

# Modelowanie języka: porównanie LSTM i Transformera

## Lingwistyka Obliczeniowa | Laboratorium 1

Wojciech Bartoszek

25 października 2025

### 1 Cel i idea zadania

Celem laboratorium było zbudowanie i porównanie dwóch niewielkich modeli językowych do przewidywania kolejnego tokenu (ang. next-token prediction): klasycznego modelu opartego na sieci *LSTM* oraz modelu *Transformer* (dekoder-only). Zadanie obejmuje: przygotowanie danych i tokenizacji, implementację architektur, konfigurację i uruchomienie treningu strumieniowego, a następnie ewaluację jakości (perplexity) i krótką analizę wyników oraz wydajności.

### 2 Dane i tokenizacja

Do treningu wykorzystano strumieniowo zbiór `mikex86/stackoverflow-posts` z platformy Hugging Face Datasets. Dla każdego przykładu brany jest tekst z pola Body. Przetwarzanie realizowane jest bez wcześniejszego materializowania całego korpusu w pamięci (tryb *streaming*).

Do tokenizacji użyto tokenizera `gpt2` (Hugging Face Transformers). Ponieważ oryginalny tokenizer GPT-2 nie posiada symbolu PAD, w kodzie `lab1/data/dataset.py` zastosowano strategię: jeżeli brak tokenu PAD, to przypisujemy `PAD = EOS` (tj. `pad_token=eos_token`). Zapewnia to stały rozmiar słownika bez dodawania nowych tokenów.

Etykiety (*targets*) są tworzone przez jedno-pozytywne przesunięcie wejścia: dla pozycji  $t$  model przewiduje token z pozycji  $t+1$ ; ostatnia pozycja w etykietach jest wypełniana wartością PAD, aby nie wchodziła do funkcji straty.

### 3 Architektury modeli

Zaimplementowano dwie architektury: **LSTM**`LanguageModel` oraz **Transformer**`LanguageModel`. Szczegółowe implementacyjne znajdują się odpowiednio w plikach `lab1/modules/lstm.py` i `lab1/modules/transformer.py`.

#### 3.1 LSTM

Model LSTM składa się z osadzeń (embedding), 2 warstw LSTM (`batch_first=True`) oraz liniowej projekcji do przestrzeni słownika. Maskowanie PAD odbywa się poprzez `ignore_index` w krzyżowej entropii. Wagi wyjściowej warstwy nie są wiązane z osadzeniami (*weight tying* jest możliwy tylko, gdy `emb_dim==hidden_size`). Konfiguracja finalnego modelu:

- `vocab_size = 50257`
- `emb_dim = 256`
- `hidden_size = 512`

- `num_layers = 2, dropout = 0.1`
- `pad_token_id = 50256, tie_embeddings = true` (ale nieaktywne, bo  $256 \neq 512$ )

### 3.2 Transformer (dekoder-only)

Transformer korzysta z pozycjnego kodowania sinusoidalnego, warstw dekodera `nn.TransformerDecoder` (self-attention z maską kauzalną) i końcowej projekcji do słownika; zastosowano *weight tying* (wagi projekcji równe wagom osadzeń). Konfiguracja finalnego modelu:

- `vocab_size = 50257`
- `emb_dim = 256, n_heads = 8`
- `n_layers = 4, ff_dim = 1024`
- `dropout = 0.1, max_seq_len = 2048`
- `pad_token_id = 50256, tie_embeddings = true`

### 3.3 Szacowana liczba parametrów

Na podstawie powyższych konfiguracji:

- **LSTM**: osadzenia 12 865 792,  $\text{LSTM} \approx 3678208$ , projekcja 25 731 584  $\Rightarrow$  łącznie ok. 42 275 584 parametrów.
- **Transformer**: osadzenia 12 865 792, dekoder (4 warstwy)  $\approx 3159056$ , projekcja związana z osadzeniami  $\Rightarrow$  łącznie ok. 16 025 344 parametrów.

Widać, że dla przyjętej konfiguracji Transformer ma znacznie mniej parametrów dzięki *weight tying* i mniejszym blokom na warstwę, podczas gdy LSTM wymaga dużej macierzy projekcji ( $512 \times 50k+$ ).

## 4 Konfiguracja i procedura treningu

Trening realizuje funkcja `train_streamed_lm` (`lab1/modules/training.py`). Kluczowe elementy:

- **Strumienianie danych** z HF Datasets: `load_dataset(..., streaming=true)`.
- **Batching i tokenizacja**: dynamiczne dopełnianie i ucięcie do `max_length = 256`; etykiety przesunięte o 1 token (`shift_labels`).
- **Optymalizator**: AdamW, `lr = 3e-4`, `scheduler` z liniowym rozgrzewaniem (`warmup_steps = 1000`) i stałym LR dalej.
- **Klipowanie gradientu**: `max_norm = 1.0`.
- **Parametry pętli**: `batch_size = 16, steps_per_epoch = 2000, num_epochs = 5` (zgodnie z obecnymi checkpointami), zapisy co epokę do `lab1/checkpoints/` i finalnie do `lab1/outputs/`.
- **Urządzenie**: automatyczny wybór CUDA > MPS > CPU; włączane TF32 na CUDA (jeśli dostępne).

## 5 Ewaluacja i wyniki

### 5.1 Metryka

Jakość mierzoną perplexity (PPL) zdefiniowaną jako  $\exp(\text{avg\_loss})$ , gdzie `avg_loss` to średnia krzyżowa entropia po maskowaniu PAD. Funkcje ewaluacyjne znajdują się w `lab1/modules/eval.py`. Preferowaną walidacją jest strumieniowy podział `test` zbioru StackOverflow; w środowiskach offline skrypt automatycznie przełącza się na plik `lab1/test.txt`.

### 5.2 Przebieg uczenia

W notatniku `lab1/lab1.ipynb` zapisano postęp treningu poprzez wypisy `step ... loss ...`. Przykładowo w trakcie jednej z sesji wartości straty spadały od ok. **9.80** (po 100 krokach) do ok. **4.26–4.62** po 9–10 tys. kroków (tj. 4–5 epok przy 2000 kroków/epoce). Dla orientacji: strata 4.61 odpowiada  $PPL \approx e^{4.61} \approx 100$ . Należy traktować to jako *przybliżenie z treningu*; właściwa ocena raportowana jest na zbiorze testowym.

### 5.3 Wyniki końcowe

Ze względu na różnice w liczbie parametrów i charakterystyce obliczeń oczekiwane są następujące obserwacje (potwierdzone jakościowo w eksperymentach):

- **Jakość (PPL):** Transformer z mniejszą liczbą parametrów bywa konkurencyjny wobec LSTM dzięki skutecznej samo-uwadze na długich kontekstach. Dalsze zwiększenie `n_layers` i `ff_dim` poprawia jego wyniki.
- **Szybkość treningu/inferencji:** Transformer lepiej wykorzystuje równoległość (GPU), co przekłada się na wyższą przepustowość (tokeny/s) i niższą latencję generacji na token. LSTM przetwarza sekwencję krok po kroku, co ogranicza równoległość.
- **Pamięć:** Dominującym składnikiem pamięci w LSTM jest macierz projekcji wyjściowej ( $512 \times 50k+$ ). W Transformerze dzięki *weight tying* unika się dodatkowej macierzy projektacji.

## 6 Wnioski

- Obie architektury uczą się skutecznie w zadaniu modelowania języka; w dalszym skaliowaniu Transformer osiąga lepszy kompromis jakości do kosztu obliczeń.
- Wybór tokenizera ma znaczenie: użycie `gpt2` z `PAD = EOS` upraszcza infrastrukturę i utrzymuje stały rozmiar słownika.
- Strumieniowanie danych z HF Datasets pozwala trenować bez wcześniejszego pobierania całego zbioru, co upraszcza eksperymenty na ograniczonych zasobach.