



Politechnika Warszawska

**Wydział Matematyki
i Nauk Informatycznych**

Praca dyplomowa inżynierska

na kierunku Inżynieria i Analiza Danych

Implementacja transformera od podstaw oraz eksperymenty z
wariantami mechanizmu uwagi (Performer, Reformer) w zadaniach
klasyfikacji tekstu

Bartłomiej Borycki

Numer albumu 327265

Michał Iwaniuk

Numer albumu 327278

promotor

Dr Robert Małysz

WARSZAWA 2026

Streszczenie

Implementacja transformera od podstaw oraz eksperymenty z wariantami mechanizmu uwagi (Performer, Reformer) w zadaniach klasyfikacji tekstu

Abstract

Słowa kluczowe: słowo klucz 1, słowo klucz 2, ...

Abstract

Transformer Implementation from Scratch and Experiments with Attention Variants
(Performer, Reformer) for Text Classification Tasks

Abstract

Keywords: keyword1, keyword2, ...

Spis treści

1. Wstęp	11
1.1. Opis problemu i motywacja	11
1.2. Inne sposoby przetwarzania języka naturalnego	12
1.3. Wizja systemu	13
1.3.1. Model	13
1.3.2. System treningowy	14
1.3.3. Dane i tokenizacja	14
1.4. Cel biznesowy	15
1.5. Wymagania funkcjonalne	15
1.5.1. WF-1 — Pretrening i dostrajanie	15
1.5.2. WF-2 — Wymienne mechanizmy uwagi	16
1.5.3. WF-4 — Pipeline danych	17
1.5.4. WF-5 — Konfiguracja	17
1.6. Wymagania niefunkcjonalne	18
1.6.1. WNF-1 — Wydajność i efektywność zasobowa	18
1.6.2. WNF-2 — Jakość, niezawodność i testowalność	19
1.6.3. WNF-3 — Użyteczność i utrzymanie	19
1.6.4. WNF-4 — Przenośność i kompatybilność	20
1.6.5. WNF-5 — Monitorowanie i obserwowalność	20
1.7. Analiza ryzyka	20
1.8. Podział pracy	21
2. Architektura systemu	23
2.1. Architektura transformera	24
2.1.1. Klasa transformera	25
2.1.2. Mechanizmy uwagi	26
2.1.3. Blok Enkodera i MLP	28
2.1.4. Kodowanie pozycyjne	29

2.1.5.	Warstwa agregacji (ang. <i>Pooling</i>)	31
2.1.6.	Główce zadaniowe	31
2.2.	Tokenizer	32
2.2.1.	Trening i inicjalizacja	32
2.2.2.	Przetwarzanie danych (Encoding)	33
2.2.3.	Maskowanie dla MLM	33
2.2.4.	Dostępne metody i ich argumenty	34
2.3.	Trening	35
2.3.1.	Punkt wejścia i inicjalizacja środowiska	35
2.3.2.	Logika pętli treningowej (TrainingLoop)	36
2.3.3.	Dynamiczna optymalizacja wsadów	36
2.3.4.	Zarządzanie stanem (ang. <i>Checkpointing</i>)	37
2.4.	Logowanie przebiegu treningu	37
2.4.1.	Integracja z <i>Weights & Biases</i>	37
2.4.2.	Lokalny zapis danych (CSV)	37
2.4.3.	Ewaluacja i metryki	38
2.5.	Konfiguracja eksperymentów	38
2.5.1.	Inicjalizacja pretreningu	38
2.5.2.	Inicjalizacja dostrajania	39
2.6.	Instrukcja instalacji i ocena narzędzi	44
2.6.1.	Wymagania systemowe	44
2.6.2.	Instalacja środowiska	44
2.6.3.	Ocena narzędzi	44
2.7.	Podręcznik użytkownika	45
2.7.1.	Przechowywanie danych	46
2.7.2.	Tokenizacja danych	46
2.7.3.	Konfiguracja eksperymentów	47
2.7.4.	Trening	49
3.	Opis i implementacja mechanizmów uwagi	50
3.1.	SDPA	50
3.1.1.	Samouwaga (ang. <i>Self Attention</i>)	50
3.1.2.	Wielogłowowa samouwaga (ang. <i>Multihead Self Attention</i>)	51
3.1.3.	Flash Attention	51

3.2.	LSH	52
3.2.1.	Konstrukcja LSH	53
3.2.2.	Wielorundowa uwaga LSH (ang. <i>Multi-round LSH Attention</i>)	54
3.2.3.	Maska uwagi w bloku	55
3.3.	FAVOR+ Attention	56
3.3.1.	Wstęp i motywacja	56
3.3.2.	Uwaga jako kernel i mapowanie cech losowych	56
3.3.3.	Implementacja	58
4.	Eksperymenty	61
4.1.	Dane	61
4.1.1.	Zbiory danych	61
4.1.2.	Przetwarzanie danych	61
4.1.3.	Aspekty danych: źródła, licencje, etyka i bias	62
4.1.4.	EDA	63
4.2.	Opis eksperymentów	64
4.2.1.	Zarys ogólny eksperymentów	64
4.2.2.	Architektura	64
4.2.3.	Hiperparametry	65
4.2.4.	Wyniki modelu bazowego (ang. <i>baseline</i>)	67
4.3.	Eksperyment 1	68
4.3.1.	Opis eksperymentu	68
4.3.2.	Wyniki eksperymentu	68
4.4.	Eksperyment 2: Ewaluacja uwagi przybliżonych (LSH i FAVOR)	69
4.4.1.	Opis eksperymentu	69
4.4.2.	Wyniki eksperymentu	70
4.5.	Eksperyment 3	70
4.5.1.	Opis eksperymentu	70
4.5.2.	Wyniki eksperymentu	71
5.	Analiza działania systemu	72
5.1.	Testy jednostkowe	72
5.2.	Testy akceptacyjne	74
6.	Podsumowanie	77

1. Wstęp

1.1. Opis problemu i motywacja

Transformery stały się dominującą architekturą w przetwarzaniu języka naturalnego (ang. *Natural Language Processing (NLP)*). Od czasu publikacji artykułu *Attention is all you need* [14] w 2017 roku, modele oparte na mechanizmie uwagi zastąpiły wcześniejsze podejścia rekurencyjne i stanowią fundament współczesnych systemów przetwarzania języka naturalnego. Standardowy mechanizm liczenia uwagi (*scaled dot-product attention*) ma złożoność obliczeniową i pamięciową $O(N^2)$, gdzie N to długość sekwencji wejściowej. W praktyce oznacza to ograniczenie długości kontekstu – oryginalny model BERT [5] operuje na sekwencjach do 512 tokenów – oraz wysokie wymagania sprzętowe. Dla wielu zastosowań (np. analiza dokumentów prawnych, artykułów naukowych, długich rozmów) limit 512 tokenów jest niewystarczający. Jednocześnie nie każda organizacja dysponuje zasobami obliczeniowymi pozwalającymi na trening i uruchamianie dużych modeli. Wraz z rosnącą popularnością transformerów pojawiło się wiele propozycji modyfikacji oryginalnego mechanizmu uwagi. Wśród nich – Performer [3], Reformer [7], na których skupimy się w naszej pracy – próbują rozwiązać problem kwadratowej złożoności obliczeniowej standardowej uwagi. Każda z metod wprowadza inne kompromisy między efektywnością obliczeniową a jakością reprezentacji. Porównywanie tych mechanizmów w praktyce jest trudne. Publikacje naukowe często używają różnych architektur bazowych, różnych zbiorów danych, różnych procedur treningowych. Utrudnia to obiektywną ocenę, nie wiadomo, czy różnice w wynikach wynikają z samego mechanizmu uwagi, czy z innych czynników. Celem pracy jest opracowanie modelu typu *encoder-only* transformer (podobnego do BERT), który przy niezmienniej architekturze bazowej umożliwia wymianę modułu uwagi i tym samym rzetelne porównanie różnych mechanizmów uwagi.

1.2. Inne sposoby przetwarzania języka naturalnego

Modele rekurencyjne

Przed transformerami w NLP dominowały architektury rekurencyjne:

RNN — przetwarzają sekwencję token po tokenie, przekazując ukryty stan między krokami. Problemem są trudności z uczeniem długoterminowych zależności (zanikający/eksplodujący gradient), brak możliwości równoległego przetwarzania.

LSTM i GRU — rozszerzenia RNN z mechanizmami, które łagodzą problem zanikającego gradientu. Pozwalają modelować dłuższe zależności, ale nadal przetwarzają sekwencję sekwencyjnie.

Modele oparte na transformerze

Transformer (2017) [14] — całkowite odejście od rekurencji. Mechanizm uwagi pozwala każdemu tokenowi patrzeć na wszystkie inne tokeny jednocześnie. Umożliwił pełne zrównoleglenie obliczeń na GPU.

BERT (2019) [5] czyli transformer tylko z enkoderem (ang. *Encoder-only transformer*) – trenowany na dwóch zadaniach: Modelowanie języka z maskowaniem (ang. *Masked Language Modeling (MLM)*) oraz predykcja następnego zdania (ang. *Next Sentence Prediction (NSP)*) – stał się standardem dla zadań rozumienia języka.

RoBERTa (2019) [10] — zoptymalizowana wersja BERT z usuniętym zadaniem NSP, dłuższym treningiem i większymi batchami. Pokazuje, że procedura treningowa ma duży wpływ na jakość modelu.

Popularne modyfikacje mechanizmu uwagi i architektury

Sparse Transformer (2019) [2] – wykorzystuje fakt, że macierz uwagi jest często rzadka. Zamiast obliczać pełną uwagę $O(N^2)$, stosuje wzorce lokalne (ang. *sliding window*) i uwagę kroczącą (ang. *strided attention*), redukując złożoność do $O(N\sqrt{N})$.

BigBird (2020) [16] – rozszerza rzadką uwagę o tokeny globalne, które widzą całą sekwencję, oraz losowe połączenia między tokenami. Dzięki temu zachowuje właściwości uniwersalnego aproksymatora przy liniowej złożoności $O(N)$.

Reformer (2020) [7] – zastępuje uwagę *dot-product* mechanizmem *Locality-Sensitive Hashing* (LSH), grupując podobne wektory i ograniczając obliczenia do $O(N \log N)$. Wprowadza również odwracalne warstwy resztowe, co drastycznie zmniejsza zużycie pamięci.

Longformer (2020) [1] – łączy lokalną uwagę okienkową z uwagą globalną dla wybranych tokenów (np. [CLS]). Umożliwia to procesowanie bardzo długich dokumentów przy liniowej zależności kosztu od długości sekwencji $O(N)$.

Linformer (2020) [15] – opiera się na obserwacji, że macierz uwagi jest niskorzędowa (*low-rank*). Wykorzystuje projekcje liniowe do zrzutowania wymiaru sekwencji na mniejszy wymiar, aproksymując uwagę w czasie $O(N)$.

Performer (2021) [3] – wykorzystuje estymatory jądrowe i mechanizm FAVOR+ do aproksymacji mechanizmu uwagi Softmax. Pozwala to na obliczanie uwagi z liniową złożonością czasową i pamięciową $O(N)$.

cosFormer (2022) [11] – proponuje linearyzację uwagi poprzez zastąpienie funkcji Softmax funkcją bazującą na cosinusie i ReLU. Umożliwia to obliczenia w czasie liniowym $O(N)$ oraz wzmacnia lokalne korelacje.

FlashAttention (2022) [4] – algorytm optymalizujący wykorzystanie pamięci GPU (IO-aware) poprzez obliczanie uwagi blokami (*tiling*) mieszczącymi się w szybkiej pamięci SRAM oraz fuzję operacji. Metoda nie aproksymuje uwagi i zachowuje dokładny wynik, zamiast tego redukuje zużycie pamięci pośredniej (unikając materializacji macierzy uwagi $N \times N$ w HBM) oraz minimalizuje transfery HBM-SRAM, co przekłada się na znaczne przyspieszenie i umożliwia pracę z dłuższymi sekwencjami w praktyce.

Wraz z szybkim rozwojem modeli LLM opartych na transformerze pojawiają się także nowsze warianty uwagi, np. **Multi-Head Latent Attention** (2024) [8], w której pamięć podręczna KV jest kompresowana do współdzielonej reprezentacji latentnej (ang. *low-rank KV compression*). Na podobnych założeniach bazuje **DeepSeek Sparse Attention** (DSA) (2025) [9], gdzie dzięki mechanizmowi indeksowania (ang. *lightning indexer*) oraz selekcji *Top-k* dla każdego zapytania oblicza się tzw. *core attention* jedynie dla podzbioru najbardziej istotnych tokenów.

1.3. Wizja systemu

1.3.1. Model

- **Mechanizmy liczenia uwagi:** System implementuje trzy warianty liczenia uwagi:
 - Standardowy mechanizm uwagi – SDPA (*Scaled Dot-Product Attention*)
 - FAVOR+ (*Fast Attention Via positive Orthogonal Random features*) z architektury Performer,
 - LSH (*Locality Sensitive Hashing*) z architektury Reformer.

- **Rdzeń modelu:** Encoder typu Transformer, odpowiedzialny za tworzenie reprezentacji wektorowych tekstu.
- **Główce zadaniowe:**
 - **Głowica MLM:** Wykorzystywana podczas fazy pretreningu. Służy do przewidywania zamaskowanych tokenów na podstawie kontekstu.
 - **Głowica klasyfikacyjna:** Wykorzystywana podczas etapu dostrajania (ang. *finetuning*). Służy do przewidywania etykiety klasy dla zadanego tekstu wejściowego.

1.3.2. System treningowy

- **Tryby działania:**
 1. **Pretrening (MLM):** Model uczy się ogólnej reprezentacji języka na dużym korpusie tekstowym w sposób nienadzorowany.
 2. **Dostrajanie (klasyfikacja):** Dostrajanie wstępnie wytrenowanego modelu do realizacji zadania klasyfikacji nadzorowanej.
- **Zarządzanie konfiguracją:** Wszystkie hiperparametry modelu oraz ustawienia treningu definiowane są w plikach formatu `.yaml`.
- **Logowanie i śledzenie eksperymentów:**
 - Integracja z platformą *Weights & Biases* do wizualizacji i monitorowania metryk.
 - Lokalne logowanie metryk do plików formatu CSV.
- **Organizacja eksperymentów:** Każdy eksperyment posiada dedykowany katalog wyjściowy, zawierający:
 - plik konfiguracyjny,
 - zapisane stany modelu (ang. *checkpoints*),
 - metryki w formacie CSV,
 - metadane z artefaktów W&B.

1.3.3. Dane i tokenizacja

- **Przechowywanie danych:** Dane zapisywane są jako gotowe tensory PyTorch (pliki `.pt`), zawierające:
 - tokenizowany tekst (ID tokenów),

1.4. CEL BIZNESOWY

- maski uwagi,
 - etykiety (dla zadań klasyfikacji).
- **Tokenizacja:** Wykorzystanie algorytmu *WordPiece* (używany oryginalnie w BERT) oraz możliwość:
 - wykorzystania gotowego słownika z oryginalnego modelu BERT,
 - wytrenowania własnego słownika na własnym korpusie tekstowym.

1.4. Cel biznesowy

Celem biznesowym projektu jest opracowanie efektywnego kosztowo rozwiązania do klasyfikacji tekstu, które umożliwi wykorzystanie nowoczesnych metod przetwarzania języka naturalnego również przy ograniczonych zasobach obliczeniowych.

Rozwiązanie miałoby zastosowanie w zadaniach takich jak moderacja treści, analiza opinii klientów, automatyczna kategoryzacja dokumentów oraz innych zadaniach opartych na klasyfikacji tekstu.

Podstawową hipotezą biznesową projektu jest założenie, że poprzez przemyślane dobór alternatywnych mechanizmów uwagi oraz właściwą konfigurację architektury i parametrów treningowych można osiągnąć istotną redukcję kosztu rozwiązania przy jednoczesnym zachowaniu jakości predykcji wymaganej w zastosowaniach produkcyjnych. Całkowity koszt obejmuje zarówno bezpośrednie nakłady finansowe na infrastrukturę i energię elektryczną, jak również czas potrzebny na trening i inferencję.

1.5. Wymagania funkcjonalne

1.5.1. WF-1 — Pretrening i dostrajanie

Opis System umożliwia pełny cykl uczenia: pretrening z maskowanym modelowaniem języka (MLM), a następnie dostrajanie do klasyfikacji na danych docelowych z wykorzystaniem zapisanego stanu z pretreningu.

Wejście/Wyjście

- **Wejście:** ztokenizowane korpusy tekstowe, plik YAML z konfiguracją, tryb `pretrain` lub `finetune`.

- **Wyjście:** kompletne zapisane stany modelu w formacie `.pt` obejmujące stan modelu, optymalizatora (ang. *optimizer*), schedulera współczynnika uczenia (ang. *learning rate scheduler*) i skamera gradientu (ang. *gradient scaler*); metryki (CSV); artefakty i metadane z platformy W&B.

Kryteria akceptacji

1. Uruchomienie treningu w trybie **pretrain** z poprawną konfiguracją generuje zapisany stan modelu na koniec treningu oraz po każdej epoce zapisuje najlepszy dotąd model (na podstawie danych walidacyjnych), a także zapisuje metryki do CSV i W&B.
2. Możliwość wznowienia treningu w trybie **pretrain** i **finetune** z ostatniego zapisanego stanu bez utraty postępu.
3. Uruchomienie treningu w trybie **finetune** inicjalizuje głowicę klasyfikacyjną zgodnie (zastępując głowicę klasyfikacyjną).

1.5.2. WF-2 — Wymienne mechanizmy uwagi

Opis Każdy blok enkodera może używać jednego z mechanizmów: `scaled_dot_product`, `reformer_lsh`, `performer_favor`. Wybór odbywa się deklaratywnie w konfiguracji YAML dla całego modelu.

Wejście/Wyjście

- **Wejście:** rodzaj mechanizmu uwagi; parametry specyficzne (patrz tab. 2.3).
- **Wyjście:** logi czasu na krok i zużycia pamięci dla wybranego wariantu.

Kryteria akceptacji

1. Zmiana mechanizmu uwagi umożliwia trening i inferencję bez modyfikacji kodu bloku enkodera.
2. Maskowanie uwagi jest respektowane przez wszystkie trzy warianty uwagi.
3. Dla wejściowego tensora o wymiarze $[B, N, D]$ każdy wariant zwraca tensor o wymiarze $[B, N, D]$.

WF-3 — Obsługa długich sekwencji

Opis Model przyjmuje wejścia o dowolnej długości i wspiera schematy kodowania pozycyjnego (ang. *position encoding*): sinusoidalne, uczone (ang. *learned*) oraz RoPE [12].

Wejście/Wyjście

- **Wejście:** sekwencja tokenów; schemat kodowania pozycyjnego (sinusoidalne, uczone, RoPE).
- **Wyjście:** zastosowane kodowanie pozycyjne na wektorach osadzeń (ang. *embedding vector*).

Kryteria akceptacji

1. Poprawne działanie kodowania pozycyjnego zweryfikowane testami jednostkowymi.
2. Wszystkie rodzaje kodowania pozycyjnego działają ze wszystkimi mechanizmami uwagi.

1.5.3. WF-4 — Pipeline danych

Opis Dane są przetwarzane przez tokenizację *WordPiece* ze wsparciem tokenów specjalnych i dynamicznego przygotowania partii (przycinanie w `DataLoader`); implementacja MLM stosuje reguły BERT.

Wejście/Wyjście

- **Wejście:** surowe rekordy tekstowe, słownik *WordPiece*.
- **Wyjście:** tensory: identyfikatory tokenów, maska uwagi, etykiety (`input_ids`, `attention_mask`, `labels`).

Kryteria akceptacji

1. Przycinanie paddingu do najdłuższej sekwencji w partii redukuje średnie zużycie pamięci względem statycznego paddingu.
2. Maskowanie w zadaniu MLM jest zgodne z zasadą stosowaną w BERT: 15% tokenów jest wybieranych do zamaskowania, z czego 80% zastępuje się tokenem `[MASK]`, 10% losowym tokenem, a 10% pozostawia bez zmian.

1.5.4. WF-5 — Konfiguracja

Opis System wykorzystuje generator konfiguracji treningu, który z szablonu pliku konfiguracyjnego tworzy katalog `pretrain/<nazwa_eksperymentu>/` lub `finetune/<nazwa_eksperymentu>/` – w zależności od trybu treningu – z plikiem `config.yaml` gotowym do ewentualnych modyfikacji.

Wejście/Wyjście

- **Wejście (tryb pretrain):** nazwa eksperymentu `pretrain` (`<PRE_EXP>`).
- **Wejście (tryb finetune):** nazwa eksperymentu `finetune` (`<FT_EXP>`) oraz nazwa eksperymentu `pretrain` (`<PRE_EXP>`).
- **Wyjście (tryb pretrain):** katalog `pretrain/<PRE_EXP>/{config.yaml, checkpoints/, metrics/, wandb/ }`.
- **Wyjście (tryb finetune):** katalog `finetune/<FT_EXP>/{config.yaml, checkpoints/, metrics/, wandb/}`.

Kryteria akceptacji

1. Uruchomienie generatora dla trybu `pretrain` z nazwą eksperymentu `<PRE_EXP>`, tworzy odpowiedni katalog oraz plik konfiguracyjny na podstawie szablonu.
2. Uruchomienie generatora dla trybu `finetune` z nazwą eksperymentu `<FT_EXP>` oraz z nazwą eksperymentu `<PRE_EXP>` tworzy odpowiedni katalog, plik konfiguracyjny na podstawie szablonu oraz dziedziczy architekturę z `pretrain/<PRE_EXP>/config.yaml`.

1.6. Wymagania нефункционалне

1.6.1. WNF-1 — Wydajność i efektywność zasobowa

System powinien umożliwiać uruchomienie i trenowanie modeli w środowisku *Google Colab* przy zachowaniu niższych czasów treningu i zużycia pamięci dla alternatywnych wariantów uwagi względem klasycznego SDPA.

- **Środowisko GPU (Colab):** docelowo *A100 40 GB* lub *T4 16 GB*.
- **Wymóg kosztowy (Performer):** $time/epoch \leq (SDPA - 20\%)$ lub $\max VRAM \leq (SDPA - 30\%)$ przy spadku jakości ≤ 1 pp macro-F1.
- **Wymóg kosztowy (Reformer):** $time/epoch \leq (SDPA - 15\%)$ lub $\max VRAM \leq (SDPA - 25\%)$ przy spadku jakości ≤ 1 pp macro-F1.
- **Techniki optymalizacji:** Wykorzystywana jest mieszana precyzja pozwalająca wykonywać obliczenia w precyzji *FP16/BF16* oraz techniki optymalizacji pamięci, takie jak akumulacja gradientów.

1.6.2. WNF-2 — Jakość, niezawodność i testowalność

System powinien zapewniać stabilność i jakość procesu uczenia. Szczegóły dotyczące danych treningowych oraz wykonywanych eksperymentów znajdują się w sek. 4.

- **Jakość (SDPA):** $\text{macro-F1} \geq (\text{TF-IDF} + \text{LogReg} + 5 \text{ pp})$, mierzone na wszystkich docelowych zbiorach danych, parametry jak w konfiguracji.
- **Jakość (Performer):** $\text{macro-F1} \geq (\text{TF-IDF} + \text{LogReg} + 3 \text{ pp})$ oraz w odległości $\leq 1 \text{ pp}$ od SDPA przy tej samej konfiguracji – na wszystkich docelowych zbiorach danych.
- **Jakość (Reformer):** $\text{macro-F1} \geq (\text{TF-IDF} + \text{LogReg} + 3 \text{ pp})$ oraz w odległości $\leq 1 \text{ pp}$ od SDPA przy tej samej konfiguracji – na wszystkich docelowych zbiorach danych.
- **Stabilność:** brak NaN/OOM; 3 różne seedy; rozrzut macro-F1 (max-min) $\leq 1 \text{ pp}$. Testowane dla `seq_len=512`.
- **Testy komponentów:** Kluczowe komponenty (uwaga, maski, pozycjonowanie, tokenizacja) są objęte testami.

1.6.3. WNF-3 — Użyteczność i utrzymanie

System powinien być łatwy w konfiguracji, rozbudowie i ponownym uruchomieniu eksperymentów.

- **Dokumentacja:** Dokumentacja opisuje sposób przygotowania danych, pretreningu i dostrajania modelu krok po kroku.
- **Struktura katalogów:** Struktura katalogów i plików jest spójna i hierarchiczna.
- **Zgodność z PEP-8:** Kod jest zgodny ze standardem PEP 8, a kluczowe moduły są opatrzone docstringami.
- **Wersjonowanie:** Kod źródłowy jest wersjonowany w systemie kontroli wersji (Git).
- **Rozszerzalność mechanizmów uwagi:** Architektura systemu umożliwia dodawanie nowych wariantów mechanizmów uwagi bez ingerencji w istniejące komponenty.
- **Konfiguracja YAML:** Wszystkie parametry konfiguracyjne są definiowane w pliku YAML.

1.6.4. WNF-4 — Przenośność i kompatybilność

System powinien być zaprojektowany z myślą o łatwym przenoszeniu między różnymi środowiskami obliczeniowymi oraz zapewnieniu kompatybilności z nowoczesnymi wersjami bibliotek.

- **Kompatybilność Python:** Kod działa w środowiskach Python ≥ 3.12 oraz z frameworkiem PyTorch ≥ 2.2 (CUDA 11.8/12.x).
- **Biblioteki:** Wykorzystywane są standardowe biblioteki do uczenia maszynowego.

1.6.5. WNF-5 — Monitorowanie i obserwowalność

System powinien oferować rozbudowane mechanizmy monitorowania metryk oraz wersjonowania stanów modelu.

- **Monitorowanie W&B:** Wszystkie metryki (loss, accuracy, F1, zużycie pamięci GPU, czas/epoka) są logowane do *Weights & Biases*.
- **Monitorowanie CSV:** Możliwość zapisu wszystkich metryk do pliku CSV.
- **Wznowienia treningu:** System udostępnia możliwość wznowienia treningu z dowolnego zapisanego stanu.

1.7. Analiza ryzyka

Celem analizy jest szybkie wykrywanie i ograniczanie ryzyk poprzez jasne wskaźniki (triggery) oraz gotowe działania (mitigacje). Poniżej przedstawiamy listę kluczowych ryzyk, ich trigger, proponowane mitigacje oraz plany awaryjne. W tab. 1.1 przedstawiamy prawdopodobieństwo, wpływ oraz właścicieli poszczególnych ryzyk.

1. **Ograniczenia zasobów (VRAM).** *Trigger:* OOM przy $N \geq 16k$. *Mitigacje:* FP16/BF16, akumulacja gradientów, dynamiczny padding. *Plan awaryjny:* skrócenie N , redukcja warstw/głów.
2. **Niestabilność treningu (NaN, eksplodujące gradienty).** *Trigger:* NaN/Inf w stracie lub gradientach. *Mitigacje:* przycinanie gradientów, dłuższy warmup, inny współczynnik uczenia, skalowanie straty. *Plan awaryjny:* zmiana optymalizatora, hiperparametrów treningu.

Tabela 1.1: Analiza ryzyk projektu z przypisanymi właścicielami i oceną prawdopodobieństwa oraz wpływu

Ryzyko	Prawd.	Wpływ	Właściciel
OOM przy długich sekwencjach	Średnie	Wysoki	Bartłomiej Borycki
Niestabilność treningu	Średnie	Wysoki	Bartłomiej Borycki
Błędy w Reformer/Performer	Średnie	Średni	Michał Iwaniuk
Niedopasowanie tokenizera	Niskie	Średni	Michał Iwaniuk
Awaria sesji Colab / W&B	Niskie	Średni	Michał Iwaniuk

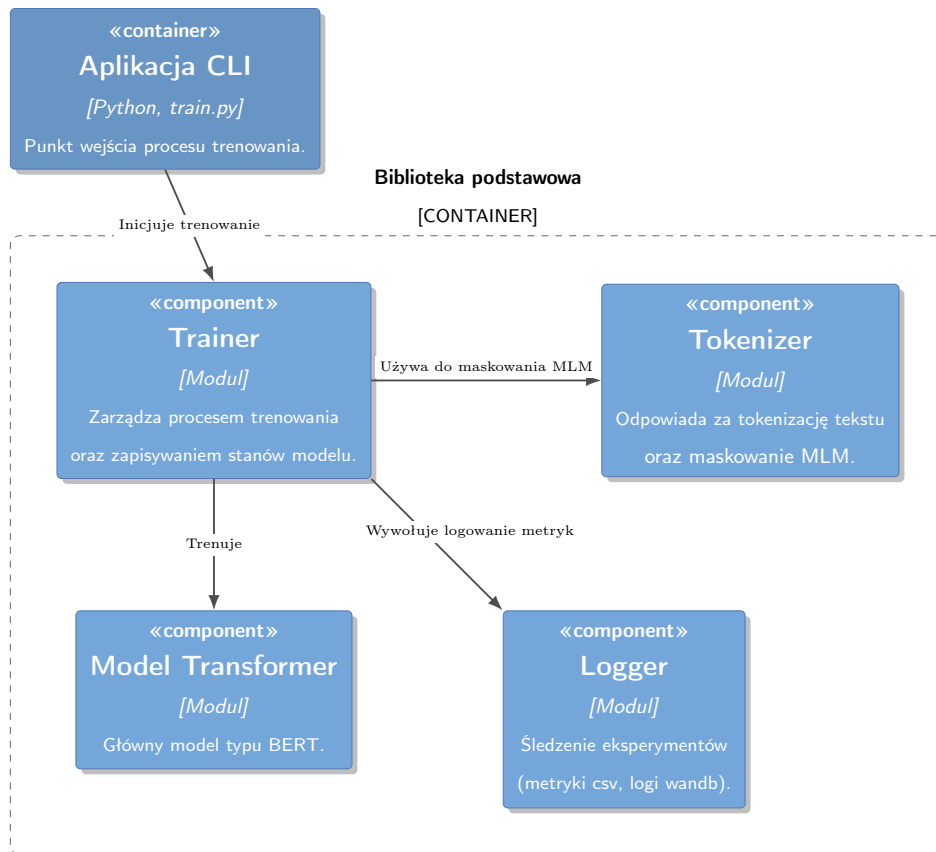
3. **Błędy implementacyjne (Reformer/Performer).** *Trigger:* niezgodność kształtów/masek, rozjazd wyników na danych syntetycznych. *Mitigacje:* testy jednostkowe i funkcjonalne, asercje. *Plan awaryjny:* Ponowna implementacja/naprawa wadliwego komponentu, po czym re-walidacja testami.
4. **Niedopasowanie tokenizera do domeny.** *Trigger:* wysoki udział tokenów [UNK]. *Mitigacje:* dostrojenie słownika (dołączenie domenowych subwordów). *Plan awaryjny:* pełny trening tokenizera z uwzględnieniem danych domenowych.
5. **Zależności zewnętrzne (Colab, W&B) i awarie sesji.** *Trigger:* przerwane sesje, brak artefaktów/logów. *Mitigacje:* zapisy lokalne, zapis stanu modelu co epokę. *Plan awaryjny:* uruchomienia lokalne i późniejsza synchronizacja z W&B.

1.8. Podział pracy

Podział prac jest przedstawiony w tab. 1.2.

Tabela 1.2: Podział prac i odpowiedzialności w projekcie.

Osoba odpowiedzialna	Zakres prac
Bartłomiej Borycki	Implementacja mechanizmu uwagi Performer FAVOR+, implementacja klasy transformera (z gotowego enkodera), implementacja skryptów treningowych i skryptów do generowania katalogów z eksperymentami.
Michał Iwaniuk	Implementacja mechanizmów uwagi SDPA i Reformer LSH, implementacja enkodera, testy jednostkowe.
Wspólne	Wybór zbiorów danych, preprocessing danych, trenowanie modeli, analiza i wizualizacja wyników, przygotowanie dokumentacji.

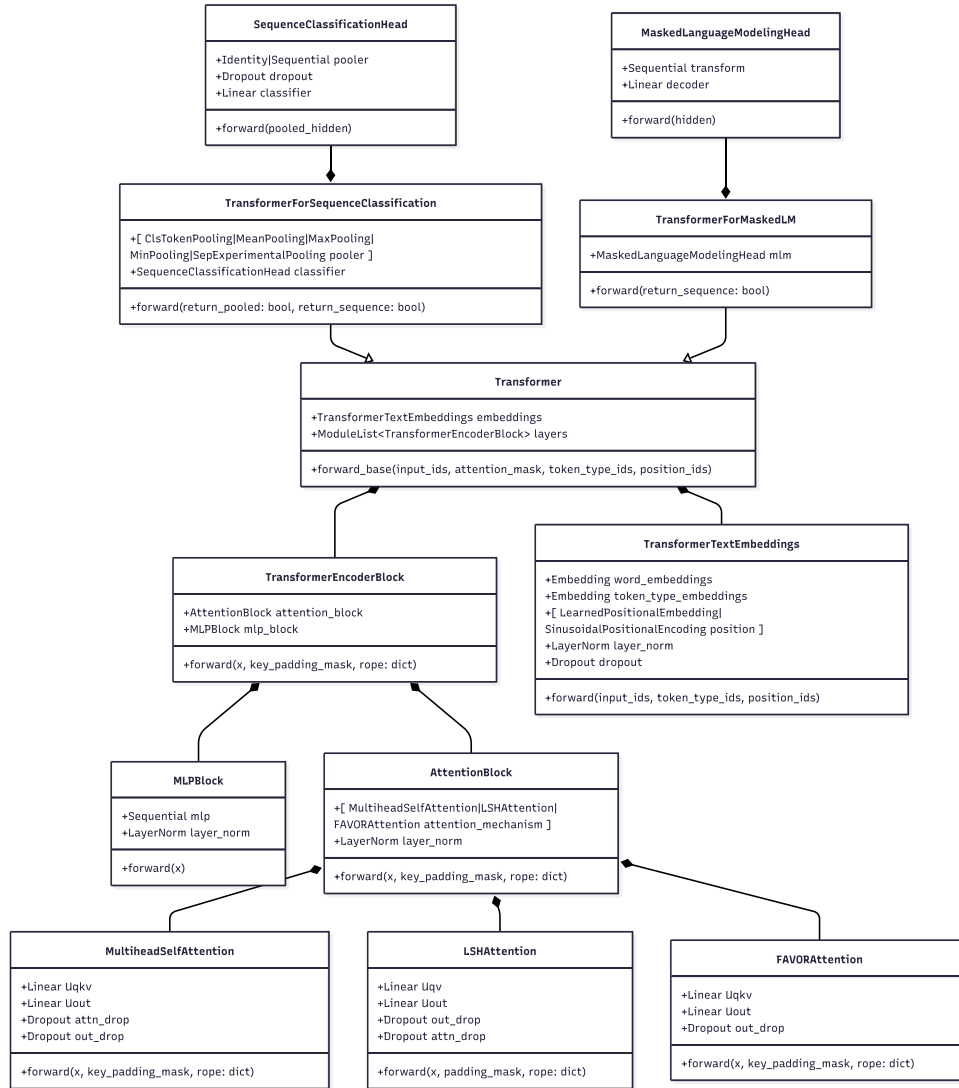


Rysunek 2.1: Diagram komponentów systemu.

2. Architektura systemu

System został zaprojektowany w sposób modułowy. Kod źródłowy projektu podzielony jest na logiczne komponenty odpowiadające poszczególnym funkcjonalnościom. W katalogu `src` znajdują się kluczowe moduły: `models` zawierający implementację architektury transformera (szczegóły w sek. 2.1), `tokenizer` odpowiedzialny za tokenizację tekstu (sek. 2.2), `training` zawierający logikę pętli treningowej (sek. 2.3.2) oraz `logger` służący do monitorowania przebiegu eksperymentów (sek. 2.4). Konfiguracje poszczególnych uruchomień oraz skrypty generujące znajdują się w katalogu `experiments` (sek. 2.5). Aplikacja jest uruchamiana poprzez skrypt `train.py` (szczegóły w sek. 2.3.1).

Diagram komponentów systemu i relacje między modułami przedstawiono na rys. 2.1.



Rysunek 2.2: Diagram klas implementacji transformera. Przedstawiono wyłącznie pola będące modułami PyTorch (dziedziczące po `nn.Module`). Wszystkie argumenty funkcji bez jawnie określonego typu są typu `torch.Tensor`, a wszystkie funkcje zwracają obiekt typu `torch.Tensor`.

2.1. Architektura transformera

Niniejszy rozdział opisuje szczegóły implementacyjne architektury transformera. Architektura wspiera zarówno zadania klasyfikacji tekstu, jak i modelowania języka (MLM). Fundamentem systemu jest klasa bazowa `Transformer`. Klasy `TransformerForSequenceClassification` oraz `TransformerForMaskedLM` – dostosowują model do konkretnych zadań uczenia. Struktura klas oraz ich zależności zostały przedstawione na diagramie klas (rys. 2.2). Szczegóły architektury wykorzystanej w dalszej części pracy w eksperymentach znajdują się w sek. 4.2.2.

2.1. ARCHITEKTURA TRANSFORMERA

2.1.1. Klasa transformera

Model bazowy

Kluczowe parametry konfiguracyjne klasy `Transformer`, definiujące jej strukturę i zachowanie, obejmują:

- `vocab_size`: Określa rozmiar słownika tokenów.
- `max_sequence_length`: Określa maksymalną długość sekwencji wejściowej.
- `embedding_dim (D)`: Główny wymiar ukryty modelu (rozmiar wektorów osadzeń).
- `attention_embedding_dim`: Opcjonalny wymiar projekcji wewnątrz mechanizmu uwagi. Pozwala na sterowanie rozmiarem reprezentacji $Q/K/V$ niezależnie od głównego wymiaru D (nie jest to standard w BERT).
- `num_layers`: Liczba warstw enkodera.
- `num_heads`: Liczba równoległych głów uwagi w każdym bloku.
- `mlp_size`: Rozmiar warstwy ukrytej w sieciach Feed-Forward wewnątrz bloków enkodera.
- `attention_kind`: Wybór konkretnej implementacji mechanizmu uwagi (np. "mha", "lsh", "favor").
- `pos_encoding`: Wybór strategii kodowania pozycji ("learned", "sinusoidal" lub "rope").

Metoda `forward_base` realizuje przetworzenie indeksów tokenów na reprezentacje wektorowe za pomocą modułu `TransformerTextEmbeddings`, a następnie iteracyjne przekształcanie ich przez listę modułów `TransformerEncoderBlock`. Z metody `forward_base` zwracana jest sekwencja stanów ukrytych o wymiarach (B, N, D) , gdzie B to rozmiar wsadu (ang. *batch size*), a N to długość sekwencji. Klasa `Transformer` zarządza również dynamicznym obliczaniem i buforowaniem wartości trygonometrycznych dla kodowania pozycyjnego RoPE, jeśli zostało ono wybrane w konfiguracji (funkcja `build_rope_cache`).

Model do klasyfikacji sekwencji

Klasa `TransformerForSequenceClassification` rozszerza model bazowy o funkcjonalność niezbędną do klasyfikacji całych tekstów. W konstruktorze inicjalizowany jest dodatkowy moduł – `pooler` – (zależny od parametru `pooling`, np. "cls", "mean") oraz głowica klasyfikująca `SequenceClassificationHead`.

Przepływ danych w metodzie `forward` obejmuje:

1. Wywołanie metody `forward_base` z klasy nadrzędnej w celu uzyskania kontekstowych reprezentacji tokenów.
2. Redukcję sekwencji do pojedynczego wektora (B, D) za pomocą wybranego mechanizmu agregacji (ang. *pooling*).
3. Przeprowadzenie transformacji wektora poprzez głowicę klasyfikacyjną, składającą się opcjonalnie z warstwy gęstej (z funkcją aktywacji Tanh) oraz warstwy Dropout aplikowanej przed finalną warstwą liniową.

Model zwraca słownik zawierający zarówno pełną sekwencję wyjściową, wektor po agregacji i logity klasyfikacji $(B, \text{num_labels})$.

Model do modelowania języka z maskowaniem (MLM)

Klasa `TransformerForMaskedLM` jest dedykowana do uczenia nienadzorowanego. Rozszerza klasę `Transformer` o głowicę `MaskedLanguageModelingHead`, która przekształca wyjścia z enkodera z powrotem na przestrzeń słownika (B, N, V) .

Implementacja obsługuje parametr `tie_mlm_weights`. Gdy jest on ustawiony na `True`, wagi warstwy wyjściowej (dekodującej) są współdzielone z wagami macierzy osadzeń wejściowych, co jest standardową praktyką w modelach typu BERT.

2.1.2. Mechanizmy uwagi

Moduł uwagi został zaprojektowany w sposób umożliwiający wymianę mechanizmu uwagi bez ingerencji w pozostałą część architektury. Wybór konkretnej implementacji następuje na podstawie parametru konfiguracyjnego `attention_kind`.

Abstrakcja bloku uwagi

Klasa `AttentionBlock` stanowi standardową implementację dla mechanizmu uwagi. Odpowiada ona za:

- Inicjalizację konkretnej klasy obliczeniowej (`MultiheadSelfAttention`, `FavorAttention`, `LSHAttention`) na podstawie konfiguracji.
- Zastosowanie połączenia rezydualnego (ang. *residual connection*).
- Normalizację wyjścia za pomocą warstwy `LayerNorm`.

Blok ten jest następnie wykorzystywany wewnątrz klasy `TransformerEncoderBlock`, gdzie występuje przed siecią Feed-Forward (MLP).

Standardowa uwaga wielogłowicowa (ang. *Multihead Self Attention (MHA)*)

Implementacja `MultiheadSelfAttention` realizuje klasyczny wzór liczenia uwagi (Scaled Dot-Product Attention (SDPA)) o złożoności obliczeniowej $O(N^2)$. Proces przetwarzania dla sekwencji wejściowej $X \in \mathbb{R}^{B \times N \times D}$ przebiega następująco:

1. Projekcja wejścia na macierze zapytań (Q), kluczy (K) i wartości (V).
2. Podział na H głowic o wymiarze $d_k = D/H$.
3. Opcjonalne zaaplikowanie rotacyjnego kodowania pozycyjnego (RoPE) na tensory Q i K .
4. Obliczenie macierzy uwagi oraz zastosowanie (opcjonalnie) warstwy `Dropout` na macierz prawdopodobieństw:

$$\text{Attention}(Q, K, V) = \text{Dropout} \left(\text{Softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) \right) V. \quad (2.1)$$

5. Scalenie wyników ze wszystkich głowic, projekcja liniowa wyjścia oraz zastosowanie warstwy `Dropout` na wyniku projekcji.

Uwaga FAVOR+ (Performer)

Klasa `FAVORAttention` implementuje mechanizm uwagi o złożoności czasowej oraz pamięciowej $O(N)$. Zamiast obliczać pełną macierz uwagi $N \times N$, model aproksymuje funkcję `Softmax` wykorzystując mapowania cech $\phi(\cdot)$ (szczegóły w sek. 3.3).

Kluczowe elementy implementacji:

- **Ortogonalne cechy losowe (ang. *Gaussian Orthogonal Random Features (GORF)*):** Metoda `_gaussian_orthogonal_random_matrix` generuje macierz ortogonalnych wektorów losowych.
- **Mapowanie cech (ϕ):** Metoda `_phi` dysponuje wariantem transformacji `phi_kind`:
 - "exp" (metoda `_phi_exp`): Realizuje cechy losowe aproksymujące jądro $\exp(x^\top y)$. Wykorzystuje bufor `_omega` przechowujący macierz projekcji losowych
- **Liczenie uwagi:** Obliczane w metodzie `forward`:

$$\hat{V} = D^{-1}(Q'(K'^T V)), \quad (2.2)$$

gdzie $Q' = \phi(Q)$, $K' = \phi(K)$, a D to czynnik normalizacyjny obliczany jako $D = Q'(K'^T \mathbf{1}_N)$.

- **Zarządzanie cechami losowymi:** Implementacja umożliwia przelosowywanie cech w trakcie treningu (parametr `redraw_interval`), co realizuje metoda `_maybe_redraw_features`. Pozwala to na uniknięcie przeuczenia się modelu do konkretnego zestawu projekcji losowych.

Uwaga LSH (Reformer)

Klasa `LSHAttention` implementuje uwagę o złożoności czasowej $O(N \log N)$ oraz pamięciowej $O(N)$. Mechanizm ten zakłada, że tokeny powinny zwracać uwagę głównie na tokeny do nich podobne (znajdujące się w tym samym tzw. kubełku haszującym).

Kluczowe elementy implementacji (szczegóły w sek. 3.2):

- **Współdzielone Q i K:** Zgodnie z architekturą Reformera, projekcje zapytań i kluczy są tożsame ($Q = K$), co jest wymagane do poprawnego działania LSH.
- **Haszowanie i sortowanie:** Wykorzystanie losowych projekcji w celu przypisania tokenów do kubełków. Następnie sekwencja jest sortowana według numeru kubełka z zachowaniem kolejności tokenów wewnątrz kubełka.
- **Przetwarzanie w blokach (Chunking):** Posortowana sekwencja jest dzielona na bloki o stałej długości (`chunk_size`). Uwaga jest obliczana wewnątrz każdego bloku, przy czym każdy blok ma dostęp do kontekstu bloku poprzedniego i następnego (co pozwala uwzględnić elementy tego samego kubełka, które w wyniku podziału trafiły do sąsiednich bloków).
- **Maskowanie:** Zaimplementowano możliwość wyboru czy tokeny mają zwracać uwagę na tokeny z tego samego bloku ale innego kubełka (`mask_within_chunks`).

2.1.3. Blok Enkodera i MLP

Sieć Feed-Forward (MLPBlock)

Klasa `MLPBlock` implementuje sieć neuronową typu Feed-Forward – warstwa ta składa się z sekwencji:

1. Projekcja liniowa z wymiaru modelu D na wymiar pośredni `mlp_size`.
2. Funkcja aktywacji GELU.
3. Projekcja liniowa powrotna na wymiar D .
4. Warstwa Dropout.

Pełny blok enkodera (TransformerEncoderBlock)

Klasa `TransformerEncoderBlock` agreguje pojedynczą warstwę modelu. W jej skład wchodzi sekwencyjnie: blok uwagi (`AttentionBlock`) oraz blok MLP (`MLPBlock`). W metodzie `forward`:

1. Dane wejściowe trafiają najpierw do bloku uwagi (z uwzględnieniem maskowania i ewentualnego kodowania RoPE).
2. Wyjście bloku uwagi jest przekazywane do bloku MLP.

2.1.4. Kodowanie pozycyjne

Implementacja w klasie `TransformerTextEmbeddings` wspiera trzy podejścia do kodowania pozycyjnego, sterowane parametrem konfiguracyjnym `pos_encoding`.

TransformerTextEmbeddings

Klasa ta łączy:

- **Osadzenia słów (Word Embeddings):** Standardowa warstwa `nn.Embedding` mapująca identyfikatory tokenów na wektory o wymiarze D .
- **Osadzenia typów (Token Type Embeddings):** Opcjonalne osadzenia segmentów (np. dla par zdań).
- **Informację pozycyjną:** W przypadku kodowania absolutnego (sinusoidalne lub wyuczone), wektory pozycji są dodawane bezpośrednio do sumy osadzeń słów i typów.

Finalna reprezentacja jest normalizowana (`LayerNorm`) oraz poddawana regularyzacji (`Dropout`).

Kodowanie sinusoidalne (Sinusoidal)

Klasa `SinusoidalPositionalEncoding` implementuje deterministyczny schemat kodowania absolutnego, zgodny z pierwotną architekturą Transformera [14]. Wektory pozycyjne nie są parametrami uczonymi, lecz są wyliczane na podstawie funkcji trygonometrycznych o geometrycznie wzrastających długościach fal.

Dla pozycji p i wymiaru i , wartość kodowania wynosi:

$$PE_{(p,2i)} = \sin\left(\frac{p}{10000^{2i/D}}\right) \quad (2.3)$$

$$PE_{(p,2i+1)} = \cos\left(\frac{p}{10000^{2i/D}}\right) \quad (2.4)$$

Implementacja wykorzystuje bufor `register_buffer`, co pozwala na wyliczenie macierzy raz przy inicjalizacji modelu i dynamiczne jej krojenie (ang. *slicing*) w zależności od długości aktualnej sekwencji.

Wyuczone kodowanie absolutne

Klasa `LearnedPositionalEmbedding` realizuje podejście, w którym pozycje są modelowane jako wyuczalne wektory wagi macierzy o wymiarach (N_{max}, D) . Każdemu indeksowi pozycji przyporządkowany jest unikalny wektor, który jest optymalizowany w procesie uczenia.

Rotacyjne kodowanie pozycyjne – (ang. *Rotary Positional Embeddings (RoPE)* [12])

W przypadku wyboru kodowania `rope` klasa `TransformerTextEmbeddings` nie dodaje addytywnych wektorów pozycyjnych do wejścia. Zamiast tego informacja pozycyjna jest aplikowana bezpośrednio na tensory zapytań Q i kluczy K wewnątrz mechanizmu uwagi.

Implementacja w module `rotary.py` składa się z dwóch etapów:

1. **Prekomputacja (`build_rope_cache`):** dla wymiaru głowy D (parzystego) definiuje się częstotliwości

$$\theta_i = 10000^{-\frac{2i}{D}}, \quad i = 0, 1, \dots, \frac{D}{2} - 1. \quad (2.5)$$

Następnie dla każdej pozycji m (oraz każdego i) wylicza się tablice:

$$\cos(m\theta_i), \quad \sin(m\theta_i). \quad (2.6)$$

2. **Aplikacja (`apply_rope`):** dla każdej pary kolejnych składowych $(x_{m,2i}, x_{m,2i+1})$ na pozycji m wykonuje się rotację o kąt $m\theta_i$:

$$\begin{pmatrix} x'_{m,2i} \\ x'_{m,2i+1} \end{pmatrix} = \begin{pmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{pmatrix} \begin{pmatrix} x_{m,2i} \\ x_{m,2i+1} \end{pmatrix}. \quad (2.7)$$

Implementacja funkcji `_rotate_half` realizuje operację `rotate_half([x2i, x2i+1]) = [-x2i+1, x2i]` w sposób zwektoryzowany, co umożliwia obliczenie rotacji bez jawnego tworzenia macierzy rotacji dla każdego tokena:

$$\mathbf{x}' = \mathbf{x} \odot \cos(m\boldsymbol{\theta}) + \text{rotate_half}(\mathbf{x}) \odot \sin(m\boldsymbol{\theta}), \quad (2.8)$$

gdzie \odot oznacza mnożenie element po elemencie, a $\boldsymbol{\theta} = (\theta_0, \dots, \theta_{\frac{D}{2}-1})$.

2.1.5. Warstwa agregacji (ang. *Pooling*)

Celem warstwy agregacji jest redukcja wymiarowości sekwencji stanów ukrytych $H \in \mathbb{R}^{B \times N \times D}$ (zwracanej przez enkoder) do pojedynczego wektora reprezentacji całego tekstu $h_{pooled} \in \mathbb{R}^{B \times D}$. Zaimplementowano pięć strategii agregacji:

Agregacja tokenu CLS (ClsTokenPooling)

Standardowa strategia dla modeli typu BERT. Jako reprezentację całej sekwencji przyjmuje się stan ukryty pierwszego tokenu specjalnego (zwyczajowo [CLS]).

$$(h_{pooled})_{b,d} = H_{b,1,d}. \quad (2.9)$$

Agregacja uśredniająca (MeanPooling)

Strategia polegająca na obliczeniu średniej arytmetycznej z wektorów wszystkich tokenów w sekwencji:

$$(h_{pooled})_{b,d} = \frac{1}{N} \sum_{i=1}^N H_{b,i,d}. \quad (2.10)$$

Pooling maksymalny i minimalny (MaxPooling, MinPooling)

Strategie wybierające odpowiednio największą lub najmniejszą wartość cechy wzdłuż wymiaru sekwencji.

$$(h_{\max})_{b,d} = \max_{1 \leq i \leq N} H_{b,i,d}, \quad (h_{\min})_{b,d} = \min_{1 \leq i \leq N} H_{b,i,d}. \quad (2.11)$$

Agregacja uśredniająca z krokiem (MeanStepPooling)

Niestandardowa strategia agregacji, polega na obliczaniu średniej arytmetycznej z podzbioru wektorów sekwencji wybieranych w stałych odstępach (parametr **step** (s)).

$$(h_{pooled})_{b,d} = \frac{1}{K} \sum_{k=0}^{K-1} H_{b,1+ks,d}, \quad K = \max\{K' \in \mathbb{N} : 1 + (K' - 1)s \leq N\}. \quad (2.12)$$

2.1.6. Głowice zadaniowe

Głowice zadaniowe to końcowe moduły sieci, które transformują reprezentację wektorową (sekwencyjną lub zagregowaną) na przestrzeń wyjściową specyficzną dla danego zadania.

Głowica do klasyfikacji sekwencji (`SequenceClassificationHead`)

Moduł ten przyjmuje na wejściu wektor po agregacji (B, D) i rzutuje go na przestrzeń etykiet $(B, \text{num_labels})$. Implementacja wspiera różne architektury warstwy pośredniej (ang. *pooler*), sterowane parametrem `pooler_type`:

- **Styl BERT:** Składa się z warstwy gęstej zachowującej wymiarowość, funkcji aktywacji Tanh, a następnie warstwy wyjściowej:

$$y = \text{Linear}(\text{Dropout}(\text{Tanh}(\text{Linear}(x)))). \quad (2.13)$$

- **Styl RoBERTa:** Charakteryzuje się dodatkowym Dropoutem na wejściu:

$$y = \text{Linear}(\text{Dropout}(\text{Tanh}(\text{Linear}(\text{Dropout}(x))))). \quad (2.14)$$

- **Brak (None):** Bezpośrednie rzutowanie wejścia na wyjście (z uwzględnieniem Dropoutu).

Głowica modelowania języka (`MaskedLanguageModelingHead`)

Głowica służąca do pretreningu w zadaniu MLM. Przyjmuje ona sekwencję stanów ukrytych (B, N, D) i zwraca predykcje dla każdego tokenu w słowniku (B, N, V) . Struktura głowicy jest zgodna ze standardem BERT i obejmuje:

1. Transformację nieliniową: Warstwa liniowa \rightarrow funkcja aktywacji GELU \rightarrow normalizacja LayerNorm.
2. Warstwę dekodującą – projekcja na rozmiar słownika.

2.2. Tokenizer

W projekcie wykorzystano algorytm tokenizacji WordPiece (używany w oryginalnym modelu BERT [5]). Aby uprościć procesy trenowania i przetwarzania danych, zaimplementowano klasę pomocniczą `WordPieceTokenizerWrapper` – stanowi ona nakładkę na bibliotekę `tokenizers` oraz `transformers` Hugging Face. Jej głównym celem jest abstrakcja operacji niskopoziomowych i dostarczenie API do przygotowywania danych dla modelu.

2.2.1. Trening i inicjalizacja

Wrapper umożliwia wytrenowanie nowego tokenizera na korpusie tekstowym użytkownika za pomocą metody `train`. Wykorzystuje ona implementację `BertWordPieceTokenizer`, która bu-

2.2. TOKENIZER

duje słownik podjednostek (subwords) o zadanej wielkości (domyślnie 30000 tokenów). Kluczowe etapy procesu to:

- Normalizacja tekstu (zamiana na małe litery, usuwanie akcentów).
- Trening algorytmu WordPiece na wskazanych plikach tekstowych.
- Konfiguracja post-processora, który automatycznie dodaje specjalne [CLS] na początku i [SEP] na końcu sekwencji (w zależności od konfiguracji).
- Zapisanie wytrenowanego modelu (plik `vocab.txt` oraz `tokenizer.json`) we wskazanym katalogu.

Metoda `load` inicjalizuje szybki tokenizer `BertTokenizerFast`, wykorzystując plik `vocab.txt` (oraz ewentualnie `tokenizer.json`). Na potrzeby eksperymentów będziemy korzystać z gotowego `vocab.txt` używanego w oryginalnym BERT.

2.2.2. Przetwarzanie danych (Encoding)

Klasa oferuje metody `encode` oraz `encode_pandas` służące do konwersji surowego tekstu na tensory wejściowe modelu (z wykorzystaniem `BertTokenizerFast`). Proces ten obejmuje:

1. Normalizację tekstu.
2. Tokenizację tekstu.
3. Obcięcie sekwencji do maksymalnej długości (`max_length`) lub dopełnienie (padding) tokenem [PAD] do tej długości.
4. Generowanie maski uwagi (`attention_mask`) na podstawie tokenów paddingu ([PAD]), gdzie wartość `True` oznacza tokeny paddingu, które powinny być ignorowane przez mechanizm uwagi.
5. Opcjonalne dołączenie etykiet.

Wynikiem jest obiekt `TensorDataset` gotowy do użycia z `DataLoader` w PyTorch, zawierający tensory `input_ids`, `attention_mask` oraz opcjonalnie `labels`.

2.2.3. Maskowanie dla MLM

Dla potrzeb uczenia nienadzorowanego (Masked Language Modeling), zaimplementowaliśmy metodę `mask_input_for_mlm`. Realizuje ona maskowanie tokenów zgodnie z następującym schematem:

- Wybór tokenów do predykcji (domyślnie 15%).
- Zastąpienie 80% wybranych tokenów tokenem specjalnym `[MASK]` (domyślnie 80%).
- Zastąpienie wybranych tokenów losowym słowem ze słownika (domyślnie 10%).
- Pozostawienie tokenów bez zmian (domyślnie 10%).

Metoda zwraca zarówno zamaskowane wejścia, jak i etykiety, gdzie tokeny niepodlegające predykcji oznaczone są wartością -100, co jest domyślną wartością dla funkcji kosztu `CrossEntropyLoss` w PyTorch.

2.2.4. Dostępne metody i ich argumenty

Metoda `train` Służy do utworzenia nowego słownika.

- `tokenizer_dir: str`: Katalog wyjściowy dla plików tokenizera.
- `input: str | list[str]`: Ścieżka do pliku tekstowego lub lista ścieżek do plików treningowych.
- `vocab_size: int`: Rozmiar słownika.
- `min_frequency: int`: Minimalna częstość występowania tokenu.

Metoda `encode` Konwertuje dane tekstowe na `TensorDataset` (lub słownik). Wymaga uprzedniego załadowania tokenizera metodą `load()`.

- `input: str | list[str]`: Może przyjmować:
 - Ścieżkę do pliku tekstowego (lub listę ścieżek) — każda linia traktowana jest jako osobny przykład.
 - Listę surowych tekstów (stringów).
- `max_length: int`: Maksymalna długość sekwencji (dopełnianie/przycinanie).
- `labels: list[int]` (opcjonalnie): Lista etykiet dla przykładów.

Metoda `encode_pandas` Alternatywa dla `encode`, pozwalająca na bezpośrednie użycie ramki danych `pandas`.

- `df: pandas.DataFrame`: Ramka danych.
- `text_col: str`: Nazwa kolumny z tekstem.

2.3. TRENING

- `max_length: int`: Maksymalna długość sekwencji.
- `label_col: str` (opcjonalnie): Nazwa kolumny z etykietami.

Metoda `mask_input_for_mlm` Pomocnicza funkcja do generowania masek dla zadania MLM. Przyjmuje `input_ids` i zwraca parę (`input_ids_masked`, `labels`).

2.3. Trening

System treningowy został zaprojektowany w architekturze składającej się ze skryptu treningowego (punkt wejścia (ang. *entry point*)) oraz klasy `TrainingLoop`, która zawiera właściwą logikę optymalizacji modelu.

2.3.1. Punkt wejścia i inicjalizacja środowiska

Główny skrypt uruchomieniowy odpowiada za zestawienie eksperymentu na podstawie argumentów CLI (nazwa eksperymentu, tryb pracy) oraz pliku konfiguracyjnego (zdefiniowanego wewnątrz katalogu eksperymentu). Proces ten przebiega wieloetapowo:

1. **Determinizm:** Na początku ustawiane są ziarna generatorów liczb losowych (Python, NumPy, PyTorch) za pomocą funkcji `set_global_seed`.
2. **Fabryka modelu:** W zależności od trybu pracy (`mode`), skrypt instancjonuje odpowiednią klasę modelu:
 - **Pretrening (MLM):** Inicjalizowany jest `TransformerForMaskedLM`.
 - **Finetuning:** Inicjalizowany jest `TransformerForSequenceClassification`. W tym przypadku następuje etap transferu wiedzy – wagi są ładowane z zapisanego stanu pretreningowego z flagą `strict=False`. Pozwala to na załadowanie parametrów enkodera przy jednoczesnym zignorowaniu braku dopasowania w warstwach wyjściowych (zastąpienie głowicy MLM nową głowicą klasyfikacyjną).
3. **Przygotowanie danych:** Tworzone są instancje `DataLoader` dla zbiorów treningowych, walidacyjnych i testowych (jeśli zbiory walidacyjne i testowe są zdefiniowane w konfiguracji).

2.3.2. Logika pętli treningowej (TrainingLoop)

Zainicjowany model przekazywany jest do obiektu `TrainingLoop`, który zarządza pełnym procesem uczenia. Przepływ danych w pojedynczym kroku treningowym (`_train_step`) obejmuje: przygotowanie wsadu (ang. *batch*), przejście w przód (ang. *forward pass*) w kontekście `torch.amp.autocast`, obliczenie funkcji straty, propagację wsteczną, a następnie – warunkowo – aktualizację wag (w zależności od kroku akumulacji gradientów). Implementacja integruje zestaw współcześnie stosowanych technik optymalizacyjnych:

- **Automatyczna mieszana precyzja (AMP):** Zastosowanie `torch.amp.autocast` umożliwia wykonywanie wybranych operacji w obniżonej precyzji (FP16 lub BF16), co zwykle przyspiesza trening oraz redukuje zużycie pamięci GPU, przy zachowaniu jakości uczenia.
- **Skalowanie gradientów (Gradient scaling):** Wykorzystywany jest `torch.amp.GradScaler`, który dynamicznie skaluje wartości funkcji straty (a tym samym gradienty) w celu poprawy stabilności numerycznej. Przed wykonaniem operacji takich jak przycinanie normy gradientów, gradienty są odskalowywane (tj. po `scaler.unscale_()`).
- **Akumulacja gradientów:** Parametr `grad_accum_steps` pozwala uniezależnić rozmiar wsadu od ograniczeń pamięci GPU poprzez akumulowanie gradientów z wielu mikro-kroków przed wykonaniem kroku optymalizatora. W praktyce odpowiada to trenowaniu z większym wsadem przy rzadszej aktualizacji wag.
- **Stabilizacja (clipping):** Przycinanie normy gradientów (`clip_grad_norm_()`).
- **Harmonogram uczenia (scheduler):** Zastosowano harmonogram współczynnika uczenia typu *cosine decay* z liniową fazą rozgrzewki (ang. *warmup*). Krok harmonogramu wykonywany jest spójnie z krokami optymalizatora, tj. w momentach faktycznej aktualizacji wag.

2.3.3. Dynamiczna optymalizacja wsadów

W celu zwiększenia wydajności przetwarzania sekwencji o zróżnicowanej długości, zaimplementowano funkcję kolacjonującą (ang. *collate function*) `make_collate_trim_to_longest`. Funkcja ta analizuje każdy wsad i przycina tensory wejściowe do długości najdłuższego rzeczywistego przykładu w danym wsadzie. Pozwala to na ograniczenie zbędnych obliczeń na tokenach [PAD].

2.3.4. Zarządzanie stanem (ang. *Checkpointing*)

Skrypt obsługuje zarządzanie stanem treningu:

- **Zapis najlepszego modelu:** Po każdej epoce następuje walidacja. Jeśli strata walidacyjna jest najniższa w historii, zapisywany jest pełny stan eksperymentu (model, optymalizator, harmonogram uczenia, skaler) do pliku `best-model.ckpt`.
- **Zapis modelu końcowego:** Po zakończeniu procesu uczenia zapisywane są finalne wagi modelu.
- **Wznawianie (Resume):** W trybie pretreningu możliwa jest kontynuacja przerwanej procesu uczenia. Funkcja `load_resume` odtwarza stan wszystkich komponentów, pozwalając na płynne wznowienie obliczeń od ostatniego zapisanego kroku.
- **Transfer wiedzy:** System umożliwia inicjalizację treningu (np. na nowym zbiorze danych) z wykorzystaniem jedynie wag modelu z wybranego zapisanego stanu – wczytywane są jedynie parametry modelu, a pozostałe obiekty pomocnicze są inicjalizowane od nowa.

2.4. Logowanie przebiegu treningu

Monitorowanie postępów eksperymentów realizowane jest przez hybrydowy system logowania zaimplementowany w klasie `WandbRun`. Rozwiązanie to integruje chmurową platformę analityczną *Weights & Biases* (W&B) z lokalnym archiwizowaniem danych w formacie CSV, zapewniając redundancję i łatwy dostęp do wyników.

2.4.1. Integracja z *Weights & Biases*

Głównym kanałem zbierania metryk jest serwis W&B, w którym agregowane są wyniki wszystkich eksperymentów. Klasa `WandbRun` odpowiada za:

- **Inicjalizację sesji:** Metoda `__init__` nawiązuje połączenie z projektem określonym w konfiguracji, przysyłając jednocześnie pełny słownik parametrów (`config`).
- **Organizacja metryk:** Metody `log_train` oraz `log_eval` automatycznie dodają odpowiednie prefiksy (`train/`, `eval/`, `test/`) do nazw zmiennych w celu grupowania wykresów.

2.4.2. Lokalny zapis danych (CSV)

W przypadku ustawienia w konfiguracji `log_metrics_csv = True` logger utrzymuje lokalną kopię wszystkich metryk. Dane są zapisywane w plikach:

- `metrics/train/metrics.csv` dla danych treningowych.
- `metrics/eval/metrics.csv` dla danych walidacyjnych i testowych.

2.4.3. Ewaluacja i metryki

- **Dla MLM:** Podstawową metryką jest perplexja (ang. *perplexity*), wyliczana jako e^{loss} , gdzie loss to średnia strata entropii krzyżowej na token.
- **Dla klasyfikacji:** Wykorzystano bibliotekę `scikit-learn` do obliczania szerokiego spektrum metryk:
 - **Metryki ogólne:** *Accuracy*, *Balanced Accuracy*.
 - **Metryki uśrednione:** *Precision*, *Recall* oraz *F1 Score* w wariantach *macro* i *micro*.
 - **Pewność modelu:** Średnia pewność predykcji (ang. *confidence*) oraz entropia rozkładu prawdopodobieństwa.
 - **Top-K:** Dokładność dla $k \in \{3, 5\}$ (ang. *Top-k Accuracy*).
 - **Metryki per klasa:** Dla zadań z niewielką liczbą etykiet (domyślnie ≤ 10) raportowane są precyzja, czułość i F1 dla każdej klasy osobno.
- **Metryki systemowe:** Platforma `Weights & Biases` automatycznie gromadzi dane o użyciu zasobów sprzętowych (GPU, CPU, pamięć), czasie trwania operacji oraz liczbie wykonanych kroków i epok.

2.5. Konfiguracja eksperymentów

Zarządzanie eksperymentami odbywa się poprzez dedykowane skrypty pomocnicze, które automatyzują tworzenie struktury plików konfiguracyjnych wewnątrz katalogów eksperymentów (`experiments/pretraining` oraz `experiments/finetuning`). Każde uruchomienie jest w pełni determinowane przez plik `config.yaml` znajdujący się w katalogu danego eksperymentu. Tabele 2.1, 2.2, 2.3, 2.4, 2.6, 2.7 prezentują szczegółowy opis wszystkich dostępnych parametrów konfiguracyjnych wraz z przykładowymi wartościami domyślnymi.

2.5.1. Inicjalizacja pretreningu

Tworzenie nowego eksperymentu pretreningowego obsługiwane jest przez skrypt `generate_pretraining_experiment.py`. Proces ten przebiega według następującego schematu:

2.5. KONFIGURACJA EKSPERYMENTÓW

1. **Walidacja i struktura:** Skrypt weryfikuje unikalność nazwy eksperymentu w katalogu `experiments/pretraining`, a następnie tworzy dedykowany katalog wraz z plikiem `config.yaml`.

2. **Szablony i wznawianie:**

- W trybie standardowym: wczytywany jest szablon bazowy z `config_templates/pretraining.yaml` i zapisywany do pliku `config.yaml`.
- W trybie wznawiania (flaga `-rp`): konfiguracja jest kopiowana z istniejącego eksperymentu, a sekcja `training.resume` jest automatycznie uzupełniana o ścieżkę do ostatniego zapisanego stanu (`model.ckpt`).

2.5.2. Inicjalizacja dostrajania

Skrypt `generate_finetuning_experiment.py` realizuje logikę niezbędną do przeprowadzenia douczania modelu na zadaniu docelowym.

Aby zapewnić kompatybilność, skrypt wymaga podania nazwy istniejącego eksperymentu pretreningowego (flaga `-p`) oraz nazwy nowego eksperymentu dostrajania (flaga `-f`). Procedura generowania konfiguracji obejmuje:

1. **Weryfikację źródeł:** Sprawdzenie istnienia katalogu i pliku konfiguracyjnego eksperymentu bazowego.
2. **Kopiowanie architektury:** Sekcje `architecture` oraz `tokenizer` są kopiowane bezpośrednio z konfiguracji pretreningu do konfiguracji dostrajania (*finetuning*). Gwarantuje to, że model docelowy będzie miał identyczne wymiary jak model bazowy, co jest warunkiem koniecznym poprawnego załadowania wag. Reszta parametrów jest kopiowana z szablonu `config_templates/finetuning.yaml`.
3. **Relatywizację ścieżek:** Ścieżka do eksperymentu bazowego jest zapisywana w sekcji `pretrained_experiment.path` jako ścieżka względna względem korzenia projektu. Umożliwia to skryptowi treningowemu zlokalizowanie zapisanego stanu pretreningowego modelu bazowego.

Tabela 2.1: Parametry eksperymentu, logowania i tokenizacji

Sekcja	Parametr	Przykład	Opis
experiment	name	<i>run_v1</i>	Unikalna nazwa eksperymentu.
	kind	finetuning	Typ: <code>pretraining</code> lub <code>finetuning</code> .
	output_dir	experiments/...	Katalog wyjściowy eksperymentu.
	seed	420	Ziarno losowości.
logging	use_wandb	true	Czy używać Weights & Biases.
	wandb.entity	wandb-team	Nazwa zespołu W&B.
	wandb.project	project-name	Nazwa projektu W&B.
	wandb.run_name	<i>run_v1</i>	Nazwa sesji W&B.
	log_eval_metrics	true	Czy logować metryki walidacyjne.
	log_metrics_csv	false	Czy zapisywać metryki do plików CSV.
	log_gpu_memory	true	Czy logować zużycie pamięci GPU.
	csv_train_metrics_path	metrics/...	Ścieżka do CSV z metrykami treningowymi.
	csv_eval_metrics_path	metrics/...	Ścieżka do CSV z metrykami walidacyjnymi.
tokenizer	wrapper_path	src/...	Ścieżka do klasy wrappera tokenizera.
	vocab_dir	.../BERT_orig	Katalog ze słownikiem tokenizera.
	max_length	512	Maksymalna długość sekwencji tokenizacji.

Tabela 2.2: Hiperparametry architektury Transformera

Sekcja	Parametr	Przykład	Opis
architecture	embedding_dim	512	Wymiar osadzeń i stanów ukrytych (D).
	num_layers	4	Liczba bloków enkodera.
	mlp_size	2048	Rozmiar warstwy ukrytej w MLP.
	mlp_dropout	0.1	Dropout w bloku MLP.
	embedding_dropout	0.1	Dropout na osadzeniach wejściowych.
	pos_encoding	rope	Typ: <code>learned</code> , <code>sinusoidal</code> , <code>rope</code> .
	rope.rope_base	10000.0	Podstawa częstotliwości θ dla RoPE.
	rope.rope_scale	1.0	Skalowanie częstotliwości RoPE.

Tabela 2.3: Parametry mechanizmu uwagi

Sekcja	Parametr	Przykład	Opis
attention	kind	lsh	Typ: mha, lsh, favor.
	attention_embedding_dim	512	Opcjonalny wymiar projekcji uwagi.
	num_heads	8	Liczba głowic uwagi (H).
	projection_bias	true	Czy dodać bias w projekcjach Q/K/V/Out.
	attn_out_drop	0.1	Dropout na wyjściu bloku uwagi.
	attn_dropout	0.0	Dropout na macierzy uwagi (po Soft-max).
attention.mha	use_native_sdpa	true	Czy użyć natywnej implementacji Flash Attention.
attention.lsh	num_hashes	4	Liczba rund haszowania.
	chunk_size	64	Rozmiar bloku lokalnej uwagi.
	mask_within_chunks	true	Czy maskować uwagę wewnątrz bloku.
attention.favor	nb_features	256	Liczba cech losowych (m).
	ortho_features	true	Czy użyć ortogonalnych cech (GORF).
	redraw_interval	0	Interwał przelosowania cech (0 = brak).
	phi	exp	Funkcja phi: exp, relu, elu.
	stabilize	true	Czy stabilizować numerycznie.
	eps	1e-6	Dodawany do denominatora.

Tabela 2.4: Hiperparametry procesu treningowego

Sekcja	Parametr	Przykład	Opis
training	batch_size	64	Rozmiar wsadu.
	epochs	10	Liczba epok treningowych.
	learning_rate	2e-4	Maksymalny współczynnik uczenia.
	warmup_ratio	0.1	Udział kroków rozgrzewki.
	min_lr_ratio	0.2	Minimalny współczynnik uczenia jako ułamek maksymalnego.
	weight_decay	0.01	Współczynnik regularizacji wag (L2).
	max_grad_norm	1.0	Maksymalna norma gradientów.
	grad_accum_steps	1	Liczba kroków akumulacji gradientu.
	use_amp	true	Czy użyć precyzji mieszanej (AMP).
	loss	cross_entropy	Funkcja straty.
	device	auto	Urządzenie: auto, cuda, cpu.
training	head_lr_mult	1.0	Mnożnik współczynnika uczenia dla głowicy klasyfikacyjnej.
(Dostrajanie)	backbone_lr_mult	0.5	Mnożnik współczynnika uczenia dla enkodera.
	freeze	true	Czy włączyć zamrożanie.
	freeze_n_layers	3	Liczba zamrożonych warstw enkodera.
	freeze_epochs	1	Liczba epok z zamrożonymi warstwami.
	freeze_embeddings	true	Czy zamrozić warstwę osadzeń.

Tabela 2.5: Parametry kontynuacji treningu

Sekcja	Parametr	Przykład	Opis
pretrained_exp	name	<i>pre_v1</i>	Nazwa eksperymentu pretreningu.
(Dostrajanie)	path	experiments/...	Ścieżka do katalogu pretreningu.
	checkpoint	checkpoints/...	Ścieżka do zapisanego stanu pretreningu.
training.resume	is_resume	false	Czy wznawiać pretrening.
(Pretrening)	resume_pretraining_name	<i>pre_v1</i>	Nazwa wznawianego eksperymentu.
	checkpoint_path	checkpoints/...	Ścieżka do zapisanego stanu pretreningu.
	strict	true	Czy wymagać pełnej zgodności wag.
	load_only_model_state	true	Czy ładować tylko wagi modelu.

Tabela 2.6: Parametry głowic zadaniowych

Typ głowicy	Parametr	Przykład	Opis
mlm_head (Pretrening)	tie_mlm_weights	true	Czy współdzielić wagi dekodera i osadzeń.
	mask_p	0.15	Prawdopodobieństwo zamaskowania tokenu.
	mask_token_p	0.8	Szansa na zastąpienie przez [MASK].
	random_token_p	0.1	Szansa na zastąpienie losowym słowem.
class_head (Dostrajanie)	num_labels	2	Liczba klas wyjściowych.
	pooling	cls	Agregacja: <code>cls</code> , <code>mean</code> , <code>max</code> , <code>min</code> .
	pooler_type	bert	Warstwa pośrednia: <code>bert</code> , <code>roberta</code> , <code>null</code> .
	classifier_dropout	0.1	Dropout przed klasyfikatorem.

Tabela 2.7: Parametry konfiguracji danych

Sekcja	Parametr	Przykład	Opis
data.train	shuffle	true	Czy mieszać dane treningowe.
	dataset_path	data/train/...	Ścieżka do zbioru treningowego.
data.val	shuffle	false	Czy mieszać dane walidacyjne.
	dataset_path	data/val/...	Ścieżka do zbioru walidacyjnego.
data.test	shuffle	false	Czy mieszać dane testowe.
	dataset_path	data/test/...	Ścieżka do zbioru testowego.

2.6. Instrukcja instalacji i ocena narzędzi

Poniżej przedstawiono kroki niezbędne do uruchomienia systemu.

2.6.1. Wymagania systemowe

- Python 3.12 lub nowszy
- CUDA (opcjonalnie, do treningu na GPU). Aby wykorzystać akcelerację GPU, upewnij się, że masz zainstalowane odpowiednie sterowniki CUDA.

2.6.2. Instalacja środowiska

1. Utworzenie wirtualnego środowiska Python:

```
1 python -m venv .venv
2 source .venv/bin/activate    # Linux/macOS
3 # lub na Windows:
4 # .\.venv\Scripts\Activate.ps1
```

Listing 2.1: Tworzenie środowiska wirtualnego

2. Aktualizacja pip i instalacja zależności:

```
1 pip install --upgrade pip
2 pip install -r requirements.txt
```

Listing 2.2: Instalacja zależności

Wszystkie pakiety oraz ich wersje (znajdujące się w pliku `requirements.txt`) są przedstawione w tabeli 2.8.

2.6.3. Ocena narzędzi

PyTorch Projekt zakłada pełną implementację modelu w środowisku PyTorch, z wykorzystaniem jedynie wybranych elementów ekosystemu HuggingFace do obsługi tokenizacji i zarządzania zbiorami danych.

HuggingFace Z HuggingFace wykorzystano wyłącznie moduły wspierające przygotowanie danych:

- **datasets** – do pobierania zbiorów danych.

Tabela 2.8: Wykorzystywane biblioteki Python

Biblioteka	Wersja
<code>torch</code>	2.8.0
<code>transformers</code>	4.56.2
<code>tokenizers</code>	0.22.1
<code>pandas</code>	2.2.3
<code>numpy</code>	1.26.4
<code>scikit-learn</code>	1.6.1
<code>wandb</code>	0.22.1
<code>PyYAML</code>	6.0.2
<code>pytest</code>	8.3.4
<code>datasets</code>	4.3.0
<code>pydantic</code>	$\geq 2.12.0$

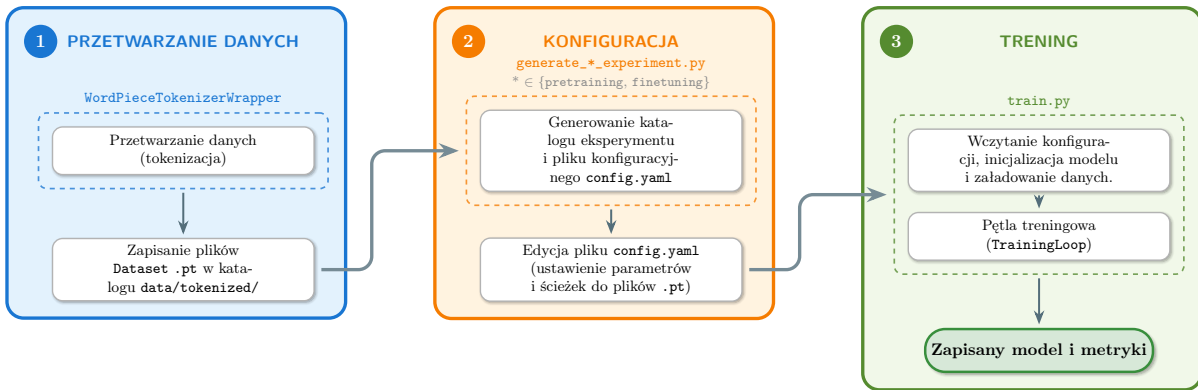
- `tokenizers` – do treningu tokenizatora (stworzenia słownika) z wykorzystaniem klasy `BertWordPieceTokenizer`.
- `transformers` – do tokenizacji danych wejściowych z wykorzystaniem klasy `BertTokenizerFast` (na podstawie gotowego słownika).

Pozostałe biblioteki. W projekcie wykorzystano również szereg innych narzędzi:

- `numpy` i `scikit-learn` – biblioteki wykorzystane do obliczania metryk ewaluacyjnych modelu.
- `pandas` – narzędzie użyte do wstępnego przetwarzania i analizy danych.
- `wandb` – platforma *Weights & Biases* służąca do śledzenia eksperymentów i logowania metryk.
- `PyYAML` – biblioteka do obsługi plików konfiguracyjnych YAML.
- `pytest` – framework do testów jednostkowych.
- `pydantic` – wykorzystany do ukrycia nieszkodliwych ostrzeżeń `wandb`.

2.7. Podręcznik użytkownika

Niniejszy podręcznik opisuje, jak korzystać z systemu. Uproszczony schemat użytkowania systemu przedstawiony jest na rys. 2.3.



Rysunek 2.3: Schemat użytkowania systemu

2.7.1. Przechowywanie danych

Katalog `data/` służy do przechowywania surowych oraz stokenizowanych danych z następującą strukturą:

- `data/raw/` — surowe pliki (np. CSV, TXT, Parquet).
- `data/tokenized/` — gotowe do użycia zestawy danych w formacie `.pt` (zserializowane przez `torch.save`), kompatybilne z oczekiwanym formatem `DataLoader`.

Oczekiwany format zestawu danych (`.pt`):

- **Pretrening (MLM)**: `TensorDataset` zawierający tensory: `input_ids`, `attention_mask`
- **Dostrajanie (CLS)**: `TensorDataset` zawierający tensory: `input_ids`, `attention_mask`, `labels`

2.7.2. Tokenizacja danych

Do tokenizacji wykorzystywany jest wrapper `WordPieceTokenizerWrapper` (szczegóły w sek. 2.2).

Jeśli nie dysponujesz gotowym tokenizerem (plik `vocab.txt`), możesz go wytrenować na własnym korpusie tekstowym:

```

1 from textc4f_transformer.tokenizer.wordpiece_tokenizer_wrapper \
2     import WordPieceTokenizerWrapper
3
4 tokenizer = WordPieceTokenizerWrapper()
5 tokenizer.train(tokenizer_dir="my_tokenizer_dir", input="data/raw/input.txt")
  
```

Listing 2.3: Trening tokenizera

```

1 from textcltf_transformer.tokenizer.wordpiece_tokenizer_wrapper \
2     import WordPieceTokenizerWrapper
3 import torch
4 import pathlib
5
6 tokenizer = WordPieceTokenizerWrapper()
7 tokenizer.load("src/textcltf_transformer/tokenizer/my_tokenizer_dir")
8
9 # Użycie encode z plikiem tekstowym
10 ds = tokenizer.encode(
11     input="data/raw/text_input.txt",
12     # labels=labels_tensor, # opcjonalne
13     max_length=512,
14 )
15
16 out = pathlib.Path("data/tokenized/train_dataset.pt")
17 out.parent.mkdir(parents=True, exist_ok=True)
18 torch.save(ds, out)

```

Listing 2.4: Tokenizacja z pliku

```

1 import pandas as pd
2
3 df = pd.read_csv("data/raw/data.csv")
4 ds = tokenizer.encode_pandas(
5     df=df,
6     text_col="text",
7     label_col="label", # opcjonalne
8     max_length=512
9 )

```

Listing 2.5: Tokenizacja z ramki danych Pandas

2.7.3. Konfiguracja eksperymentów

Katalog `experiments/` służy do definiowania eksperymentów i ich konfiguracji. Każdy podkatalog w `pretraining/` lub `finetuning/` reprezentuje pojedynczy, powtarzalny przebieg eksperymentu. Szablony plików konfiguracyjnych `pretraining.yaml` i `finetuning.yaml` przechowywane są w `experiments/config_templates/`, na ich podstawie generowane są pliki `config.yaml` w odpowiednich katalogach eksperymentów. Szczegóły dotyczące konfiguracji eksperymentów są opisane w sek. 2.5.

Generowanie eksperymentów pretreningu i dostrajania

```
1 python experiments/generate_pretraining_experiment.py -p <pre_name>
```

Listing 2.6: Generowanie eksperymentu pretreningu

```
1 python experiments/generate_finetuning_experiment.py \
2     -f <fin_name> -p <pre_name>
```

Listing 2.7: Generowanie eksperymentu dostrajania

Wynikami są pliki konfiguracyjne `config.yaml` w odpowiednich katalogach eksperymentów – odpowiednio: `experiments/pretraining/<pre_name>/config.yaml`, `experiments/finetuning/<fin_name>/config.yaml`.

Wznawianie pretreningu

```
1 python experiments/generate_pretraining_experiment.py \
2     -p <pre_name> -rp <resume_pretraining_name>
```

Listing 2.8: Generowanie eksperymentu pretreningu (wznawianego z wcześniejszego eksperymentu).

Skrypt automatycznie kopiuje plik konfiguracyjny `config.yaml` i aktualizuje sekcję `training.resume`:

- Ustawia flagę `is_resume` na `true`.
- Przypisuje nazwę wznawianego eksperymentu do `resume_pretraining_name`.
- Ustawia ścieżkę `checkpoint_path` na ostatni zapisany model (`model.ckpt`).

W zależności od celu wznawiania, należy zweryfikować i ewentualnie dostosować parametr `load_only_model_state`:

- **Kontynuacja przerwane go treningu:** Ustaw `false`, aby wczytać pełny stan (model, optymalizator, scheduler, skaler).
- **Transfer learning / TAPT:** Ustaw `true`, aby wczytać wyłącznie wagi modelu.

Dodatkowo, w razie potrzeby można ręcznie zmienić ścieżkę do punktu kontrolnego, np. na `best-model.ckpt`.

2.7.4. Trening

Głównym interfejsem do uruchamiania treningu jest skrypt `train.py`.

```
1 python train.py -n <pre_name> -m pretraining
```

Listing 2.9: Uruchomienie pretreningu

```
1 python train.py -n <fin_name> -m finetuning
```

Listing 2.10: Uruchomienie dostrajania

Pliki generowane w folderze eksperymentu

Po skończeniu treningu w folderze eksperymentu znajdują się następujące pliki:

- **Zapisane stany: checkpoints/**
 - `best-model.ckpt` – najlepszy model (pełny stan z optymalizatorem/schedulerem/skalerem)
 - `model.ckpt` – model końcowy (tylko wagi)
- **Metryki CSV:** `metrics/train/metrics.csv`, `metrics/eval/metrics.csv` (gdy `logging.log_metrics_csv=True`)
- **Logowanie W&B:** metadane i artefakty przechowywane w katalogu `wandb/` (gdy `logging.use_wandb=True`)

3. Opis i implementacja mechanizmów uwagi

3.1. SDPA

W Transformerze podstawową operacją jest wielogłówna samouwaga (*Multi-Head Self Attention*) realizowana przez SDPA (*Scaled Dot-Product Attention*). Dzięki temu mechanizmowi każda pozycja sekwencji buduje kontekstową reprezentację, która następnie jest wykorzystywana w kolejnych warstwach modelu.

3.1.1. Samouwaga (ang. *Self Attention*)

Niech $z = (z^1, \dots, z^N)$ oznacza sekwencję N elementów (tokenów), gdzie $z^i \in \mathbb{R}^D$ jest wektorem reprezentacji i -tego elementu. Wagi uwagi A_{ij} są wyznaczone na podstawie parowej miary podobieństwa pomiędzy dwiema pozycjami sekwencji, tj. pomiędzy reprezentacją zapytania (query) q^i dla elementu i oraz reprezentacją klucza (key) k^j dla elementu j .

- $N \in \mathbb{N}$ – długość sekwencji.
- $D \in \mathbb{N}$ – wymiar wejściowej reprezentacji (embeddingu).
- $d_k \in \mathbb{N}$ – wymiar przestrzeni zapytań/kluczy oraz wartości.
- $z \in \mathbb{R}^{N \times D}$ – macierz, w której i -ty wiersz odpowiada z^i .

Niech $U_{qkv} \in \mathbb{R}^{D \times 3d_k}$ będzie macierzą parametrów (uczoną), która realizuje jednoczesną projekcję wejścia do przestrzeni zapytań, kluczy i wartości. Definiujemy:

$$[Q, K, V] = z U_{qkv}, \quad (3.1)$$

gdzie $Q, K, V \in \mathbb{R}^{N \times d_k}$, a zapis $[Q, K, V]$ oznacza konkatencję bloków macierzy wzdłuż wymiaru kolumn.

Macierz wag uwagi $A \in \mathbb{R}^{N \times N}$ wyznaczamy jako:

$$A = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right), \quad A \in \mathbb{R}^{N \times N}. \quad (3.2)$$

3.1. SDPA

Operator uwagi dla wejścia z definiujemy jako ważoną kombinację wartości:

$$SA(z) = AV \in \mathbb{R}^{N \times d_k}. \quad (3.3)$$

Powyższe dwa kroki można zapisać jedną definicją – *Scaled Dot-Product Attention*, SDPA:

$$\text{SDPA}(Q, K, V) = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V. \quad (3.4)$$

3.1.2. Wielogłówna samouwaga (ang. *Multihead Self Attention*)

Wielogłówna samouwaga (*Multihead Self-Attention*, MSA) stanowi uogólnienie operatora samouwagi SA polegające na równoległym uruchomieniu h niezależnych mechanizmów samouwagi, zwanych głowami (ang. *heads*), a następnie na złączeniu (konkatenacji) ich wyników i przekształceniu liniowym do wymiaru modelu.

- $h \in \mathbb{N}$ – liczba głów uwagi.
- Przyjmujemy, że $h \mid D$ oraz definiujemy wymiar pojedynczej głowy jako

$$d_k = \frac{D}{h}. \quad (3.5)$$

- Dla każdej głowy $\ell \in \{1, \dots, h\}$ operator $SA_\ell(z) \in \mathbb{R}^{N \times d_k}$ oznacza samowagę obliczaną analogicznie jak w przypadku pojedynczej głowy.

Wynik wielogłownej samouwagi definiujemy jako:

$$\text{MSA}(z) = [SA_1(z); SA_2(z); \dots; SA_h(z)] U_{\text{out}}, \quad (3.6)$$

gdzie $[\cdot; \cdot]$ oznacza konkatenację wzdłuż wymiaru cech (column), zatem

$$[SA_1(z); \dots; SA_h(z)] \in \mathbb{R}^{N \times (hd_k)} = \mathbb{R}^{N \times D}, \quad (3.7)$$

natomiast macierz wyjściowej projekcji spełnia:

$$U_{\text{out}} \in \mathbb{R}^{D \times D}. \quad (3.8)$$

3.1.3. Flash Attention

W ramach realizacji celów projektowych, biblioteka została wyposażona w autorską implementację mechanizmu *Scaled Dot-Product Attention* (SDPA) – opisanego powyżej – zgodną z oryginalnym transformerem z 2017 roku [14]. Implementacja ta ma walor edukacyjny i demonstracyjny, pozwalając na pełną transparentność obliczeń. Posiada ona jednak charakter naiwny – wymaga

obliczania pełnej macierzy uwagi o wymiarach $N \times N$, co skutkuje kwadratową złożonością pamięciową $O(N^2)$ i brakiem niskopoziomowych optymalizacji dla jąderek CUDA.

W kontekście planowanych badań (szczegóły w sek. 4), na zbiorach o długich sekwencjach (Hyperpartisan: 4096 tokenów, ArXiv: 16384 tokenów), wykorzystanie naiwnej implementacji okazuje się niemożliwe ze względu na ograniczenia pamięciowe akceleratorów oraz nieakceptowalny czas treningu.

Aby umożliwić przeprowadzenie eksperymentów w rozsądnym czasie oraz zapewnić rzetelny punkt odniesienia, doimplementowano obsługę natywnej funkcji biblioteki PyTorch (`torch.nn.functional.scaled_dot_product_attention`). Implementacja ta automatycznie dobiera backend obliczeń w zależności od dostępnego sprzętu oraz własności danych i – gdy spełnione są wymagane warunki – może wykorzystywać zoptymalizowane algorytmy, takie jak FlashAttention [4]. W praktyce przekłada się to na następujące korzyści:

- **Niższe zużycie pamięci pośredniej:** FlashAttention nie materializuje jawnie pełnej macierzy uwagi o rozmiarze $N \times N$, dzięki czemu złożoność pamięciowa (dla pośrednich wyników) może spaść z $O(N^2)$ do $O(N)$ względem długości sekwencji N . Złożoność obliczeniowa pozostaje $O(N^2)$, ponieważ liczba operacji wynikająca z iloczynów skalarnych między elementami sekwencji nie ulega zmianie.
- **Lepsza lokalność pamięci:** algorytm wykorzystuje kafelkowanie (ang. *tiling*) oraz obliczenia blokowe, co ogranicza koszt transferów do pamięci globalnej GPU i poprawia wykorzystanie pamięci podręcznej.
- **Wysoka wydajność na GPU:** dzięki redukcji ruchu pamięci oraz fuzji operacji w wyspecjalizowanych kernelach, metoda osiąga wysoki stopień wykorzystania zasobów GPU, co zazwyczaj skutkuje znacznym przyspieszeniem obliczeń.

W związku z powyższym, w opisanych w dalszej części pracy eksperymentach (sek. 4), wykorzystujemy tę zoptymalizowaną, natywną implementację (Flash Attention). Pozwala to na traktowanie wyników SDPA jako silnego, przemysłowego punktu odniesienia dla badanych uwagi przybliżonych (LSH i FAVOR+).

3.2. LSH

Locality-Sensitive Hashing attention jest przybliżeniem pełnej uwagi, które redukuje liczbę porównań przez ograniczenie uwagi do elementów podobnych — identyfikowanych za pomocą haszowania.

3.2. LSH

Zamiast liczyć podobieństwo $q_i \cdot k_j$ dla wszystkich j , najpierw haszujemy wektory (zapytania i klucze) tak, by wektory podobne (bliskie kątowno) trafiały do tego samego koszyka – realizujemy to przez losowe projekcje. W efekcie każdy token otrzymuje numer koszyka, a uwagę liczymy tylko w obrębie swojego koszyka. Dzięki temu złożoność obliczeń obniża się z $O(N^2)$ do $O(N \log N)$, a złożoność pamięciowa z $O(N^2)$ do $O(N)$.

3.2.1. Konstrukcja LSH

Wprowadźmy oznaczenie \mathcal{P}_i jako zbioru elementów, do których zapytanie w pozycji i kieruje uwagę.

Definiujemy maskę jako:

$$m(j, \mathcal{P}_i) = \begin{cases} \infty & \text{jeśli } j \notin \mathcal{P}_i, \\ 0 & \text{w przeciwnym razie.} \end{cases} \quad (3.9)$$

Niech z oznacza wyraz normalizujący w softmaksie, to znaczy:

$$z(i, \mathcal{P}_i) = \log \sum_{j \in \mathcal{P}_i} \exp(q_i \cdot k_j). \quad (3.10)$$

Wtedy dla elementu i , $Attention(i)$ można zapisać jako:

$$o_i = \sum_{j=0}^{N-1} \exp(q_i \cdot k_j - m(j, \mathcal{P}_i) - z(i, \mathcal{P}_i)) v_j. \quad (3.11)$$

Dla przejrzystości pomijamy skalowanie przez $\sqrt{d_k}$.

Teraz przechodzimy do uwagi LSH. Aby uzyskać b koszyków, najpierw ustalamy losową macierz R o rozmiarze $[d_k, b/2]$, to znaczy Q, K, V (wiersze to tokeny, kolumny to cechy) $R \in \mathbb{R}^{d_k \times (b/2)}$, $R_{ij} \sim \mathcal{N}(0, \frac{1}{d_k})$, niezależnie dla wszystkich i, j . Następnie definiujemy funkcję haszującą (której wartość oznacza numer koszyka) jako: $h(x) = \arg \max([xR; -xR])$, gdzie $[u; v]$ oznacza konkatencję dwóch wektorów. Wtedy zbiór \mathcal{P}_i przyjmuje postać:

$$\mathcal{P}_i = \left\{ j : h(q_i) = h(k_j) \right\}, \quad (3.12)$$

co oznacza że element i zwraca uwagę tylko na elementy z tego samego koszyka.

Liczba zapytań i kluczy w danym koszyku może się różnić — czasem koszyk może zawierać wiele zapytań, ale brak kluczy. Aby rozwiązać ten problem, zapewniamy, że $h(k_j) = h(q_j)$ poprzez ustawienie $K = Q$, czyli wykorzystujemy tę samą macierz zarówno dla zapytań (Q), jak i kluczy (K). Typowe implementacje Transformera pozwalają pozycji zwracać uwagę na samą siebie. Takie zachowanie jest niepożądane w przypadku wspólnej reprezentacji Q i K , ponieważ iloczyn skalarny wektora zapytania z samym sobą prawie zawsze będzie większy niż iloczyn skalarny

tego wektora z wektorem z innej pozycji. Dlatego modyfikujemy maskowanie w taki sposób, aby zabronić tokenowi zwracania uwagi na samego siebie, to znaczy $m(j, \mathcal{P}_i) = \infty$ jeśli $j \notin \mathcal{P}_i$ albo $i = j$. Po zastosowaniu funkcji haszującej, sortujemy zapytania według numeru koszyka oraz pozycji w sekwencji; to definiuje permutację $i \mapsto s_i$ po sortowaniu.

Aby zmniejszyć liczbę obliczeń, uwagę obliczamy blokowo — to znaczy C kolejnych zapytań (po sortowaniu) liczy uwagę względem swojego oraz sąsiednich bloków, gdzie C to rozmiar bloku.

Dla elementu i , zbiór elementów do których będzie on miał fizyczny dostęp w trakcie liczenia uwagi możemy zapisać jako:

$$\tilde{\mathcal{P}}_i = \left\{ j : \left\lfloor \frac{s_i}{C} \right\rfloor - 1 \leq \left\lfloor \frac{s_j}{C} \right\rfloor \leq \left\lfloor \frac{s_i}{C} \right\rfloor + 1 \right\}, \quad (3.13)$$

gdzie $\left\lfloor \frac{s_j}{C} \right\rfloor$ to numer bloku w którym znajduje się element j .

Wtedy zbiór \mathcal{P}_i możemy zapisać jako

$$\mathcal{P}_i = \tilde{\mathcal{P}}_i \cap \left\{ j : h(q_i) = h(q_j) \right\}. \quad (3.14)$$

Dzięki temu możemy zapisać:

$$o_i = \sum_{j \in \tilde{\mathcal{P}}_i} \exp \left(q_i \cdot k_j - m(j, \mathcal{P}_i) - z(i, \mathcal{P}_i) \right) v_j. \quad (3.15)$$

W praktyce ustawiamy $n_{\text{buckets}} = \frac{N}{C}$, co oznacza że liczba koszyków jest równa liczbie bloków.

3.2.2. Wielorundowa uwaga LSH (ang. *Multi-round LSH Attention*)

Aby zmniejszyć szansę, że podobne elementy trafią do różnych koszyków, wykonujemy kilka rund haszowania z różnymi funkcjami skrótu $h^{(1)}, h^{(2)}, \dots, h^{(n_{\text{rounds}})}$.

Definiujemy:

$$\tilde{\mathcal{P}}_i^{(r)} = \left\{ j : \left\lfloor \frac{s_i^{(r)}}{C} \right\rfloor - 1 \leq \left\lfloor \frac{s_j^{(r)}}{C} \right\rfloor \leq \left\lfloor \frac{s_i^{(r)}}{C} \right\rfloor + 1 \right\}, \quad (3.16)$$

$$\mathcal{P}_i^{(r)} = \tilde{\mathcal{P}}_i^{(r)} \cap \left\{ j : h^{(r)}(q_i) = h^{(r)}(q_j) \right\}. \quad (3.17)$$

W oryginalnej pracy zastosowano:

$$U_i = \bigcup_{r=1}^{n_{\text{rounds}}} \mathcal{P}_i^{(r)} \quad , \quad \tilde{U}_i = \bigcup_{r=1}^{n_{\text{rounds}}} \tilde{\mathcal{P}}_i^{(r)}, \quad (3.18)$$

oraz obliczono o_i jako:

$$o_i = \sum_{j \in \tilde{U}_i} \exp \left(q_i \cdot k_j - m(j, U_i) - z(i, U_i) \right) v_j, \quad (3.19)$$

czyli element i zwraca uwagę tylko na te elementy które trafiły do tego samego koszyka w którejkolwiek z rund haszowania.

3.2. LSH

W naszej pracy zmodyfikowaliśmy to w następujący sposób:

$$o_i = \frac{1}{n_{\text{rounds}}} \sum_{r=1}^{n_{\text{rounds}}} \sum_{j \in \tilde{\mathcal{P}}_i^{(r)}} \exp \left(q_i \cdot k_j - m(j, \mathcal{P}_i^{(r)}) - z(i, \mathcal{P}_i^{(r)}) \right) v_j. \quad (3.20)$$

Czyli dla każdej rundy haszowania obliczamy uwagę w standardowy sposób, a następnie uśredniamy.

Zdefiniujmy:

$$w_{i,j} = \frac{1}{n_{\text{rounds}}} \sum_{r=1}^{n_{\text{rounds}}} \exp \left(q_i \cdot k_j - m(j, \mathcal{P}_i^{(r)}) - z(i, \mathcal{P}_i^{(r)}) \right). \quad (3.21)$$

Wtedy o_i możemy zapisać jako:

$$\begin{aligned} o_i &= \sum_{j \in U_i} \frac{1}{n_{\text{rounds}}} \sum_{r=1}^{n_{\text{rounds}}} \exp \left(q_i \cdot k_j - m(j, \mathcal{P}_i^{(r)}) - z(i, \mathcal{P}_i^{(r)}) \right) v_j \\ &= \sum_{j \in U_i} w_{i,j} v_j. \end{aligned} \quad (3.22)$$

W praktyce odpowiada to średniej ważonej, w której wagi $w_{i,j}$ są miarą podobieństwa pomiędzy elementami i oraz j , to znaczy, im w większej liczbie rund j znajdzie się w tym samym koszyku co i (czyli im bardziej j jest podobny do i), tym większy jest $w_{i,j}$, a co za tym idzie — większy wkład wektorów v_j w końcowy wektor uwagi o_i .

3.2.3. Maska uwagi w bloku

Dla elementu i , zamiast ograniczać liczenie uwagi o_i jedynie do elementów należących do tego samego koszyka, w bloku, do którego należy i , oraz sąsiednich bloków, można rozszerzyć uwagę na wszystkie elementy znajdujące się w tych blokach — niezależnie od przynależności do koszyka. W praktyce sprowadza się to do usunięcia maski pomiędzy elementami z różnych koszyków w obrębie tego samego lub sąsiednich bloków. Odpowiada to modyfikacji wzoru na uwagę do postaci:

$$o_i = \frac{1}{n_{\text{rounds}}} \sum_{r=1}^{n_{\text{rounds}}} \sum_{j \in \tilde{\mathcal{P}}_i^{(r)}} \exp \left(q_i \cdot k_j - z(i, \tilde{\mathcal{P}}_i^{(r)}) \right) v_j. \quad (3.23)$$

Niech macierz A_S oznacza macierz A ograniczoną do wierszy o indeksach należących do zbioru S . Zdefiniujmy również zbiór elementów należących do tego samego bloku co i :

$$\hat{\mathcal{P}}_i^{(r)} = \left\{ j : \left\lfloor \frac{s_i^{(r)}}{m} \right\rfloor = \left\lfloor \frac{s_j^{(r)}}{m} \right\rfloor \right\}. \quad (3.24)$$

Wtedy nasza modyfikacja odpowiada obliczeniu standardowej uwagi (dla elementów z bloku $\hat{\mathcal{P}}_i^{(r)}$) w ograniczonym zakresie:

$$\text{Attention}(Q', K', V') = \text{Softmax} \left(\frac{Q' K'^T}{\sqrt{d_k}} \right) V', \quad (3.25)$$

gdzie

$$Q' = Q_{\tilde{\mathcal{P}}_i^{(r)}} \in \mathbb{R}^{|\hat{\mathcal{P}}_i^{(r)}| \times d_k}, \quad K' = K_{\tilde{\mathcal{P}}_i^{(r)}} \in \mathbb{R}^{|\tilde{\mathcal{P}}_i^{(r)}| \times d_k}, \quad V' = V_{\tilde{\mathcal{P}}_i^{(r)}} \in \mathbb{R}^{|\tilde{\mathcal{P}}_i^{(r)}| \times d_v}. \quad (3.26)$$

TODO: Wpływ tej modyfikacji na uczenie – wykres jest na prezentacji seminarium – dać to gdzieś do eksperymentów i w tym miejscu się do tego odwołać

3.3. FAVOR+ Attention

3.3.1. Wstęp i motywacja

Klasyczny mechanizm *scaled dot-product attention* dla sekwencji długości N operuje na macierzach $Q, K, V \in \mathbb{R}^{N \times d}$ i wykorzystuje macierz wag uwagi

$$A \in \mathbb{R}^{N \times N}, \quad A_{ij} = \exp(q_i^\top k_j). \quad (3.27)$$

Wyjście attention można zapisać jako

$$\text{Att}(Q, K, V) = D^{-1} \begin{pmatrix} AV \end{pmatrix}, \quad D = \text{diag} \begin{pmatrix} A \mathbf{1}_N \end{pmatrix}. \quad (3.28)$$

Główną wadą tej postaci jest konieczność jawnego zbudowania macierzy A rozmiaru $N \times N$, co prowadzi do złożoności czasowej rzędu $O(N^2 d)$ (mnożenie AV) oraz pamięciowej $O(N^2)$ na same wagi uwagi. Metoda FAVOR+ (*Fast Attention Via positive Orthogonal Random features*) reinterpretuje uwagę jako problem aproksymacji funkcji jądra i eliminuje potrzebę konstruowania A .

3.3.2. Uwaga jako kernel i mapowanie cech losowych

Zauważmy, że elementy A_{ij} są wartościami dodatniego jądra

$$K(x, y) = \exp(x^\top y), \quad x, y \in \mathbb{R}^d. \quad (3.29)$$

FAVOR+ zakłada istnienie losowego odwzorowania cech $\phi_\omega : \mathbb{R}^d \rightarrow \mathbb{R}^m$ takiego, że jądro można zapisać jako wartość oczekiwaną iloczynu skalarnego cech:

$$K(x, y) = \mathbb{E}_\omega [\phi_\omega(x) \phi_\omega(y)]. \quad (3.30)$$

W praktyce zastępujemy wartość oczekiwaną średnią. Dla niezależnych próbek $\omega_1, \dots, \omega_m \stackrel{i.i.d.}{\sim} p(\omega)$ definiujemy

$$\widehat{K}_m(x, y) := \frac{1}{m} \sum_{i=1}^m \phi_{\omega_i}(x) \phi_{\omega_i}(y), \quad \text{wtedy} \quad K(x, y) \approx \widehat{K}_m(x, y). \quad (3.31)$$

3.3. FAVOR+ ATTENTION

Definiujemy macierze cech dla całej sekwencji:

$$Q' \in \mathbb{R}^{N \times m}, \quad K' \in \mathbb{R}^{N \times m}, \quad \text{gdzie wiersze to } (Q')_i = \phi(q_i)^\top, \quad (K')_j = \phi(k_j)^\top. \quad (3.32)$$

Wówczas macierz wag uwagi można przybliżyć przez

$$A_{ij} = K(q_i, k_j) \approx \phi(q_i)^\top \phi(k_j), \quad \Rightarrow \quad A \approx Q'(K')^\top. \quad (3.33)$$

Po podstawieniu do wzoru na attention otrzymujemy efektywną postać obliczeń z zachowaniem kolejności mnożeń:

$$\widehat{\text{Att}}(Q, K, V) = \widehat{D}^{-1} \left(Q' \left((K')^\top V \right) \right), \quad \widehat{D} = \text{diag} \left(Q' \left((K')^\top \mathbf{1}_N \right) \right). \quad (3.34)$$

Kluczowe jest to, że nie tworzymy jawnie macierzy $N \times N$. Zamiast tego wykonujemy dwa mnożenia macierzy o rozmiarach $m \times N$ i $N \times m$, co daje złożoność czasową $O(Nmd)$ oraz pamięciową $O(Nm + Nd + md)$, zamiast odpowiednio $O(N^2d)$ i $O(N^2 + Nd)$ w standardowym attention.

Lemat 3.1. Jeśli $\omega \sim \mathcal{N}(0, I_d)$ oraz $\phi_\omega(x) = \exp(\omega^\top x - \frac{1}{2}\|x\|^2)$, to

$$\exp(x^\top y) = \mathbb{E}_\omega [\phi_\omega(x) \phi_\omega(y)]. \quad (3.35)$$

DOWÓD: Korzystamy z faktu, że to zmienna $\omega^\top z$ ma rozkład:

$$\omega^\top z \sim \mathcal{N}(0, \|z\|^2). \quad (3.36)$$

Dla zmiennej normalnej $g \sim \mathcal{N}(0, \sigma^2)$ funkcja generująca momenty (MGF) ma postać:

$$M_g(t) = \mathbb{E}[\exp(tg)] = \exp\left(\frac{1}{2}\sigma^2 t^2\right). \quad (3.37)$$

Podstawiając $t = 1$ i $\sigma^2 = \|z\|^2$, otrzymujemy kluczowy fakt:

$$\mathbb{E}_\omega[\exp(\omega^\top z)] = \exp\left(\frac{1}{2}\|z\|^2\right). \quad (3.38)$$

Wtedy:

$$\begin{aligned} \mathbb{E}_\omega[\phi_\omega(x) \phi_\omega(y)] &= \mathbb{E}[\exp(\omega^\top (x + y))] \cdot \exp\left(-\frac{1}{2}(\|x\|^2 + \|y\|^2)\right) \\ &= \exp\left(\frac{1}{2}\|x + y\|^2\right) \exp\left(-\frac{1}{2}(\|x\|^2 + \|y\|^2)\right) \\ &= \exp\left(\frac{1}{2}(\|x + y\|^2 - \|x\|^2 - \|y\|^2)\right) \\ &= \exp(x^\top y). \end{aligned} \quad (3.39)$$

□

3.3.3. Implementacja

Estymator cosh

Zauważmy, że:

$$\exp(t) = \cosh(t) + \sinh(t), \quad (3.40)$$

gdzie \cosh jest funkcją parzystą, a \sinh nieparzystą. Ponieważ rozkład $\omega^\top(x+y)$ jest symetryczny względem zera, mamy:

$$\mathbb{E}[\sinh(\omega^\top(x+y))] = 0. \quad (3.41)$$

Zatem:

$$\mathbb{E}[\exp(\omega^\top(x+y))] = \mathbb{E}[\cosh(\omega^\top(x+y))]. \quad (3.42)$$

Otrzymujemy:

$$\exp(x^\top y) = \mathbb{E}_\omega [\cosh(\omega^\top(x+y))] \cdot \exp\left(-\frac{1}{2}(\|x\|^2 + \|y\|^2)\right). \quad (3.43)$$

Porównanie wariancji

Mając zdefiniowane dwa nieobciążone estymatory możemy porównać ich wariancję:

$$\hat{K}_{\text{exp}} = \exp(\omega^\top(q+k)) \exp\left(-\frac{1}{2}(\|q\|^2 + \|k\|^2)\right), \quad (3.44)$$

$$\hat{K}_{\text{cosh}} = \cosh(\omega^\top(q+k)) \exp\left(-\frac{1}{2}(\|q\|^2 + \|k\|^2)\right). \quad (3.45)$$

$$\begin{aligned} \mathbb{E}[\hat{K}_{\text{exp}}^2] &= \mathbb{E}[\exp(2\omega^\top(q+k))] \exp\left(-(\|q\|^2 + \|k\|^2)\right) \\ &= \exp(2\|q+k\|^2) \exp\left(-(\|q\|^2 + \|k\|^2)\right). \end{aligned} \quad (3.46)$$

Korzystamy z zależności:

$$\cosh^2(x) = \frac{1 + \cosh(2x)}{2}. \quad (3.47)$$

Wówczas:

$$\begin{aligned} \mathbb{E}[\hat{K}_{\text{cosh}}^2] &= \frac{1 + \mathbb{E}[\cosh(2\omega^\top(q+k))]}{2} \exp(-(\|q\|^2 + \|k\|^2)) \\ &= \frac{1 + \exp(2\|q+k\|^2)}{2} \exp(-(\|q\|^2 + \|k\|^2)). \end{aligned} \quad (3.48)$$

Porównujemy teraz wariancję:

$$\begin{aligned} \Delta_{\text{Var}} &= \text{Var}(\hat{K}_{\text{cosh}}) - \text{Var}(\hat{K}_{\text{exp}}) \\ &= \mathbb{E}[\hat{K}_{\text{cosh}}^2] - \mathbb{E}[\hat{K}_{\text{exp}}^2] \\ &= \exp(-(\|q\|^2 + \|k\|^2)) \frac{1 - \exp(2\|q+k\|^2)}{2} < 0 \\ &\implies \text{Var}(\hat{K}_{\text{cosh}}) < \text{Var}(\hat{K}_{\text{exp}}). \end{aligned} \quad (3.49)$$

Estymator exp

W implementacji Performera wykorzystano następującą funkcję ϕ :

$$\phi(x) = \frac{1}{\sqrt{m}} \exp\left(-\frac{1}{2}\|x\|^2\right) \begin{bmatrix} \exp(\omega_1^\top x) \\ \vdots \\ \exp(\omega_m^\top x) \end{bmatrix}, \quad \omega_i \sim \mathcal{N}(0, I), \quad (3.50)$$

wtedy

$$\phi(q)^\top \phi(k) = \frac{1}{m} \exp\left(-\frac{1}{2}(\|q\|^2 + \|k\|^2)\right) \sum_{i=1}^m \left[\exp(\omega_i^\top (q+k)) \right] \quad (3.51)$$

$$\approx \exp(q^\top k) \quad , \text{ dla dużego } m \quad (3.52)$$

Modyfikacja

W naszej pracy używamy modyfikacji oryginalnego estymatora exp:

$$\phi(x) = \frac{1}{\sqrt{2m}} \exp\left(-\frac{1}{2}\|x\|^2\right) \begin{bmatrix} \exp(\omega_1^\top x) \\ \vdots \\ \exp(\omega_m^\top x) \\ \exp(-\omega_1^\top x) \\ \vdots \\ \exp(-\omega_m^\top x) \end{bmatrix}, \quad \omega_i \sim \mathcal{N}(0, I). \quad (3.53)$$

Wówczas:

$$\begin{aligned} \phi(q)^\top \phi(k) &= \frac{1}{2m} \exp\left(-\frac{1}{2}(\|q\|^2 + \|k\|^2)\right) \sum_{i=1}^m \left[\exp(\omega_i^\top (q+k)) + \exp(-\omega_i^\top (q+k)) \right] \\ &= \frac{1}{m} \exp\left(-\frac{1}{2}(\|q\|^2 + \|k\|^2)\right) \sum_{i=1}^m \cosh(\omega_i^\top (q+k)) \\ &\approx \exp(q^\top k) \quad , \text{ dla dużego } m. \end{aligned} \quad (3.54)$$

W ten sposób otrzymujemy estymator o mniejszej wariancji niż oryginalny.

$$\text{Var}(\phi(q)^\top \phi(k)) = \frac{1}{m} \text{Var}(\hat{K}_{\cosh}) < \frac{1}{m} \text{Var}(\hat{K}_{\exp}). \quad (3.55)$$

Uwaga: Należy zauważyć, że przy użyciu metody cosh macierze Q' oraz K' rosną dwukrotnie, tzn. $Q' \in \mathbb{R}^{N \times 2m}$, $K' \in \mathbb{R}^{N \times 2m}$, a co za tym idzie rośnie złożoność obliczeniowa. Aby zachować ten sam budżet obliczeniowy, rozmiar próbki w przypadku estymatora cosh redukujemy do $m/2$, podczas gdy dla estymatora exp wynosi on m .

Porównujemy wariancje estymatorów:

$$\begin{aligned}
\Delta_{\text{Var}} &= \frac{1}{m/2} \text{Var}(\hat{K}_{\text{cosh}}) - \frac{1}{m} \text{Var}(\hat{K}_{\text{exp}}) \\
&= \frac{2}{m} \left(\mathbb{E}[\hat{K}_{\text{cosh}}^2] - \mu^2 \right) - \frac{1}{m} \left(\mathbb{E}[\hat{K}_{\text{exp}}^2] - \mu^2 \right) \\
&= \left(\frac{2}{m} \mathbb{E}[\hat{K}_{\text{cosh}}^2] - \frac{1}{m} \mathbb{E}[\hat{K}_{\text{exp}}^2] \right) - \frac{1}{m} \mu^2 \\
&= \frac{1}{m} \exp \left(-(\|q\|^2 + \|k\|^2) \right) \left[\left(1 + \exp(2\|q + k\|^2) \right) - \exp(2\|q + k\|^2) \right] - \frac{1}{m} \mu^2 \\
&= \frac{1}{m} \left[\exp \left(-(\|q\|^2 + \|k\|^2) \right) - \exp(2q^\top k) \right] \leq 0.
\end{aligned} \tag{3.56}$$

Ponieważ $\|q + k\|^2 \geq 0$, czyli $2q^\top k \geq -(\|q\|^2 + \|k\|^2)$.

Uwaga: W związku z powyższym, parametr, który oznaczamy jako `nb_features` (patrz tab. 2.3) określa liczbę kolumn macierzy Q' i K' (patrz rów. (3.32)), a nie dosłowny rozmiar próbki w sensie rów. (3.53).

4. Eksperymenty

4.1. Dane

4.1.1. Zbiory danych

- **Wikipedia.** Korpus artykułów z anglojęzycznej Wikipedii w wersji ze zrzutu 20231101 (Hugging Face Datasets: `wikimedia/wikipedia`, konfiguracja `20231101.en`). Zbiór wykorzystano do pretreningu modelu w zadaniu (MLM).¹
- **IMDb.** Zbiór recenzji filmowych do binarnej klasyfikacji sentymentu (*neg/pos*).²
- **arXiv.** Zbiór dokumentów naukowych z arXiv do wieloklasowej klasyfikacji tematycznej (11 klas). W wersji `ccdv/arxiv-classification` etykiety odpowiadają kategoriom arXiv: `math.AC`, `cs.CV`, `cs.AI`, `cs.SY`, `math.GR`, `cs.CE`, `cs.PL`, `cs.IT`, `cs.DS`, `cs.NE`, `math.ST`.³
- **Hyperpartisan News Detection.** Zbiór artykułów prasowych do klasyfikacji binarnej: *hyperpartisan* vs *non-hyperpartisan*. W kontekście zadania *hyperpartisan news* definiuje się jako wiadomości prezentujące skrajnie lewicowy lub skrajnie prawicowy punkt widzenia. Wariant `bypublisher` (pliki `articles-*-bypublisher-20181122.xml` oraz `ground-truth-*-bypublisher-20181122.xml`) jest etykietowany na podstawie ogólnego uprzedzenia (obciążenia) wydawcy, zgodnie z ocenami dziennikarzy BuzzFeed lub serwisu MediaBiasFactCheck. Zbiór uczący i walidacyjny są rozdzielone tak, aby wydawcy nie nakładali się między podziałami danych (ang. *splits*).⁴

4.1.2. Przetwarzanie danych

Wikipedia. Z korpusu Wikipedii wybrano podzbiór 600 000 artykułów spełniających kryterium tematyczne: tekst artykułu zawierał co najmniej 3 słowa kluczowe z ustalonej listy. Następnie:

¹<https://huggingface.co/datasets/wikimedia/wikipedia/viewer/20231101.en>

²<https://huggingface.co/datasets/stanfordnlp/imdb>

³<https://huggingface.co/datasets/ccdv/arxiv-classification>

⁴<https://zenodo.org/records/1489920>

- 450 000 artykułów ztokenizowano do maksymalnej długości 128 tokenów⁵,
- pozostałe 150 000 artykułów ztokenizowano do maksymalnej długości 512 tokenów⁵.

Lista słów kluczowych użyta do filtracji artykułów:

film, movie, cinema, television, series, episode, season, actor, actress, director, screenwriter, producer, soundtrack, box, office, award, academy, oscar, review, reception, critics, plot, culture, algorithm, theorem, proof, model, dataset, data, method, approach, analysis, experiment, simulation, complexity, optimization, neural, network, machine, learning, statistics, statistical, physics, quantum, entropy, differential, equation, mathematics, computer, engineering, artificial, intelligence, algorithmic, election, politics, political, party, ideology, government, policy, law, immigration, climate, abortion, gun, control, foreign, propaganda, media, bias, populism, nationalism, controversy, criticism, debate, movement, public, opinion

IMDb. Z treści recenzji usunięto znaczniki HTML odpowiadające nowym liniom (regex: `<br\s*/?>`). Następnie połączono oryginalne podziały `train` i `test` w jeden zbiór, po czym wykonano ponowny podział na zbiory uczący, walidacyjny i testowy w proporcjach 80%/10%/10%:

40 000 (trening) | 5 000 (walidacja) | 5 000 (test).

Dane ztokenizowano do maksymalnej długości 512 tokenów⁵.

arXiv. Nie wykonywano dodatkowego czyszczenia danych. Połączono oryginalne podziały `train`, `validation` i `test`, a następnie zastosowano podział na zbiory uczący, walidacyjny i testowy w proporcjach 80%/10%/10%:

26 710 (trening) | 3 339 (walidacja) | 3 339 (test).

Dane ztokenizowano do maksymalnej długości 16 384 tokenów⁵.

Hyperpartisan News Detection. **TODO: preprocessing + splity** Dane ztokenizowano do maksymalnej długości 4 096 tokenów⁵.

4.1.3. Aspekty danych: źródła, licencje, etyka i bias

Licencje i sposób użycia. Zbiory wykorzystano wyłącznie do trenowania i ewaluacji modeli klasyfikacji w ramach pracy. Treści źródłowe (np. artykuły, recenzje, pełne teksty) nie były

⁵Dane zostały ztokenizowane z wykorzystaniem `WordPieceTokenizerWrapper` (patrz sek. 2.2).

4.1. DANE

cytowane ani w żaden sposób redystrybuowane. Dla Wikipedii zastosowanie mają licencje CC BY-SA i GFDL, natomiast dla pozostałych zbiorów obowiązują warunki wskazane przez ich wydawców oraz metadane repozytoriów.

Etyka, prywatność i bias. Dane pochodzą z treści publicznych, jednak mogą zawierać elementy wrażliwe lub kontrowersyjne (np. tematy polityczne). W szczególności zbiór Hyperpartisan dotyczy wykrywania wiadomości prezentujących skrajnie lewicowy lub skrajnie prawicowy punkt widzenia; w wariancie etykietowania `bypublisher` część etykiet wynika z profilu wydawcy, co może wprowadzać bias i powodować uczenie się sygnałów ubocznych (np. stylu medium) zamiast samej stroniczości treści.

4.1.4. EDA

W niniejszej sekcji przedstawiono statystyki długości tekstów po tokenizacji (zgodnie z sek. 4.1.2) dla poszczególnych zbiorów danych. Statystyki obejmują średnią, odchylenie standardowe, medianę oraz wartości minimalne i maksymalne liczby tokenów. Statystyki zostały przedstawione w tab. 4.1.

Tabela 4.1: Statystyki długości tokenów po tokenizacji

Podział	Statystyki tokenów				
	μ	σ	Med.	Min	Max
<i>Wikipedia</i>					
450k (128)	127	6	128	19	128
150k (512)	452	112	512	20	512
<i>IMDb</i>					
trening	263	137	220	10	512
walidacja	263	138	220	18	512
test	262	137	221	13	512
<i>arXiv</i>					
trening	12 095	4 261	12 836	910	16 384
walidacja	12 055	4 321	12 795	650	16 384
test	12 063	4 342	12 863	1 009	16 384
<i>Hyperpartisan</i>					
trening	2 423	807	2 150	100	4 096
walidacja	2 503	896	2 197	1 255	4 096
test	2 504	899	2 181	1 296	4 096

4.2. Opis eksperymentów

4.2.1. Zarys ogólny eksperymentów

W celu przeprowadzenia pełnego eksperymentu porównawczego zaprojektowano wieloetapowy proces uczenia, który rozpoczyna się od pretreningu na korpusie ogólnym (Wikipedia) z wykorzystaniem zadania modelowania języka z maskowaniem (ang. *Masked Language Modeling*, MLM) w celu nauczenia modelu ogólnych struktur językowych i semantyki, co kończy się utworzeniem zapisanego stanu modelu (ang. *checkpoint*) `model_base.pt`. Następnie realizowana jest adaptacja do zadania (ang. *Task-Adaptive Pretraining*, TAPT [6]), polegająca na kontynuacji uczenia MLM na nieetykietowanych danych z domeny docelowej — w niniejszej pracy wykorzystano do tego celu te same zbiory, które służą do późniejszej klasyfikacji (IMDB, Hyperpartisan, ArXiv), przeprowadzając proces TAPT każdorazowo wyłącznie na zbiorze docelowym startując z modelu bazowego i zapisując wynik w pliku `model_tapt.pt`. Ostatnim etapem jest końcowe dostrajanie (ang. *fine-tuning*), w którym model jest trenowany w sposób nadzorowany na konkretnym, etykietowanym zbiorze danych startując z odpowiedniego punktu kontrolnego TAPT, co pozwala na uzyskanie końcowego modelu `model_finetuned.pt` oraz jego najlepszej wersji `model_finetuned_best.pt`, wyłonionej na podstawie wyników na zbiorze walidacyjnym. Po każdym z etapów następuje transfer wag z odpowiedniego punktu kontrolnego, podczas gdy pozostałe parametry, takie jak stan optymalizatora, harmonogram współczynnika uczenia (ang. *learning rate scheduler*) oraz skaler gradientów (ang. *gradient scaler*) są resetowane.

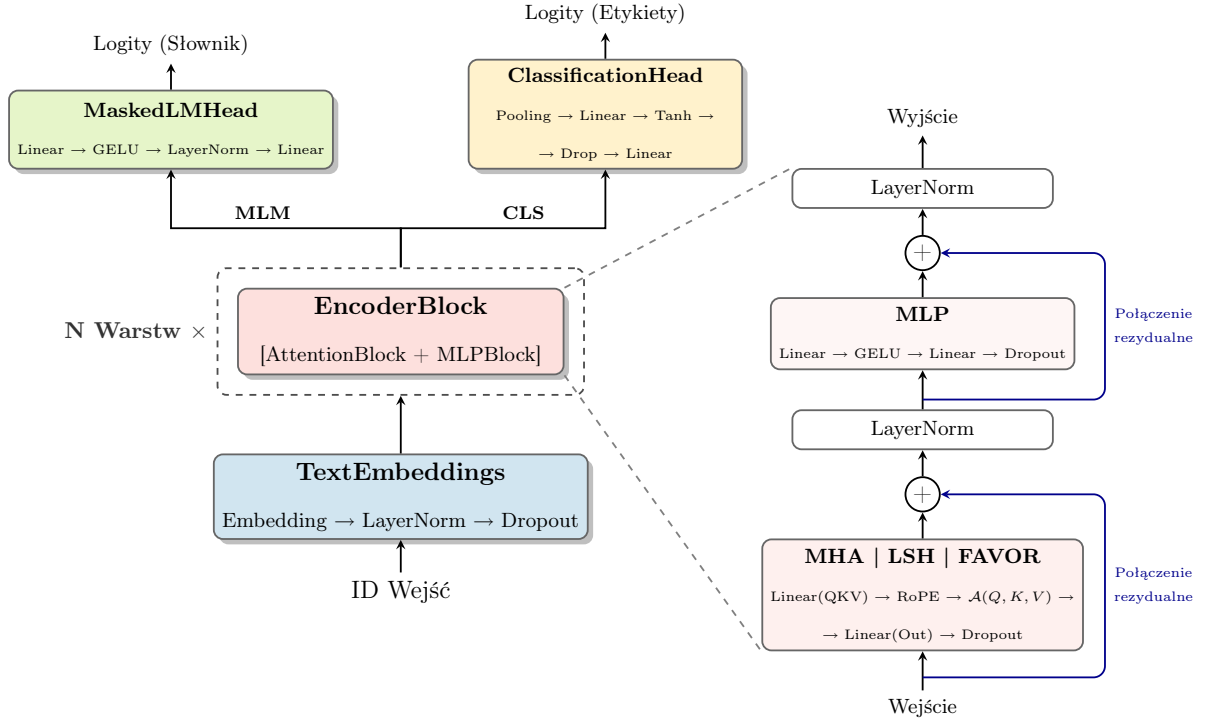
4.2.2. Architektura

Schemat architektury użytej w eksperymentach przedstawiono na rys. 4.1.

Oznaczenia Dla opisu konfiguracji przyjmujemy następujące oznaczenia parametrów: l^t – liczba warstw enkodera, d^h – wymiar ukryty, d^f – rozmiar warstwy pośredniej FFN, h – liczba głowic uwagi, $d^{q|k|v}$ – wymiar przestrzeni zapytań, kluczy i wartości w pojedynczej głowicy. Warto zaznaczyć, że w klasycznym BERT przyjmuje się zazwyczaj $d^q = d^k = d^v = d^h/h$.

BERT_{SMALL} Traktujemy ten model jako standardową małą architekturę modelu BERT. Została ona po raz pierwszy wprowadzona w artykule Turc i in. [13], który pokazuje, że małe modele BERT mogą osiągać bardzo dobre wyniki, jeśli są najpierw porządnie pretrenowane, zamiast polegać wyłącznie na destylacji z dużych modeli. Konfiguracja: $l^t = 4$, $d^h = 512$, $d^f = 2048$, $h = 8$, $d^{q|k|v} = 512$.

4.2. OPIS EKSPERYMENTÓW



Rysunek 4.1: Schemat architektury użytej w eksperymentach (nazewnictwo zgodne z nazwami klas w sek. 2.1)

Liczba parametrów W modelu BERT_{SMALL} liczba parametrów poszczególnych modułów wynosi: 15.6 mln (warstwa osadzeń), 3.15 mln (pojedynczy blok enkodera), 15.9 mln (głowica MLM) oraz 0.26 mln (głowica klasyfikacyjna). Dla architektury AutoTinyBERT_{S1} wartości te wynoszą odpowiednio: 17.2 mln, 2.35 mln, 17.5 mln oraz 0.32 mln. Należy zaznaczyć, że w obu przypadkach ostatnia warstwa liniowa w głowicy MLM współdzieli wagi z warstwą osadzeń. Z tego względu całkowitą liczbę parametrów modelu wyrażamy wzorem:

$$P_{\text{total}} = P_{\text{emb}} + l^t \cdot P_{\text{enc}} + P_{\text{head}}, \quad (4.1)$$

gdzie P_{head} to liczba parametrów trenowalnych danej głowicy. Sumaryczna liczba parametrów dla BERT_{SMALL} wynosi zatem ok. 28.5 mln (zarówno dla MLM i klasyfikacji), a dla AutoTinyBERT_{S1} ok. 29.3 mln (zarówno dla MLM i klasyfikacji).

4.2.3. Hiperparametry

Hiperparametry użyte w poszczególnych etapach treningu są przedstawione w tab. 4.2. Hiperparametry wspólne między wszystkimi etapami treningu i eksperymentami są przedstawione w tab. 4.3. Pozostałe hiperparametry i proces ich optymalizacji jest przedstawiony w sekcjach 4.3,

4.4 i 4.5. **TODO:** poprawić referencje do sekcji Szczegółowy opis użytych parametrów znajduje się w tabelach: 2.1, 2.2, 2.3, 2.4 i 2.6.

Tabela 4.2: Hiperparametry treningu dla poszczególnych etapów uczenia

Etap	Zbiór danych	epochs	learning_rate	min_lr_ratio	max_length	batch_size
Pretrening (MLM)	Wikipedia	10	5×10^{-4}	0.5	512	64
TAPT (MLM)	IMDB	15			512	64
	Hyperpartisan	6	2×10^{-4}	0.5	4096	8
	Arxiv	2			16384	2
Dostrajanie (klasyfikacja)	IMDB	8	3×10^{-5}		512	64
	Hyperpartisan	7	2×10^{-5}	0.2	4096	64*
	Arxiv	4	3×10^{-4}		16384	16*

* Z wykorzystaniem mechanizmu akumulacji gradientów (8 kroków).

Tabela 4.3: Wspólne hiperparametry dla wszystkich etapów treningu

Kategoria	Parametr	Wartość
Architektura	mlp_dropout	0.1
	embedding_dropout	0.1
	pos_encoding	rope
	rope_base	10000
	rope_scale	1.0
	att_dropout	0.0
	att_out_drop	0.1
	projection_bias	true
MLM	tie_mlm_weights	true
	mask_p	0.15
	mask_token_p	0.8
	random_token_p	0.1
Klasyfikacja	pooler_type	bert
Trening	warmup_ratio	0.1
	weight_decay	0.01
	max_grad_norm	1.0
	loss	cross_entropy

4.2.4. Wyniki modelu bazowego (ang. *baseline*)

Jako punkt odniesienia dla eksperymentów z modelami Transformer zaimplementowano prosty klasyfikator oparty na metodzie TF-IDF (*Term Frequency-Inverse Document Frequency*) w połączeniu z regresją logistyczną. Klasyfikator bazowy wykorzystuje następującą konfigurację:

- **TF-IDF:** maksymalnie 20 000 cech, n-gramy (1,2), minimalna częstość dokumentowa 2, maksymalna częstość dokumentowa 0.9,
- **Regresja logistyczna:** solver LBFGS, maksymalnie 1000 iteracji.

Wyniki klasyfikacji (na zbiorach testowych) dla zbioru IMDB przedstawiono w tab. 4.4, dla arXiv w tab. 4.5, dla Hyperpartisan w tab. 4.6.

Tabela 4.4: Wyniki klasyfikatora TF-IDF + LR dla zbioru IMDB

Klasa	Precyzja	Czułość	Miara F1
negative	89.77	89.16	89.46
positive	89.23	89.84	89.54
Dokładność		89.50	
Średnia makro	89.50	89.50	89.50

Tabela 4.5: Wyniki klasyfikatora TF-IDF + LR dla zbioru arXiv

Klasa	Precyzja	Czułość	Miara F1
math.AC	95.24	94.34	94.79
cs.CV	80.00	82.47	81.22
cs.AI	69.01	57.56	62.77
cs.SY	85.12	88.41	86.74
math.GR	94.92	95.73	95.32
cs.CE	78.14	72.96	75.46
cs.PL	88.19	92.95	90.51
cs.IT	86.21	84.75	85.47
cs.DS	87.69	89.62	88.65
cs.NE	74.77	74.43	74.60
math.ST	81.22	87.74	84.35
Dokładność		84.36	
Średnia makro	83.68	83.72	83.62

Tabela 4.6: Wyniki klasyfikatora TF-IDF + LR dla zbioru Hyperpartisan

Klasa	Precyzja	Czułość	Miara F1
not_hyperpartisan	48.60	13.88	21.59
hyperpartisan	49.77	85.32	62.87
Dokładność		49.60	
Średnia makro	49.18	49.60	42.23

4.3. Eksperyment 1

TODO: co chcemy zrobić, z jakimi parametrami, po co - co mierzymy: metryki + czas/ram

4.3.1. Opis eksperymentu

Pierwsza faza eksperymentów koncentruje się na wyznaczeniu optymalnych hiperparametrów procesu dostrajania dla bazowej architektury BERT_{SMALL}, wykorzystującej standardową uwagę SDPA. Przetestowano wpływ trzech czynników: liczby zamrożonych warstw enkodera, wartości dropoutu w głowicy klasyfikacyjnej oraz metody agregacji wektorów (ang. *pooling*). W kontekście pierwszego czynnika, liczba zamrożonych warstw odnosi się do dolnych bloków enkodera. Dla wartości parametru > 0 , wskazane warstwy oraz warstwa osadzeń (ang. *embeddings*) pozostają zamrożone przez pierwsze $\lfloor \text{liczba epok}/2 \rfloor$ epok dostrajania. W przypadku wartości parametru $= 0$, wszystkie parametry modelu są aktualizowane od początku procesu uczenia. Szczegółową przestrzeń poszukiwań (ang. *Grid Search*) przedstawiono w tabeli 4.7.

Wybór metody agregacji jest podyktowany specyfiką przetwarzania długich sekwencji i jest zgodny z procesem tokenizacji (sek. 4.1.2), gdzie co 512 tokenów wstawiany jest token [SEP].

Ze względu na odmienne właściwości poszczególnych zbiorów danych (m.in. różnice w długości sekwencji oraz specyfikę domeny), optymalizacja ta przeprowadzana jest niezależnie (tj. dla każdego zbioru danych osobno). Zestawy optymalnych hiperparametrów, wyłonione na podstawie metryki F_1 -macro uzyskanej na zbiorze walidacyjnym, zostaną wykorzystane dla tych zbiorów danych w kolejnych eksperymentach (sek. 4.4 i 4.5).

4.3.2. Wyniki eksperymentu

Wyniki optymalizacji przedstawione w tabeli 4.9 - będą używane dalej **TODO: ref**

4.4. EKSPERYMENT 2: EWALUACJA UWAGI PRZYBLIŻONYCH (LSH I FAVOR)

Tabela 4.7: Przestrzeń poszukiwań hiperparametrów w Eksperymentcie 1

Hiperparametr	Testowane wartości
Liczba zamrożonych warstw (N_{freeze})	$\{0, 1, 2\}$
Dropout klasyfikatora (P_{drop})	$\{0.1, 0.2\}$
Metoda agregacji*	$\{CLS, \text{Mean}, \text{MeanStep-512}\}$

* CLS i Mean dla IMDB; CLS i MeanStep-512 dla ArXiv i Hyperpartisan.

Tabela 4.8: Wyniki F1-macro na zbiorze testowym dla różnych konfiguracji hiperparametrów.

Model	IMDB	Hyperpartisan	Arxiv
<i>TF-IDF+LR</i>	–	–	–
<i>Agregacja CLS</i>			
$N_{freeze} = 0, P_{drop} = 0.1$	93.22	–	–
$N_{freeze} = 0, P_{drop} = 0.2$	93.22	–	–
$N_{freeze} = 1, P_{drop} = 0.1$	92.95	–	–
$N_{freeze} = 1, P_{drop} = 0.2$	92.67	–	–
$N_{freeze} = 2, P_{drop} = 0.1$	92.97	–	–
$N_{freeze} = 2, P_{drop} = 0.2$	92.93	–	–
<i>Agregacja średnia</i>			
$N_{freeze} = 0, P_{drop} = 0.1$	92.90	–	–
$N_{freeze} = 0, P_{drop} = 0.2$	92.93	–	–
$N_{freeze} = 1, P_{drop} = 0.1$	92.99	–	–
$N_{freeze} = 1, P_{drop} = 0.2$	92.87	–	–
$N_{freeze} = 2, P_{drop} = 0.1$	92.83	–	–
$N_{freeze} = 2, P_{drop} = 0.2$	92.85	–	–

Tabela 4.9: Optymalne hiperparametry warstwy wybrane w fazie E1.

Zbiór danych	N_{freeze}	P_{drop}	Agregacja
IMDB	2	0.2	Mean
Hyperpartisan	–	–	–
ArXiv	–	–	–

4.4. Eksperyment 2: Ewaluacja uwagi przybliżonych (LSH i FAVOR)

4.4.1. Opis eksperymentu

TODO: co chcemy zrobić, z jakimi parametrami, po co - co mierzymy: metryki + czas/ram

Tabela 4.10: Przestrzeń poszukiwań hiperparametrów w Eksperymentcie 2

Uwaga	Hiperparametr	Testowane wartości
LSH	Liczba haszy (N_{hashes})	$\{2, 4\}$
	Wielkość bloku ($Chunk$)	$\{64, 128\}$
FAVOR+	Liczba losowych cech ($N_{features}$)	$\{0.125, 0.25, 0.5, 1.0\} \times d^{q k v}$

Tabela 4.11: Wyniki F1-macro dla różnych konfiguracji mechanizmów uwagi.

Model	IMDB	Hyperpartisan	Arxiv
<i>TF-IDF+LR</i>	–	–	–
<i>LSH</i>			
$N_h=2, C=64$	–	–	–
$N_h=2, C=128$	–	–	–
$N_h=4, C=64$	–	–	–
$N_h=4, C=128$	–	–	–
<i>FAVOR+</i>			
$N_f=0.125$	–	–	–
$N_f=0.25$	–	–	–
$N_f=0.5$	–	–	–
$N_f=1.0$	–	–	–

W drugiej fazie porównujemy warianty uwagi **LSH** (Locality Sensitive Hashing) oraz **FAVOR** (Fast Attention Via positive Orthogonal Random features) zgodnie z tab. 4.10, wykorzystując architekturę BERT_{SMALL}.

4.4.2. Wyniki eksperymentu

4.5. Eksperyment 3

4.5.1. Opis eksperymentu

TODO: co chcemy zrobić, z jakimi parametrami, po co - co mierzymy: metryki + czas/ram

W ostatniej fazie porównujemy architektury BERT_{SMALL} z architekturą AutoTinyBERT_{S1} aby sprawdzić, czy wnioski z tego artykułu przekładają się na inne mechanizmy uwagi oraz na inne dane. Wykorzystamy parametry z eksperymentu 1 oraz parametry z eksperymentu 2, które uznaliśmy za najlepsze.

4.5. EKSPERYMENT 3

4.5.2. Wyniki eksperymentu

5. Analiza działania systemu

5.1. Testy jednostkowe

Testy są uruchamiane przy użyciu `pytest`. Dla izolacji i kontroli zależności stosowany jest mechanizm `monkeypatch`.

- **Mockowanie usług:** W testach loggera W&B wykorzystywany jest obiekt zastępczy (`DummyRun`) oraz `monkeypatch` do podmiany `wandb.init`. Dzięki temu testy nie wykonują połączeń sieciowych.
- **Izolacja I/O:** Zapisy stanów, CSV i sztucznych zbiorów danych wykonywane są do katalogów tymczasowych (`tmp_path`).

Testy weryfikują poprawność działania całego pakietu `src/textclf_transformer`. Testy są zorganizowane w katalogu `tests/unit/` zgodnie z konwencją nazewnictwa `pytest` (`test_*.py`). Raport pokrycia znajduje się w tabeli 5.1. Testy obejmują następujące obszary:

- **Logger** (`tests/unit/logger`):
 - Poprawność zapisu metryk do plików CSV (przy wyłączonym W&B) oraz logowanie do serwisu W&B.
 - Rejestrowanie metryk systemowych (np. zużycie pamięci GPU).
- **Tokenizer** (`tests/unit/tokenizer`):
 - Walidacja ładowania i treningu tokenizera z plików.
 - Poprawność kodowania tekstów (pojedynczych oraz z ramek `Pandas`) i obsługa etykiet.
 - Mechanizm maskowania wejścia dla modelu MLM (z pominięciem tokenów specjalnych).
- **Training** (`tests/unit/training` oraz `training/utils`):
 - Logika pętli treningowej dla klasyfikacji (zapis najlepszego modelu) i MLM, wznawianie treningu oraz tryb dostrajania (ładowanie wag z pretreningu).

Tabela 5.1: Raport pokrycia kodu testami

Moduł	Instrukcje	Pominięte	Pokrycie
<i>Core</i>			
__init__	5	0	100%
<i>Logger</i>			
__init__	1	0	100%
wandb_logger	98	16	84%
<i>Tokenizer</i>			
__init__	1	0	100%
wordpiece_tokenizer_wrapper	117	8	93%
<i>Training</i>			
__init__	2	0	100%
train	52	9	83%
training_loop	238	41	83%
utils/__init__	4	0	100%
utils/config	56	28	50%
utils/dataloader_utils	42	1	98%
utils/metrics_utils	42	0	100%
utils/train_utils	59	1	98%
<i>Modele – ogólne i bloki</i>			
__init__	13	0	100%
consts	2	0	100%
transformer	40	10	75%
transformer_classification	29	3	90%
transformer_mlm	16	0	100%
blocks/attention_block	24	0	100%
blocks/mlp_block	10	0	100%
blocks/transformer_encoder_block	12	0	100%
<i>Mechanizmy uwagi</i>			
multihead_sdp_self_attention	79	4	95%
multihead_favor_self_attention	180	13	93%
multihead_lsh_self_attention	145	3	98%
<i>Embeddingi</i>			
positional_encodings	24	0	100%
rotary	27	0	100%
text_embeddings	57	1	98%
<i>Pooling i Głowice</i>			
pooling	37	1	97%
classifier_head	26	0	100%
mlm_head	22	1	95%
SUMA	1460	140	90%
Liczba testów: 115 Błędy: 0 Niepowodzenia: 0 Pominięto: 0			

- Rozwiązywanie ścieżek względem katalogu głównego repozytorium, obsługa konfiguracji YAML, zapis i odczyt zapisanych stanów (pełnych stanów oraz samych wag).
- Obliczanie metryk (perplexity dla MLM, metryki sklearn dla klasyfikacji), poprawne działanie `collate_fn` (padding, przycinanie) oraz załadowanie `TensorDataset`.

- **Modele - ogólne i bloki** (tests/unit/models, blocks):
 - Propagacja maski paddingu i poprawność kształtów tensorów wyjściowych.
 - Struktura i inicjalizacja bloków MLP, Uwagi oraz Bloku enkodera (mechanizmy rezydualne i normalizacja).
 - Współdzielenie wag w modelu MLM.
- **Mechanizmy uwagi** (tests/unit/models/attention):
 - MHA: Zgodność numeryczna z implementacją PyTorch, obsługa SDPA, determinizm w trybie ewaluacji.
 - FAVOR+: Stabilność numeryczna wariantów funkcji ϕ (w tym \exp), obsługa masek, poprawność gradientów i mechanizmów stabilizacji.
 - LSH: Respektowanie masek (dopełnianie i podział na bloki), stabilność haszowania oraz weryfikacja analityczna na małych próbkach.
- **Embeddingi** (tests/unit/models/embeddings):
 - Poprawność wzorów kodowania pozycyjnego: sinusoidalne, uczone oraz RoPE (rotacja, cache, obsługa `position_ids`).
 - Inicjalizacja embeddingów tekstowych, zerowanie wektora paddingu.
- **Pooling i Głowice** (tests/unit/models/pooling, heads):
 - Poprawność algorytmów CLS, Mean, Max, Min.
 - Architektury agregatorów (poolerów) w głowicach klasyfikacyjnych (BERT/RoBERTa) oraz struktura głowicy MLM.

5.2. Testy akceptacyjne

Tabela 5.2 przedstawia ocenę spełnienia wymagań niefunkcjonalnych zdefiniowanych w sek. 1.6.

Tabela 5.3 przedstawia ocenę spełnienia wymagań funkcjonalnych zdefiniowanych w sek. 1.5.

Tabela 5.2: Wymagania niefunkcjonalne

Wymaganie	Status	Komentarz
<i>WNF-1 — Wydajność i efektywność zasobowa</i>		
Środowisko GPU (Colab)	Spełnione	Wykorzystano GPU <i>A100 40GB</i>
Wymóg kosztowy (Performer)	TODO: Częściowo	Zgodnie z tabelami ?? i ??
Wymóg kosztowy (Reformer)		Zgodnie z tabelami ?? i ??
Techniki optymalizacji	Spełnione	Zgodnie z sek. 2.3.2
<i>WNF-2 — Jakość, niezawodność i testowalność</i>		
Jakość (SDPA)		Zgodnie z tabelami ?? i ??
Jakość (Performer)		Zgodnie z tabelami ?? i ??
Jakość (Reformer)		Zgodnie z tabelami ?? i ??
Stabilność	TODO:	TODO: tabela rozrzut fl
Testy komponentów	Spełnione	Zgodnie z sek. 5.1
<i>WNF-3 — Użyteczność i utrzymanie</i>		
Dokumentacja	Spełnione	Zgodnie z sek. 2.7
Struktura katalogów	Spełnione	Zgodnie z sek. 2
Zgodność z PEP-8	Spełnione	TODO: załącznik repo
Wersjonowanie	Spełnione	TODO: załącznik repo
Rozszerzalność mechanizmów uwagi	Spełnione	Zgodnie z sek. 2.1.2
Konfiguracja YAML	Spełnione	Zgodnie z sek. 2.5
<i>WNF-4 — Przenośność i kompatybilność</i>		
Kompatybilność Python	Spełnione	Zgodnie z sek. 2.6
Biblioteki	Spełnione	Zgodnie z sek. 2.6
<i>WNF-5 — Monitorowanie i obserwowalność</i>		
Monitorowanie W&B	Spełnione	Zgodnie z sek. 2.4
Monitorowanie CSV	Spełnione	Zgodnie z sek. 2.4
Wznowienia treningu	Spełnione	Zgodnie z sek. 2.3.4 i 2.7.3

Tabela 5.3: Weryfikacja wymagań funkcjonalnych

Wymaganie	Status	Komentarz
<i>WF-1 — Pretrening i dostrajanie</i>		
Pełny cykl uczenia (MLM + CLS)	Spełnione	Zgodnie z sek. 2.3.2
Zapis i wznawianie stanu	Spełnione	Zgodnie z sek. 2.3.4 i 2.7.3
Logowanie metryk (CSV, W&B)	Spełnione	Zgodnie z sek. 2.4
<i>WF-2 — Wymienne mechanizmy uwagi</i>		
Deklaratywny wybór w YAML	Spełnione	Zgodnie z sek. 2.5
Obsługa SDPA, LSH, FAVOR+	Spełnione	Zgodnie z sek. 2.1.2
Kompatybilność interfejsów	Spełnione	Zgodnie z sek. 5.1
<i>WF-3 — Obsługa długich sekwencji</i>		
Kodowanie pozycyjne	Spełnione	Zgodnie z sek. 2.1.4
Obsługa sekwencji > 512	Spełnione	Parametr <code>max_length</code> przyjmuje dowolną wartość (tab. 2.1).
<i>WF-4 — Pipeline danych</i>		
Tokenizacja WordPiece	Spełnione	Zgodnie z sek. 2.2
Dynamiczny padding	Spełnione	Zgodnie z sek. 2.3.3
Maskowanie MLM (BERT)	Spełnione	Zgodnie z sek. 2.2
<i>WF-5 — Konfiguracja</i>		
Generator eksperymentów	Spełnione	Zgodnie z sek. 2.5
Separacja pretreningu i dostrajania	Spełnione	Zgodnie z sek. 2.5

6. Podsumowanie

: **TODO:** I Is the final product a success? I What could be done better? I What we achieved and how we avoided dangers? I What are the plans for future? I Are there any open problems for future?

Pomysly na Future works: -Niskopoziomowa optymalizacja lsh i favor (znalezc prace ktore tak robia).

-Taktyka dla krotkich sekwencji (np przy ptretreningu uzywac mha) dla dlugich zmienic (sprawdzic czy takie cos istnieje - longformer tak robi startujac z zapisanego stanu roberta i zmienia uwagi na swoja

-Doimplementowanie innych uwagi do systemu

TODO: pz-5.4 - moze dac to ale jakos przeredagowane

Bibliografia

- [1] Iz Beltagy, Matthew E Peters i Arman Cohan. “Longformer: The long-document transformer”. W: *arXiv preprint arXiv:2004.05150* (2020).
- [2] Rewon Child. “Generating long sequences with sparse transformers”. W: *arXiv preprint arXiv:1904.10509* (2019).
- [3] Krzysztof Marcin Choromanski i in. “Rethinking Attention with Performers”. W: *International Conference on Learning Representations*. 2021.
- [4] Tri Dao i in. “Flashattention: Fast and memory-efficient exact attention with io-awareness”. W: *Advances in neural information processing systems* 35 (2022), s. 16344–16359.
- [5] Jacob Devlin i in. “Bert: Pre-training of deep bidirectional transformers for language understanding”. W: *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, s. 4171–4186.
- [6] Suchin Gururangan i in. “Don’t Stop Pretraining: Adapt Language Models to Domains and Tasks”. W: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Red. Dan Jurafsky i in. Online: Association for Computational Linguistics, lip. 2020, s. 8342–8360. DOI: 10.18653/v1/2020.acl-main.740.
- [7] Nikita Kitaev, Lukasz Kaiser i Anselm Levskaya. “Reformer: The Efficient Transformer”. W: *International Conference on Learning Representations*. 2020.
- [8] Aixin Liu i in. “Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model”. W: *arXiv preprint arXiv:2405.04434* (2024).
- [9] Aixin Liu i in. “Deepseek-v3. 2: Pushing the frontier of open large language models”. W: *arXiv preprint arXiv:2512.02556* (2025).
- [10] Yinhan Liu i in. “Roberta: A robustly optimized bert pretraining approach”. W: *arXiv preprint arXiv:1907.11692* (2019).
- [11] Zhen Qin i in. “cosformer: Rethinking softmax in attention”. W: *arXiv preprint arXiv:2202.08791* (2022).

- [12] Jianlin Su i in. “RoFormer: Enhanced transformer with Rotary Position Embedding”. W: *Neurocomputing* 568 (2024), s. 127063. ISSN: 0925-2312. DOI: <https://doi.org/10.1016/j.neucom.2023.127063>. URL: <https://www.sciencedirect.com/science/article/pii/S0925231223011864>.
- [13] Iulia Turc i in. “Well-read students learn better: On the importance of pre-training compact models”. W: *arXiv preprint arXiv:1908.08962* (2019).
- [14] Ashish Vaswani i in. “Attention is all you need”. W: *Advances in neural information processing systems* 30 (2017).
- [15] Sinong Wang i in. “Linformer: Self-attention with linear complexity”. W: *arXiv preprint arXiv:2006.04768* (2020).
- [16] Manzil Zaheer i in. “Big bird: Transformers for longer sequences”. W: *Advances in neural information processing systems* 33 (2020), s. 17283–17297.

Spis rysunków

2.1	Diagram komponentów systemu.	23
2.2	Diagram klas implementacji transformera. Przedstawiono wyłącznie pola będące modułami PyTorch (dziedziczące po <code>nn.Module</code>). Wszystkie argumenty funkcji bez jawnie określonego typu są typu <code>torch.Tensor</code> , a wszystkie funkcje zwracają obiekt typu <code>torch.Tensor</code>	24
2.3	Schemat użytkowania systemu	46
4.1	Schemat architektury użytej w eksperymentach (nazewnictwo zgodne z nazwami klas w sek. 2.1)	65

Spis tabel

1.1	Analiza ryzyk projektu z przypisanymi właścicielami i oceną prawdopodobieństwa oraz wpływu	21
1.2	Podział prac i odpowiedzialności w projekcie.	22
2.1	Parametry eksperymentu, logowania i tokenizacji	40
2.2	Hiperparametry architektury Transformera	40
2.3	Parametry mechanizmu uwagi	41
2.4	Hiperparametry procesu treningowego	42
2.5	Parametry kontynuacji treningu	42
2.6	Parametry głowic zadaniowych	43
2.7	Parametry konfiguracji danych	43
2.8	Wykorzystywane biblioteki Python	45
4.1	Statystyki długości tokenów po tokenizacji	63
4.2	Hiperparametry treningu dla poszczególnych etapów uczenia	66
4.3	Wspólne hiperparametry dla wszystkich etapów treningu	66
4.4	Wyniki klasyfikatora TF-IDF + LR dla zbioru IMDb	67
4.5	Wyniki klasyfikatora TF-IDF + LR dla zbioru arXiv	67
4.6	Wyniki klasyfikatora TF-IDF + LR dla zbioru Hyperpartisan	68
4.7	Przestrzeń poszukiwań hiperparametrów w Eksperymentcie 1	69
4.8	Wyniki F1-macro na zbiorze testowym dla różnych konfiguracji hiperparametrów.	69
4.9	Optymalne hiperparametry warstwy wybrane w fazie E1.	69
4.10	Przestrzeń poszukiwań hiperparametrów w Eksperymentcie 2	70
4.11	Wyniki F1-macro dla różnych konfiguracji mechanizmów uwagi.	70
5.1	Raport pokrycia kodu testami	73
5.2	Wymagania нефункционалне	75
5.3	Weryfikacja wymagań funkcjonalnych	76

Spis załączników

1. repo