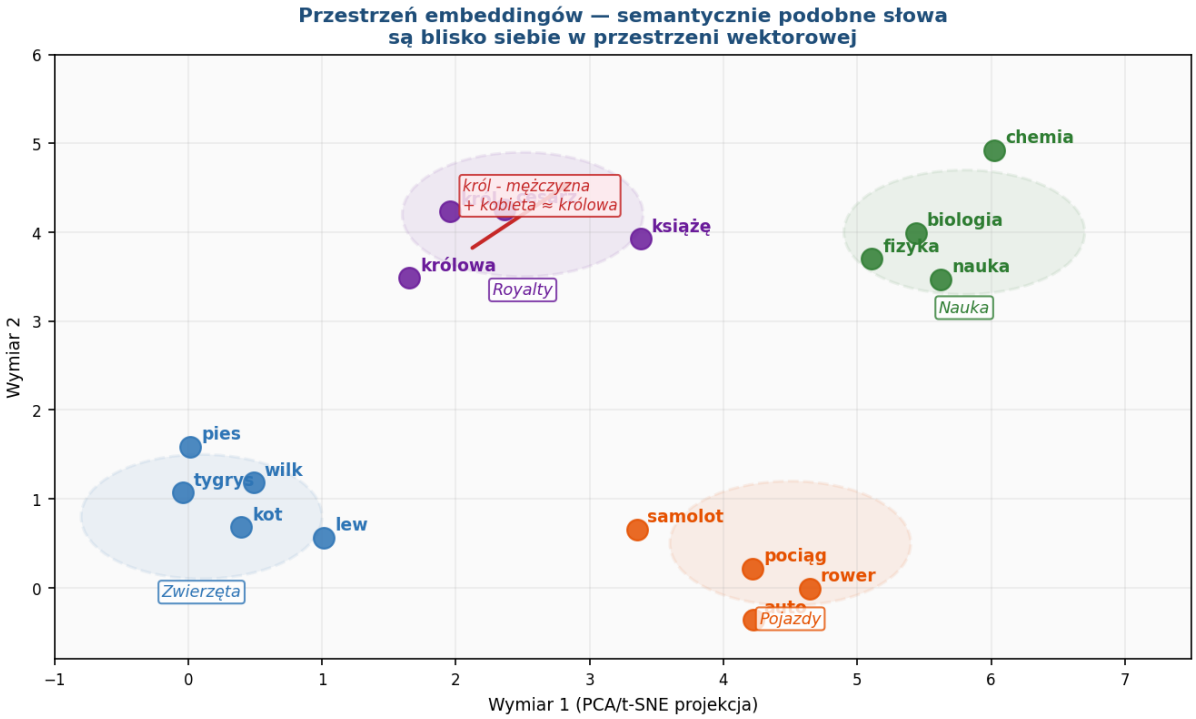
2.1 Czym jest Embedding?

Model językowy nie operuje na tekście bezpośrednio — operuje na **liczbach**. Embedding to funkcja mapująca dyskretny token (słowo, podslowo) na gęsty wektor liczb rzeczywistych w przestrzeni **R^d**, gdzie d to wymiarowość embeddingu (typowo d=512 lub d=768).

> **Formalnie:** `E: {token_1, ..., token_V} → R^d`
> gdzie V to rozmiar słownika (vocabulary), a d to liczba wymiarów. Tablica embeddingów to macierz `W_E ∈ R^(V×d)` — każdy wiersz to wektor jednego tokenu.

2.2 Właściwości przestrzeni embeddingów

Kluczową właściwością dobrze wytrenowanych embeddingów jest zachowanie **semantycznych relacji** w przestrzeni wektorowej:



Rysunek 3. Projekcja 2D przestrzeni embeddingów. Semantycznie podobne słowa tworzą klastry. Operacje arytmetyczne na wektorach zachowują semantykę: wektor('król') - wektor('mężczyzna') + wektor('kobieta') ≈ wektor('królowa').

Najważniejsze właściwości:

- Semantyczne podobieństwo: słowa o podobnym znaczeniu mają wysoką cosinusową miarę podobieństwa cos(e_1, e_2) = (e_1·e_2) / (|e_1| · |e_2|)
- Analogie wektorowe: relacje liniowe np. Paryż − Francja + Polska ≈ Warszawa
- Klasteryzacja: pojęcia z tej samej kategorii grupują się w spójnych regionach przestrzeni
- Gęste reprezentacje: przeciwieństwo one-hot encoding (V=50000, tylko 1 bit ustawiony) — embedding niesie informację w każdym wymiarze

2.3 Tokenizacja — skąd biorą się tokeny?

Zanim zastosujemy embedding, tekst jest dzielony na **tokeny** przez tokenizer. W modelach opartych na Transformerze powszechnie stosuje się **Byte Pair Encoding (BPE)** lub **WordPiece**. Tokeny to fragmenty słów, nie zawsze całe słowa:

Tekst wejściowy	Tokeny BPE	Liczba tokenów	Uwaga
"unhappiness"	['un','happin','ess']	3	prefiks + rdzeń + sufiks
"transformers"	['transform','ers']	2	typowe dla NLP
"AI"	['AI']	1	częste słowo = 1 token

"Rzeszów"	['Rz','esz','ów']	3	rzadkie słowo → więcej tokenów
"1+1=2"	['1','+','1','=','2']	5	cyfry i symbole osobno

2.4 Pre-trenowanie i fine-tuning embeddingów

W przypadku modeli takich jak BERT i GPT embeddingi są **trenowane end-to-end** razem z całą siecią. Możliwe podejścia:

Statyczne embeddingi

Word2Vec, GloVe, FastText

- Ten sam wektor dla danego słowa niezależnie od kontekstu
- 'bank' (finansowy) = 'bank' (rzeki) → ten sam wektor
- Szybkie, małe, gotowe do użycia
- Słabość: polisemia nierozwiązana

Kontekstualne embeddingi

BERT, GPT, ELMo

- Wektor tokenu zależy od CAŁEGO zdania
- 'bank' w różnych zdaniach → różne wektory
- Wymagają całego przejścia przez model
- Znacznie lepsza jakość dla zadań NLP

3. Positional Encoding — kodowanie pozycji

3.1 Problem braku informacji o pozycji

Architektura Transformera przetwarza wszystkie tokeny **równolegle**. Dla mechanizmu uwagi zdanie "Pies gryzie człowieka" i "Człowiek gryzie psa" dają te same embeddingi tokenów w losowej kolejności — **model nie wie, który token jest pierwszy**. Rekurencyjne sieci (RNN) miały pozycję "wbudowaną" w kolejność przetwarzania. Transformer potrzebuje jawnego kodowania pozycji.

Positional Encoding: x_i = e_i + PE_i

Rozwiązanie: do każdego wektora embeddingu e_i dodawany jest wektor pozycji PE_i, który jest unikalny dla każdej pozycji i niesie informację o miejscu tokenu w sekwencji.

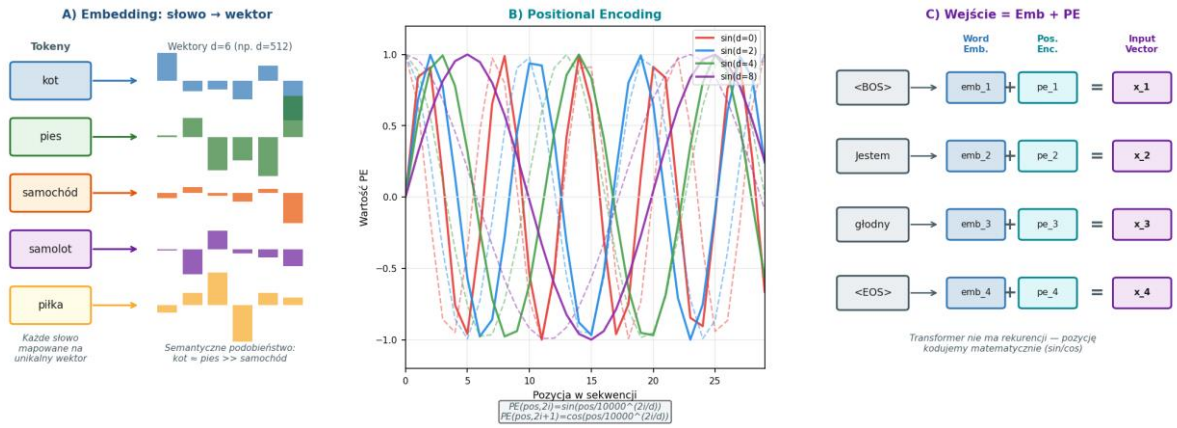
3.2 Formuła sinusoidalna (Vaswani et al. 2017)

Oryginalne kodowanie pozycji używa funkcji sinus i cosinus o różnych częstotliwościach:

$$PE(pos, 2i) = \sin(pos / 10000^{(2i/d)})$$

$$PE(pos, 2i+1) = \cos(pos / 10000^{(2i/d)})$$

gdzie **pos** to pozycja tokenu w sekwencji (0, 1, 2, ...), **i** to indeks wymiaru (0, 1, ..., d/2-1), a **d** to wymiarowość embeddingu.



Rysunek 4. A) Mapowanie tokenów na wektory embeddingów. B) Sinusoidalne fale Positional Encoding dla różnych wymiarów — niskie częstotliwości kodują globalną pozycję, wysokie — lokalną. C) Wejście do Transformera = Word Embedding + Positional Encoding.

3.3 Dlaczego funkcje trygonometryczne?

Wybór sin/cos nie jest przypadkowy — ta formuła spełnia kilka kluczowych wymagań:

Wymaganie	Jak spełnione?
Unikalność pozycji	Każda pozycja ma unikalny wektor d-wymiarowy
Generalizacja	Model może interpolować dla pozycji > max treningu
Odległości liniowe	PE(pos+k) daje się wyrazić jako liniowa transformacja PE(pos)
Ograniczony zakres	Wartości w [-1,+1], nie rosną nieograniczenie
Invariancja długości	Działa dla sekwencji dowolnej długości

3.4 Learned Positional Encoding

Alternatywnie, pozycje mogą być **uczone (learned)** — każdej pozycji odpowiada trenowany wektor. Stosowane w GPT-2/3 i BERT. Ograniczenie: model nie generalizuje poza rozmiar sekwencji z treningu.

Nowoczesne rozwiązania (RoPE, ALiBi, YaRN) wprowadzają zaawansowane metody kodowania pozycji, pozwalające na **rozszerzenie okna kontekstu** poza rozmiar z treningu bez utraty jakości.

4. Attention — mechanizm uwagi

4.1 Intuicja: zapytanie o słownik

Mechanizm uwagi można rozumieć jako **"miękkie" wyszukiwanie w pamięci**. Mamy:

Symbol	Nazwa	Analogia	Wymiar
Q	Query (zapytanie)	Co szukam / o czym pytam?	d_q
K	Key (klucz)	Jaki temat reprezentuję?	d_k
V	Value (wartość)	Jaką informację przekazuję?	d_v

Każdy token generuje swoje Q, K, V przez **liniowe projekcje** macierzami W_Q, W_K, W_V. Mechanizm uwagi oblicza, jak bardzo każde "pytanie" (Query) **pasuje** do każdego "klucza" (Key), a następnie sumuje odpowiednie wartości (Value).

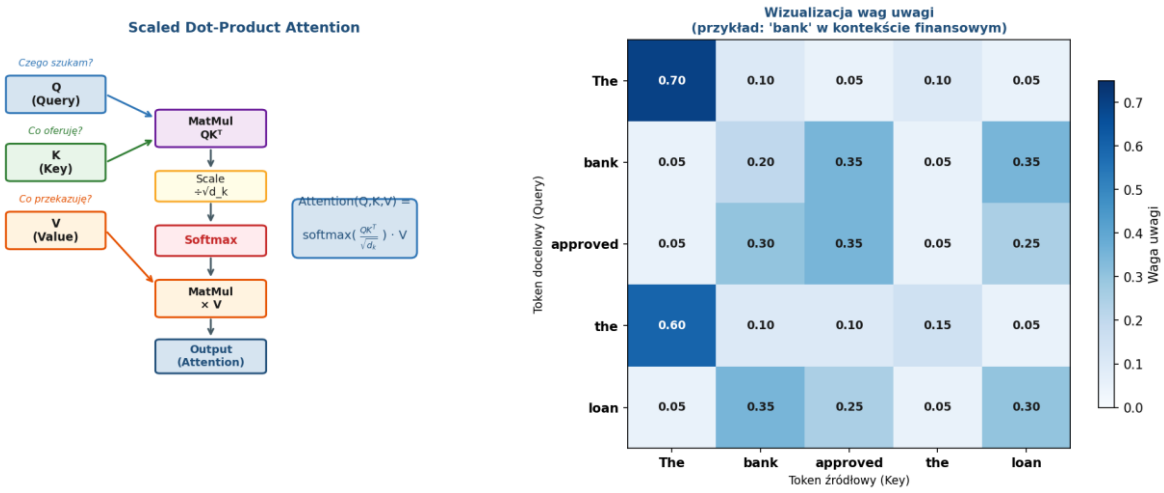
4.2 Scaled Dot-Product Attention

Formuła mechanizmu uwagi w zwartej postaci macierzowej:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(Q \cdot K^T / \sqrt{d\_k} \right) \cdot V$$

Kroki algorytmu:

- Krok 1 — Dot Product: oblicz iloczynY skalarne $Q \cdot K^T$ → macierz podobieństwa (n×n) dla sekwencji długości n
- Krok 2 — Scale: podziel przez √d_k, aby zapobiec eksplozji wartości dla dużych d_k (softmax stałby się nasycony)
- Krok 3 — Softmax: zamień wyniki na prawdopodobieństwa (sumy wierszy = 1) → macierz wag uwagi
- Krok 4 — Weighted sum: pomnóż wagi przez V → każdy token otrzymuje ważoną sumę wartości



Rysunek 5. Lewy panel: graf obliczeniowy Scaled Dot-Product Attention. Prawy panel: macierz wag uwagi dla zdania 'The bank approved the loan' — słowo 'bank' silnie uwzględnia 'approved' i 'loan', co wskazuje na finansowy kontekst.

4.3 Masking — maskowanie w dekoderze

W **dekoderze** (tryb generowania) token na pozycji t **nie może** widzieć tokenów na pozycjach t+1, t+2, ... (bo jeszcze nie zostały wygenerowane). Stosuje się **masking** — przed softmax dodaje się -∞ do zabronionych pozycji, co po softmax daje wagę = 0:

```
score(i,j)  =  -∞   jeśli j > i  (pozycja j jest w przyszłości)
```

Self-Attention w **enkoderze** nie używa maskowania — każdy token może uwzględniać cały kontekst (lewy i prawy).

4.4 Self-Attention vs Cross-Attention

Self-Attention (enkoder/dekoder)	Cross-Attention (dekoder ↔ enkoder)
• Q, K, V pochodzą z TEJ SAMEJ sekwencji • Token 'widzi' inne tokeny w tym samym zdaniu • Modeluje wewnętrzne zależności w sekwencji • Przykład: 'bank' odnosi się do 'kredyt'	• Q pochodzi z dekodera, K i V z enkodera • Token dekodera 'pyta' o tokeny wejścia • Implementuje tłumaczenie i kondycjonowanie • Przykład: generując 'credito' pytamy o 'credit'

5. Multi-Head Attention — wielogłowowy mechanizm uwagi

5.1 Motywacja: jedna głowica nie wystarczy

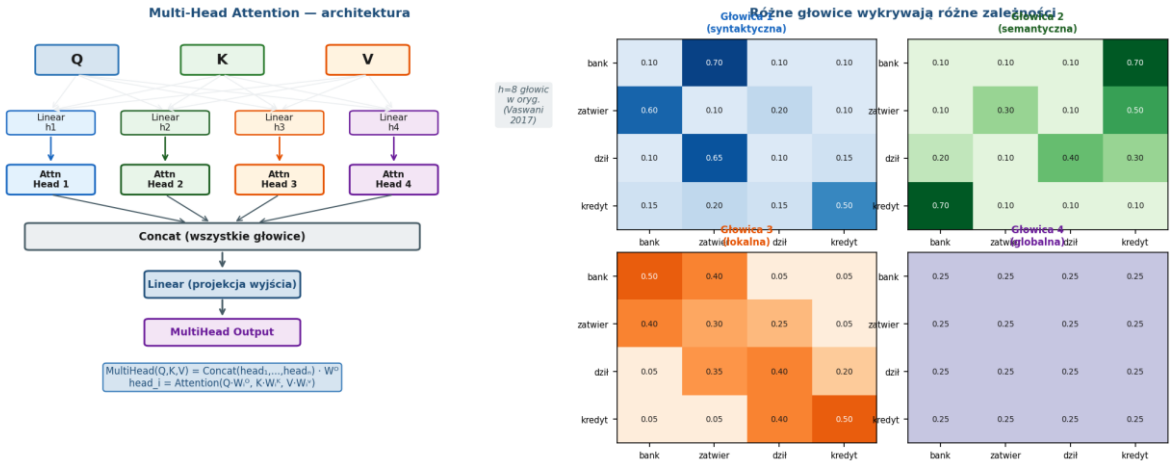
Pojedynczy mechanizm uwagi może skupić się na jednej relacji jednocześnie. Ale języki są wielowymiarowe — to samo zdanie zawiera jednocześnie relacje **syntaktyczne, semantyczne, referencyjne i lokalne**. Multi-Head Attention uruchamia **h równoległych głów uwagi**, każda z **własnymi, uczonymi projekcjami**, pozwalając modelowi wychwycić wiele rodzajów zależności jednocześnie.

5.2 Architektura i formuła

$$\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) \cdot W_O$$
$$\text{head}_i = \text{Attention}(Q \cdot W_i^Q,\ K \cdot W_i^K,\ V \cdot W_i^V)$$

Każda głowa i ma **osobne macierze projekcji** $W_i^Q \in R^{d \times d_k}$, $W_i^K \in R^{d \times d_k}$, $W_i^V \in R^{d \times d_v}$. Następnie wyniki wszystkich głowy są **konkatenowane** i rzutowane macierzą $W_O \in R^{hd_v \times d}$.

W oryginalnym Transformerze (Vaswani 2017): **h=8 głów**, d=512, stąd $d_k = d_v = d/h = 64$.



Rysunek 6. Lewy panel: pełna architektura Multi-Head Attention — linearne projekcje Q,K,V dla każdej głowy, równoległe obliczenia uwagi, konkatenacja i końcowa projekcja. Prawy panel: 4 przykładowe głowy wykrywają różne typy zależności w tym samym zdaniu.

5.3 Co wychwytują poszczególne głowy ?

Badania (Voita et al., 2019; Tenney et al., 2019) pokazały, że głowy specjalizują się w różnych typach relacji:

Typ głowicy	Przykładowe zachowanie	Użyteczność
Syntaktyczna	Podmiot → orzeczenie, przymiotnik → rzeczownik	Parsowanie zdań
Semantyczna	Sinonim, hiponim, korefencja ('bank' → 'kredyt')	Rozumienie znaczenia
Lokalna/pozycyjna	Token uwzględnia bezpośrednich sąsiadów	Wykrywanie fraz
Globalna	Równomierne wagi → identyfikacja kluczowych tokenów	Streszczanie kontekstu
Refencyjna	'on', 'ona' → konkretna postać w tekście	Rozwiązywanie zaimków

5.4 Złożoność obliczeniowa

Główna słabość mechanizmu uwagi: złożoność **$O(n^2 \cdot d)$** gdzie n to długość sekwencji. Dla n=1000 tokenów, każdy z każdym = milion par. Dlatego przetwarzanie bardzo długich sekwencji (np. n=100,000) jest kosztowne.

Długość n	Pary tokenów	Pamięć (approx.)	Typ modeli
512	262 144	~1 GB	BERT-base
2 048	4 194 304	~4 GB	GPT-3
8 192	67 108 864	~16 GB	GPT-4 (approx.)
128 000	16.4 miliarda	>>100 GB	wymaga Flash Attention

Nowoczesne rozwiązania jak **Flash Attention** (Dao et al., 2022) i **Sparse Attention** redukują złożoność do $O(n \cdot \sqrt{n})$ lub $O(n \cdot \log n)$.

6. Pełna architektura Transformer

6.1 Encoder-Decoder — przegląd

Oryginalny Transformer ("Attention Is All You Need", Vaswani et al., 2017) to architektura **Encoder-Decoder** zaprojektowana do tłumaczenia maszynowego. Enkoder przetwarza sekwencję wejściową i buduje jej reprezentację; dekoder generuje sekwencję wyjściową token po tokenie.

Rysunek 7. Pełna architektura Transformera Encoder-Decoder. Enkoder (niebieski) buduje reprezentacje wszystkich tokenów wejściowych. Dekoder (pomarańczowy) generuje tokeny wyjściowe, korzystając z reprezentacji enkodera przez Cross-Attention (fioletowe strzałki K,V).

6.2 Blok enkodera (jeden layer)

Komponent	Opis	Szczegóły
Multi-Head Self-Attention	Każdy token uwzględnia wszystkie inne tokeny wejścia	Q=K=V=token embeddingi
Add & Norm	Połączenie resztkowe + Layer Normalization	x = LayerNorm(x + sublayer(x))
Feed Forward Network	Dwie liniowe projekcje z ReLU pomiędzy	FFN(x) = max(0, $xW_1 + b_1$)$W_2 + b_2$
Add & Norm	Kolejne połączenie resztkowe i normalizacja	Stabilizuje trening głębokich sieci

Kluczowy element: **Residual Connections** (x + sublayer(x)) — wzorowane na ResNet. Umożliwiają trening bardzo głębokich sieci przez zapewnienie "autostrady" dla gradientu.

6.3 Blok dekodera (jeden layer)

Komponent	Opis	Szczegóły
Masked Multi-Head Self-Attention	Token widzi tylko poprzednie tokeny (nie przyszłe)	Maska trójkąt dolny
Add & Norm	Połączenie resztkowe + Layer Normalization	Jak w enkoderze
Cross-Attention	Q z dekodera, K+V z ENKODERA	Główna komunikacja enc→dec
Add & Norm	Kolejne połączenie resztkowe	Stabilizacja
Feed Forward Network	Identyczna z enkoderem	$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$
Add & Norm	Finalna normalizacja warstwy	

6.4 Warianty architektury Transformera

Na bazie oryginalnego Transformera rozwinęły się trzy główne warianty:

Wariant	Architektura	Przykłady	Zastosowanie
Encoder-only	Tylko enkoder, pełna uwaga (bez masek)	BERT, RoBERTa, DeBERTa	Klasyfikacja, NER, QA
Decoder-only	Tylko dekoder, maskowana uwaga	GPT-2/3/4, LLaMA, Mistral	Generowanie tekstu, chatboty
Encoder-Decoder	Oba moduły, Cross-Attention	T5, BART, FLAN-T5	Tłumaczenie, summaryzacja

6.5 Skala i przełom (Brief)

Model	Rok	Parametry	Kluczowa innowacja
Transformer (original)	2017	~65M	Mechanizm uwagi, Encoder-Decoder
BERT-large	2018	340M	Masked LM, dwukierunkowy enkoder
GPT-2	2019	1.5B	Autoreg. generowanie, zero-shot
GPT-3	2020	175B	Few-shot learning, emergent abilities
ChatGPT / GPT-4	2022-23	~1T?	RLHF, instruction tuning
LLaMA 3 / Mistral	2024	8B-70B	Efektywne otwarte modele

Podsumowanie:

- RNN: przetwarza sekwencyjnie, cierpi na znikający gradient, trudno zrównoleglic
- LSTM/GRU: bramki kontrolują przepływ informacji, lepsza pamięć długoterminowa
- Transformer: zastępuje rekurencję mechanizmem uwagi, przetwarza równolegle
- Embedding: tokeny → gęste wektory zachowujące semantyczne relacje
- Positional Encoding: dodaje informację o pozycji przez sin/cos lub uczenie
- Attention: każdy token 'pyta' każdy inny o dopasowanie — $O(n^2)$ złożoność
- Multi-Head Attention: wiele równoległych perspektyw uwagi, wychwytuje różne relacje
- Wynik: BERT, GPT i ich następcy zrewolucjonizowali NLP i AI