Politechnika Warszawska

Wydział Matematyki
i Nauk Informacyjnych

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

Standardowy mechanizm uwagi w architekturze Transformer charakteryzuje się kwadratową złożonością obliczeniową $O(N^2)$, co stanowi istotne ograniczenie przy przetwarzaniu długich sekwencji tekstowych. Celem pracy jest implementacja enkodera typu BERT umożliwiającego wymianę modułu uwagi oraz porównanie trzech mechanizmów: standardowego SDPA, LSH (Reformer) oraz FAVOR+ (Performer), pod kątem zarówno jakości klasyfikacji, jak i czasu uczenia oraz wymaganych zasobów. Zaprojektowano modułowy system obejmujący architekturę transformera, tokenizer, pętlę treningową z obsługą wstępnego trenowania i dostrajania, deklaratywną konfigurację eksperymentów oraz integrację z platformą Weights & Biases. Dla mechanizmu LSH zaproponowano autorskie podejście do sposobu maskowania przy obliczaniu uwagi. Przeprowadzono eksperymenty na trzech zbiorach danych (IMDb, Hyperpartisan, arXiv) o długościach sekwencji 512–16384 tokenów, stosując trzyetapowy proces uczenia: wstępne trenowanie z wykorzystaniem maskowanego modelowania języka, adaptację domenową oraz dostrajanie do zadania klasyfikacji. Dla sekwencji długości 16384 tokenów przybliżone mechanizmy uwagi (w najszybszych konfiguracjach) umożliwiły redukcję czasu uczenia o ponad 50% względem SDPA, przy niewielkim spadku metryki F1-macro.

Słowa kluczowe: transformer, mechanizm uwagi, BERT, Performer, Reformer, klasyfikacja tekstu, przetwarzanie języka naturalnego

Abstract

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

The standard attention mechanism in the Transformer architecture exhibits quadratic computational complexity $O(N^2)$, which poses a significant limitation when processing long text sequences. The aim of this thesis is to implement a BERT-type encoder with interchangeable attention modules and to compare three mechanisms: standard SDPA, LSH (Reformer), and FAVOR+ (Performer), in terms of classification quality, training time, and resource requirements. A modular system was designed, comprising the transformer architecture, a tokenizer, a training loop supporting pretraining and fine-tuning, declarative experiment configuration, and integration with the Weights & Biases platform. For the LSH mechanism, a novel approach to attention masking was proposed. Experiments were conducted on three datasets (IMDb, Hyperpartisan, arXiv) with sequence lengths ranging from 512 to 16384 tokens, employing a three-stage training process: pretraining using masked language modeling, domain adaptation, and fine-tuning for the classification task. For sequences of 16384 tokens, approximate attention mechanisms (in their fastest configurations) enabled a reduction in training time of over 50% compared to SDPA, with a slight decrease in the F1-macro metric.

Keywords: transformer, attention mechanism, BERT, Performer, Reformer, text classification, NLP

Spis treści

Podział pracy autorów

Rola	**Bartłomiej Borycki** **nr albumu: 327265**	**Michał Iwaniuk** **nr albumu: 327278**
Przygotowanie tekstu pracy – wersja początkowa	Rozdziały: 1, 4, 5, 6	Rozdziały: 2, 3, 5, 6
Przygotowanie tekstu pracy – przegląd i redakcja	Rozdziały: 1, 2, 3, 4, 5, 6	Rozdziały: 1, 2, 3, 4, 5, 6
Oprogramowanie	• Mechanizm uwagi FAVOR+ • Klasa Transformer • Skrypty treningowe • Generator eksperymentów	• Mechanizm uwagi SDPA • Mechanizm uwagi LSH • Bloki enkodera • Testy jednostkowe
Kluczowe rozwiązania pracy: konceptualizacja, badania oraz metodologia	Projekt architektury systemu, trening modeli, optymalizacja procesu treningu i dobór hiperparametrów, implementacja rozwiązań badawczych (mechanizm FAVOR+)	Projekt architektury systemu, trening modeli, opracowanie metodyki eksperymentów, implementacja rozwiązań badawczych (mechanizm LSH)
Analiza formalna i walidacja	Analiza wyników eksperymentów, porównanie mechanizmów uwagi	Analiza wyników eksperymentów, walidacja testami jednostkowymi, analiza eksploracyjna danych
Przygotowanie danych	Przygotowanie zbiorów IMDb i Hyperpartisan	Przygotowanie zbiorów Wikipedia i ArXiv
Zarządzanie projektem	TAK	TAK
Zasoby	TAK – środowisko Google Colab	TAK – środowisko Google Colab
Wizualizacja	TAK	TAK
Pozyskanie finansowania	NIE	NIE

Deklaracja użycia narzędzi AI

Niniejszym oświadczamy, że:

1. Nie używaliśmy narzędzi informatycznych do generowania treści niniejszej pracy.
2. Używaliśmy narzędzi informatycznych do generowania kodu programów stworzonych na potrzeby niniejszej pracy.
3. Bierzemy pełną odpowiedzialność za zawartość niniejszej pracy, która składa się zarówno z jej tekstu, jak i stworzonego na jej potrzeby oprogramowania.

Zakres użycia narzędzi informatycznych do generowania tekstu pracy	**Zakres użycia narzędzi informatycznych do generowania kodu**
	Generowanie opisów funkcji (ang. *docstring*) oraz komentarzy, generowanie testów jednostkowych kodu, funkcje pomocnicze do wczytywania i obsługi plików YAML i CSV

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* [23] 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 [7] 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 [4], Reformer [13], 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 niezmiennej 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 przestrzeni stanów

Mamba (2024) [8] – architektura oparta na selektywnych modelach przestrzeni stanów (ang. *Selective State Space Models*, SSM), stanowiąca alternatywę dla mechanizmu uwagi. W przeciwieństwie do transformerów, Mamba przetwarza sekwencję w czasie liniowym $O(N)$ zarówno podczas treningu, jak i inferencji, dynamicznie dostosowując parametry modelu w zależności od wejścia. Architektura ta osiąga konkurencyjne wyniki przy znacznie niższych wymaganiach obliczeniowych dla bardzo długich sekwencji.

Modele oparte na transformerze

Transformer (2017) [23] – całkowite odejście od rekurencji. Mechanizm uwagi pozwala każdemu tokenowi patrzeć na wszystkie inne tokeny jednocześnie. Umożliwia pełne zrównoleglenie obliczeń na GPU.

BERT (2019) [7], 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) [16] – zoptymalizowana wersja BERT z usuniętym zadaniem NSP, dłuższym treningiem i większymi batchami. Pokazuje, że procedura treningowa ma duży wpływ na jakość modelu. W naszej pracy wzorujemy się na tym podejściu, rezygnując z zadania NSP.

Popularne modyfikacje mechanizmu uwagi i architektury

Sparse Transformer (2019) [3] – 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) [25] – 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) [13] – 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) [2] – łączy lokalną uwagę okienkową z uwagą globalną dla wybranych tokenów (np. `[CLS]`). Umożliwia to przetwarzanie bardzo długich dokumentów przy liniowej zależności kosztu od długości sekwencji $O(N)$.

Linformer (2020) [24] – 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) [4] – 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) [17] – 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.

Mistral 7B (2023) [11] – model językowy, który spopularyzował efektywne połączenie uwagi z oknem przesuwnym (ang. *Sliding Window Attention*, SWA) oraz *Grouped-Query Attention* (GQA). Każdy token zwraca uwagę jedynie na ograniczoną liczbę poprzedzających tokenów (np. 4096), co redukuje złożoność pamięciową do $O(N \cdot w)$, gdzie w to rozmiar okna. Dzięki propagacji informacji przez kolejne warstwy model efektywnie agreguje kontekst z całej sekwencji. Podejście to zostało zaadoptowane w wielu późniejszych modelach, m.in. Gemma 2 [20] i Phi-3 [1].

FlashAttention (2022) [6] – 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.

FlashAttention-2 (2024) [5] oraz **FlashAttention-3** (2024) [18] – kolejne wersje algorytmów IO-aware dla dokładnej uwagi, kładące nacisk na lepszy podział pracy i wykorzystanie nowoczesnych GPU (szczególnie H100), co daje istotne przyspieszenie bez aproksymacji.

Wraz z szybkim rozwojem modeli LLM opartych na transformerze pojawiają się także nowsze warianty uwagi, np. **Multi-Head Latent Attention** (2024) [14], 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) [15], 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.

Prawa skalowania

Równolegle z rozwojem architektur prowadzone są badania nad prawami skalowania (ang. *scaling laws*), które empirycznie opisują zależności między wielkością modelu, ilością danych treningowych i budżetem obliczeniowym a uzyskiwaną jakością [12, 10]. Należy jednak podkreślić, że większość tych wyników dotyczy modeli średnich i dużych (miliardy parametrów), a w ustawieniach niskozasobowych obserwuje się istotne odchylenia od klasycznych zależności – optymalna relacja między liczbą parametrów a liczbą tokenów może być przesunięta w stronę większej ilości danych, a prognozy dla małych modeli oparte na prawach wyznaczonych dla dużych modeli mogą być nietrafne [22]. W niniejszej pracy nie ekstrapolujemy praw skalowania wprost na mały model (rzędu $30M$ parametrów), lecz wykorzystujemy ich główną, jakościową implikację – przy ograniczonym budżecie obliczeniowym istotne jest możliwie efektywne wykorzystanie dostępnych danych. Motywuje to poszukiwanie bardziej efektywnych mechanizmów uwagi, które redukują koszt obliczeniowy i umożliwiają dłuższe trenowanie (lub użycie dłuższych sekwencji / większej liczby tokenów) przy stałym budżecie.

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**: Enkoder typu Transformer, odpowiedzialny za tworzenie reprezentacji wektorowych tekstu.

- **Głowice 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:
  - stokenizowany tekst (ID tokenów),
  - 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 projektu jest opracowanie rozwiązania do klasyfikacji tekstu, które będzie efektywnie przetwarzać długie dokumenty przy ograniczonych zasobach obliczeniowych. Pozwoliłoby to na wykorzystanie nowoczesnych metod przetwarzania języka naturalnego również w środowiskach, w których dostęp do wydajnej infrastruktury jest ograniczony.

Główna hipoteza projektu zakłada, że odpowiedni dobór architektury modelu – w tym mechanizmów uwagi – pozwoli znacząco obniżyć koszty obliczeniowe, jednocześnie zachowując jakość predykcji na poziomie wymaganym w zastosowaniach produkcyjnych.

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:** stokenizowane 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 skalera 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. A.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. Maska uwagi jest respektowana 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 [19].

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 niefunkcjonalne

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):** $time/epoch \leqslant$ (SDPA -20%) lub $\max VRAM \leqslant$ (SDPA -30%) przy spadku jakości $\leqslant 1$ pp macro-F1.

- **Wymóg kosztowy (Reformer):** $time/epoch \leqslant$ (SDPA -15%) lub $\max VRAM \leqslant$ (SDPA -25%) przy spadku jakości $\leqslant 1$ 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):** macro-F1 $\geqslant$ (TF-IDF+LogReg +5 pp), mierzone na wszystkich docelowych zbiorach danych, parametry jak w konfiguracji.

- **Jakość (Performer):** macro-F1 $\geqslant$ (TF-IDF+LogReg +3 pp) oraz w odległości $\leqslant$ 1 pp od SDPA przy tej samej konfiguracji – na wszystkich docelowych zbiorach danych.
- **Jakość (Reformer):** macro-F1 $\geqslant$ (TF-IDF+LogReg +3 pp) oraz w odległości $\leqslant$ 1 pp od SDPA przy tej samej konfiguracji – na wszystkich docelowych zbiorach danych.
- **Stabilność:** brak NaN/OOM; 3 różne seedy; rozrzut macro-F1 (max-min) $\leqslant$ 1 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 $\geqslant$ 3.12 oraz z frameworkiem PyTorch $\geqslant$ 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 triggery, proponowane mitigacje oraz plany awaryjne. W tab. 1.1 przedstawiamy prawdopodobieństwo, wpływ oraz właścicieli poszczególnych ryzyk.

1. **Ograniczenia zasobów (VRAM).** *Trigger:* OOM przy $N \geqslant 16k$. *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.
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 rewalidacja 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.

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

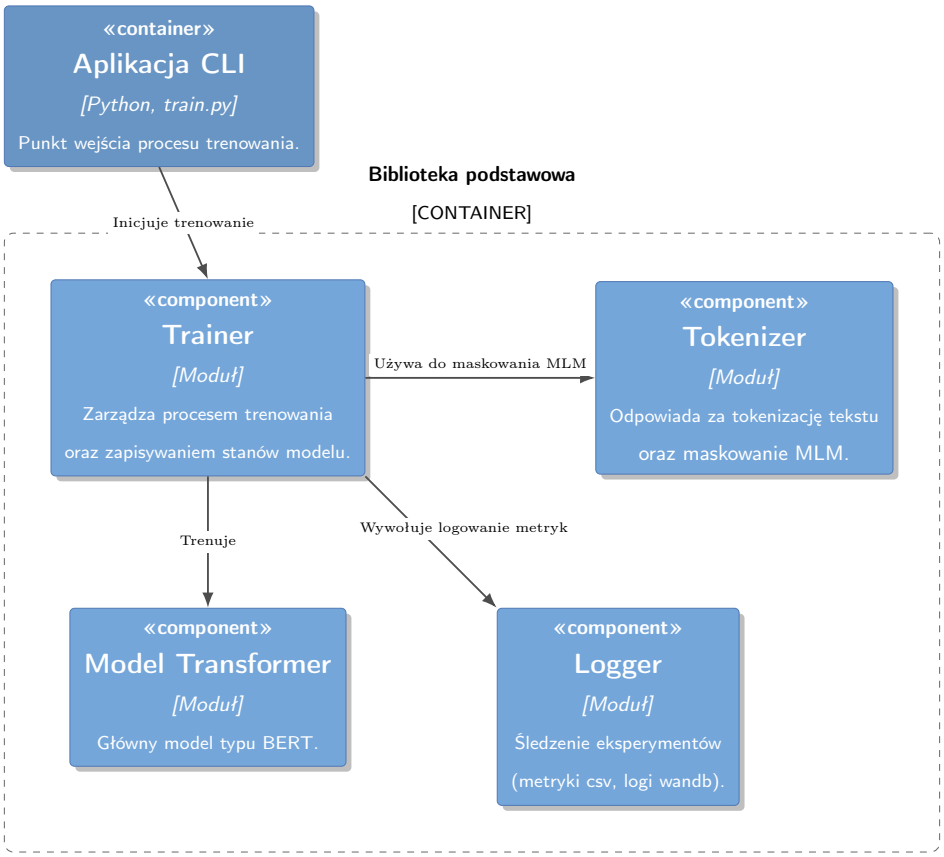
Ryzyko	**Prawd.**	**Wpływ**	**Właściciel**
OOM przy długich sekwencjach	Średnie	Wysoki	Bartłomiej Borycki
Niestabilność treningu	Średnie	Wysoki	Bartłomiej Borycki
Błędy w Reformer/Performer	Średnie	Średni	Michał Iwaniuk
Niedopasowanie tokenizera	Niskie	Średni	Michał Iwaniuk
Awaria sesji Colab / W&B	Niskie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

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.

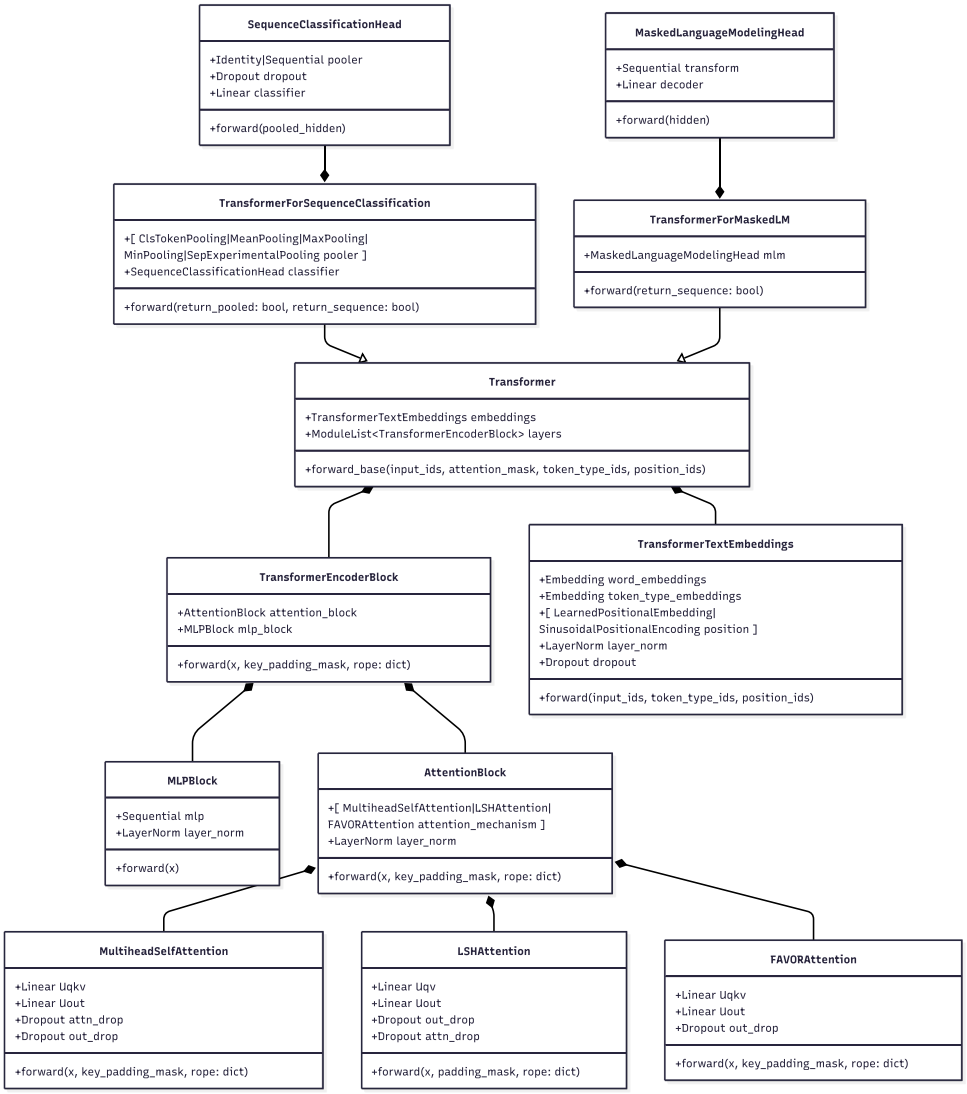
2. Architektura systemu

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 2.1: Diagram komponentów systemu.



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.

2.1.1. 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, jak i logity klasyfikacji $(B, \texttt{num\_labels})$.

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 [7].

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:
$$\text{Attention}(Q,K,V) = \text{Dropout}\left(\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)\right)V. \tag{2.1}$$
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 $O(N)$ [4]. 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`:
$$\hat{V} = D^{-1}(Q'(K'^T V)), \tag{2.2}$$
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 $O(N)$ [13]. 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 [13], 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 wchodzą 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 pierwotna architekturą Transformera [23]. 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:

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

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

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)* [19])

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
$$\theta_i = 10000^{-\frac{2i}{D}}, \qquad i = 0, 1, \ldots, \frac{D}{2} - 1. \tag{2.5}$$
Następnie dla każdej pozycji m (oraz każdego i) wylicza się tablice:
$$\cos(m\theta_i), \qquad \sin(m\theta_i). \tag{2.6}$$

2. **Aplikacja (`apply_rope`):** dla każdej pary kolejnych składowych $(x_{m,2i}, x_{m,2i+1})$ na pozycji m wykonuje się rotację o kąt $m\theta_i$:
$$\begin{pmatrix} x'_{m,2i} \\ x'_{m,2i+1} \end{pmatrix} = \begin{pmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{pmatrix} \begin{pmatrix} x_{m,2i} \\ x_{m,2i+1} \end{pmatrix}. \tag{2.7}$$

Implementacja funkcji `_rotate_half` realizuje operację $\text{rotate\_half}([x_{2i}, x_{2i+1}]) = [-x_{2i+1}, x_{2i}]$ w sposób zwektoryzowany, co umożliwia obliczenie rotacji bez jawnego tworzenia macierzy rotacji dla każdego tokenu:

$$\mathbf{x}' = \mathbf{x} \odot \cos(m\boldsymbol{\theta}) \; + \; \text{rotate\_half}(\mathbf{x}) \odot \sin(m\boldsymbol{\theta}), \tag{2.8}$$

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

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 [7]. Jako reprezentację całej sekwencji przyjmuje się stan ukryty pierwszego tokenu specjalnego (zwyczajowo `[CLS]`).

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

Agregacja uśredniająca (MeanPooling)

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

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

Pooling maksymalny i minimalny (MaxPooling, MinPooling)

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

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

Agregacja uśredniająca z krokiem (MeanStepPooling)

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

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

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 [7]:** Składa się z warstwy gęstej zachowującej wymiarowość, funkcji aktywacji Tanh, a następnie warstwy wyjściowej:

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

- **Styl RoBERTa [16]:** Charakteryzuje się dodatkowym Dropoutem na wejściu:

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

- **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 [7] i obejmuje:

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

2.2. Tokenizer

W projekcie wykorzystano algorytm tokenizacji WordPiece (używany w oryginalnym modelu BERT [7]). 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 buduje słownik podjednostek (subwords) o zadanej wielkości (domyślnie 30 000 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.
- `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`.
   - **Dostrajanie:** 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

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:** 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 przerwanego 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 chmurowa platforma 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, przesyłając 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 metryka jest perpleksja (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 $\leqslant 10$) raportowane są precyzja, czułość i F1 dla każdej klasy osobno.
- **Metryki systemowe:** Platforma Weights & Biases automatycznie gromadzi dane o utylizacji 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 A.1, A.2, A.3, A.4, A.6, A.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:

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. **Weryfikacja źródła:** Sprawdzenie istnienia katalogu i pliku konfiguracyjnego eksperymentu bazowego.

2. **Kopiowanie architektury:** Sekcje `architecture` oraz `tokenizer` są kopiowane bezpośrednio z konfiguracji pretreningu do konfiguracji dostrajania (*finetuningu*). 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. **Relatywizacja 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.

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.

Tabela 2.1: Wykorzystywane biblioteki Python

Biblioteka	**Wersja**
`torch`	2.8.0
`transformers`	4.56.2
`tokenizers`	0.22.1
`pandas`	2.2.3
`numpy`	1.26.4
`scikit-learn`	1.6.1
`wandb`	0.22.1
`PyYAML`	6.0.2
`pytest`	8.3.4
`datasets`	4.3.0
`pydantic`	$\geqslant$2.12.0

2.6.2. Instalacja środowiska

1. **Utworzenie wirtualnego środowiska Python:**

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

Listing 2.1: Tworzenie środowiska wirtualnego

2. **Aktualizacja pip i instalacja zależności:**

```
pip install --upgrade pip
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 2.1.

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.
- `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`.

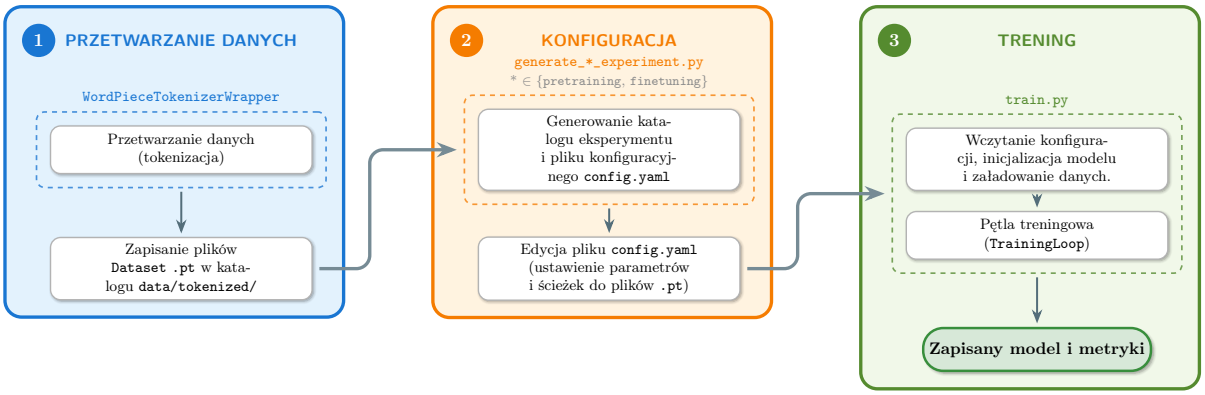
2.7. Podręcznik użytkownika

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

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`.



Rysunek 2.3: Schemat użytkowania systemu

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:

```
from textclf_transformer.tokenizer.wordpiece_tokenizer_wrapper \
    import WordPieceTokenizerWrapper

tokenizer = WordPieceTokenizerWrapper()
tokenizer.train(tokenizer_dir="my_tokenizer_dir", input="data/raw/input.txt")
```

Listing 2.3: Trening tokenizera

```
from textclf_transformer.tokenizer.wordpiece_tokenizer_wrapper \
    import WordPieceTokenizerWrapper
import torch
import pathlib

tokenizer = WordPieceTokenizerWrapper()
tokenizer.load("src/textclf_transformer/tokenizer/my_tokenizer_dir")

# Użycie encode z plikiem tekstowym
```

```
ds = tokenizer.encode(
    input="data/raw/text_input.txt",
    # labels=labels_tensor, # opcjonalne
    max_length=512,
)

out = pathlib.Path("data/tokenized/train_dataset.pt")
out.parent.mkdir(parents=True, exist_ok=True)
torch.save(ds, out)
```

Listing 2.4: Tokenizacja z pliku

```
import pandas as pd

df = pd.read_csv("data/raw/data.csv")
ds = tokenizer.encode_pandas(
    df=df,
    text_col="text",
    label_col="label", # opcjonalne
    max_length=512
)
```

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

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

Listing 2.6: Generowanie eksperymentu pretreningu

```
python experiments/generate_finetuning_experiment.py \
    -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

```
python experiments/generate_pretraining_experiment.py \
    -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 przerwanego 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`.

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

Listing 2.9: Uruchomienie pretreningu

```
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 [23] podstawową operacją jest wielogłowowa 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 $z = (z^1, \ldots, z^N)$ oznacza sekwencję N elementów (tokenów), gdzie $z^i \in \mathbb{R}^D$ jest wektorem reprezentacji i-tego elementu. Wagi uwagi A_{ij} są wyznaczane 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:

$$[Q, K, V] = z\, U_{qkv}, \tag{3.1}$$

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

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

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

Operator uwagi dla wejścia z definiujemy jako ważoną kombinację wartości:

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

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

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

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

Wielogłowowa 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

$$d_k = \frac{D}{h}. \tag{3.5}$$

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

Wynik wielogłowowej samouwagi definiujemy jako:

$$\mathrm{MSA}(z) = [SA_1(z);\; SA_2(z);\; \ldots;\; SA_h(z)]\, U_{\mathrm{out}}, \tag{3.6}$$

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

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

natomiast macierz wyjściowej projekcji spełnia:

$$U_{\mathrm{out}} \in \mathbb{R}^{D \times D}. \tag{3.8}$$

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 [23]. 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 jąder 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. 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 (`torch.nn.functional.scaled_dot_product_attention`). Pozwala to na traktowanie wyników SDPA jako silnego, przemysłowego punktu odniesienia dla badanych uwag przybliżonych (LSH i FAVOR+).

3.2. LSH

Locality-Sensitive Hashing attention (zaproponowana w [13]) jest przybliżeniem pełnej uwagi, które redukuje liczbę porównań przez ograniczenie uwagi do elementów podobnych – identyfiko-

wanych za pomocą haszowania.

Zamiast liczyć podobieństwo $q_i \cdot k_j$ dla wszystkich j, najpierw haszujemy wektory (zapytania i klucze) tak, by wektory podobne (bliskie kątowo) 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:

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

Niech z oznacza wyraz normalizujący w softmaksie, to znaczy:

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

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

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

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) $R \in \mathbb{R}^{d_k \times (b/2)}$, $R_{ij} \sim \mathcal{N}\Big(0, \frac{1}{d_k}\Big)$, 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 konkatenację dwóch wektorów. Wtedy zbiór $\mathcal{P}_i$ przyjmuje postać:

$$\mathcal{P}_i = \Big\{ j : h(q_i) = h(k_j) \Big\}, \tag{3.12}$$

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 $m(j, \mathcal{P}_i) = \infty \quad$ 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:

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

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

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

Dzięki temu możemy zapisać:

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

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 $h^{(1)}, h^{(2)}, \ldots, h^{(n_{\text{rounds}})}$.

Definiujemy:

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

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

W oryginalnej pracy zastosowano:

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

oraz obliczono o_i jako:

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

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

W naszej pracy zmodyfikowaliśmy to w następujący sposób:

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

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

Zdefiniujmy:

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

Wtedy o_i możemy zapisać jako:

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

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:

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

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:

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

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

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

gdzie

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

Wpływ tej modyfikacji na trening modelu znajduje się w sek. 4.7.2. W opisanych w dalszej części pracy eksperymentach (sek. 4) będziemy korzystać z tej zmodyfikowanej wersji LSH (brak maski).

3.3. FAVOR+

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

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

Wyjście attention można zapisać jako

$$\mathrm{Att}(Q, K, V) = D^{-1}\Big(AV\Big), \qquad D = \mathrm{diag}\Big(A\mathbf{1}_N\Big). \tag{3.28}$$

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* zaproponowana w [4]) 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

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

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

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

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

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

Definiujemy macierze cech dla całej sekwencji:

$$Q' \in \mathbb{R}^{N\times m}, \quad K' \in \mathbb{R}^{N\times m}, \qquad \text{gdzie wiersze to } (Q')_{i\cdot} = \phi(q_i)^\top,\ (K')_{j\cdot} = \phi(k_j)^\top. \tag{3.32}$$

Wówczas macierz wag uwagi można przybliżyć przez

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

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

$$\widehat{\mathrm{Att}}(Q,K,V) = \widehat{D}^{-1}\Big(Q'\Big((K')^\top V\Big)\Big), \qquad \widehat{D} = \mathrm{diag}\Big(Q'\Big((K')^\top \mathbf{1}_N\Big)\Big). \tag{3.34}$$

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 $O(N^2 d)$ i $O(N^2 + Nd)$ w standardowym attention.

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

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

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

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

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

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

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

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

Wtedy:

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

□

3.3.3. Implementacja

Estymator cosh

Zauważmy, że:

$$\exp(t) = \cosh(t) + \sinh(t), \tag{3.40}$$

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

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

Zatem:

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

Otrzymujemy:

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

Porównanie wariancji

Mając zdefiniowane dwa nieobciążone estymatory możemy porównać ich wariancję:

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

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

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

Korzystamy z zależności:

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

Wówczas:

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

Porównujemy teraz wariancję:

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

Estymator exp

W implementacji Performera wykorzystano następującą funkcję ϕ:

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

wtedy

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

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

Modyfikacja

W naszej pracy używamy modyfikacji estymatora exp zaproponowanej i opisanej teoretycznie w [4], lecz nieprzetestowanej eksperymentalnie przez autorów:

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

Wówczas:

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

W ten sposób otrzymujemy estymator o mniejszej wariancji niż oryginalny.

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

Uwaga: Należy zauważyć, że przy użyciu metody cosh macierze Q' oraz K' rosną dwukrotnie, tzn. $Q' \in \mathbb{R}^{N \times 2m}, \quad K' \in \mathbb{R}^{N \times 2m}$, 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:

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

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

Uwaga: W związku z powyższym, parametr, który oznaczamy jako `nb_features` (patrz tab. A.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]

- **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 (*biasu*) 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*), co uniemożliwia modelowi nauczenie się zwrotów charakterystycznych dla konkretnych autorów. [3]

- **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`. [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. Następ-

[1] `https://huggingface.co/datasets/wikimedia/wikipedia/viewer/20231101.en`

[2] `https://huggingface.co/datasets/stanfordnlp/imdb`

[3] `https://zenodo.org/records/1489920`

[4] `https://huggingface.co/datasets/ccdv/arxiv-classification`

nie:

- 450 000 artykułów stokenizowano do maksymalnej długości 128 tokenów[5],
- pozostałe 150 000 artykułów stokenizowano do maksymalnej długości 512 tokenów[5].

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 stokenizowano do maksymalnej długości 512 tokenów[5].

Hyperpartisan News Detection. Ze zbiorów treningowego i walidacyjnego odfiltrowano rekordy o długości mniejszej niż 7000 znaków. Ze względu na ograniczenia sprzętowe i rozmiar danych zbiory zostały losowo zmniejszone, zachowując przy tym oryginalne proporcje klas. Następnie dokonano podziału zbioru walidacyjnego na podzbiory `validation` i `test`. Po wykonaniu powyższych operacji otrzymano trzy zbiory danych:

52 000 (trening) | 6 500 (walidacja) | 6 500 (test).

Dane stokenizowano do maksymalnej długości 4 096 tokenów[5].

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).

[5]Dane zostały stokenizowane z wykorzystaniem `WordPieceTokenizerWrapper` (patrz sek. 2.2).

Dane stokenizowano do maksymalnej długości 16 384 tokenów[5].

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. 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` etykiety wynikają z ogólnego profilu wydawcy, co potencjalnie mogłoby prowadzić do uczenia się sygnałów ubocznych (np. stylu medium) zamiast samej stronniczości treści. Problem ten jest jednak mitygowany przez konstrukcję zbioru – wydawcy są rozdzieleni między podziałami danych, co uniemożliwia modelowi nauczenie się zwrotów charakterystycznych dla konkretnych źródeł.

4.1.4. EDA

Tabela 4.1 przedstawia 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.

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 zapisem stanu modelu (ang. *checkpoint*) w odpowiednim eksperymencie. Następnie realizowana jest adaptacja domenowa (ang. *Task-Adaptive Pretraining*, TAPT [9]), 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

Tabela 4.1: Statystyki długości tokenów po tokenizacji

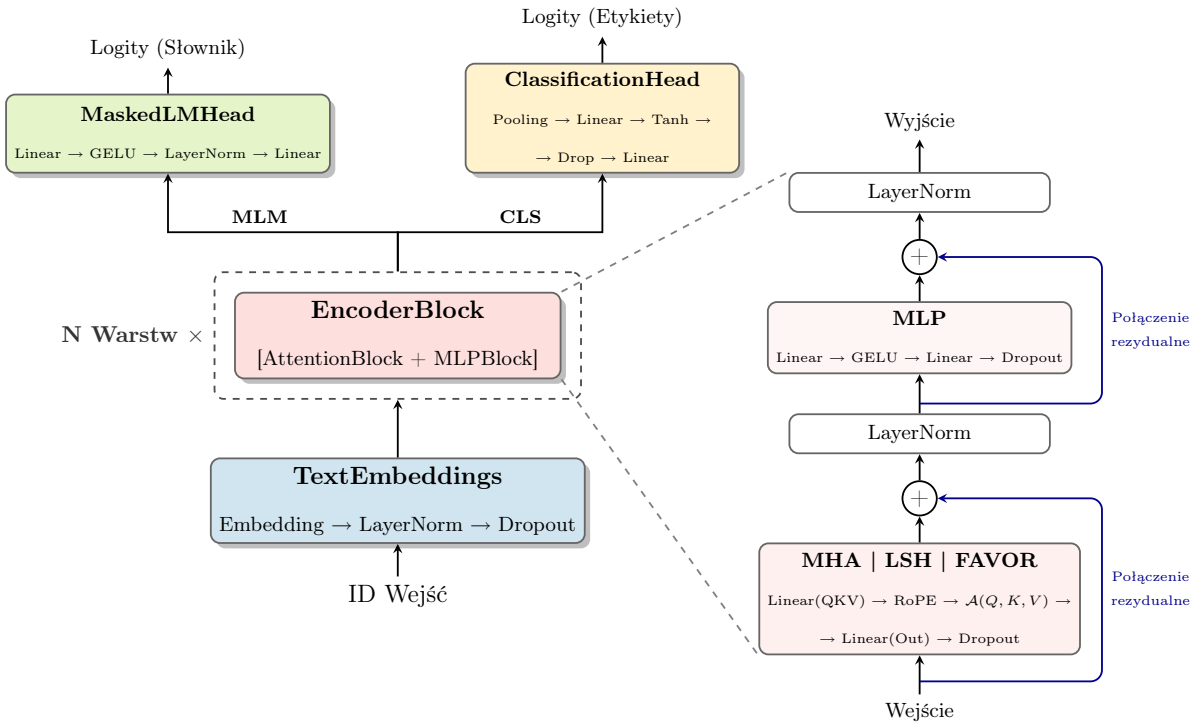
	Statystyki tokenów				
Split	μ	σ	**Med.**	**Min**	**Max**
Wikipedia					
450k (128)	127	6	128	19	128
150k (512)	452	112	512	20	512
IMDb					
trening	263	137	220	10	512
walidacja	263	138	220	18	512
test	262	137	221	13	512
arXiv					
trening	12 095	4 261	12 836	910	16 384
walidacja	12 055	4 321	12 795	650	16 384
test	12 063	4 342	12 863	1 009	16 384
Hyperpartisan					
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

i zapisując stan modelu po eksperymencie TAPT. Ostatnim etapem jest końcowe dostrajanie (ang. *fine-tuning*), w którym model jest trenowany w sposób nadzorowany na konkretnym, etykietowanym zbiorze danych, startując z odpowiedniego zapisanego stanu TAPT, co pozwala na uzyskanie końcowego modelu `model.pt` oraz jego najlepszej wersji `best_model.pt`, wyłonionej na podstawie wyników F1-macro na zbiorze walidacyjnym. Po każdym z etapów następuje transfer wag z odpowiedniego punktu zapisanego stanu, podczas gdy pozostałe parametry, takie jak stan optymalizatora, harmonogram współczynnika uczenia (ang. *learning rate scheduler*) oraz skaler gradientów (ang. *gradient scaler*), są resetowane. Ze względu na wysokie koszty obliczeniowe, każdą konfigurację trenowano jednokrotnie. Przy interpretacji wyników należy uwzględnić, że różnice rzędu dziesiątych części punktu procentowego mogą wynikać z losowości procesu uczenia.

4.2.2. Architektura

Schemat architektury użytej w eksperymentach przedstawiono na rys. 4.1.

Oznaczenia Dla opisu konfiguracji wykorzystywanej architektury przyjmujemy następujące oznaczenia parametrów: l^t – liczba warstw enkodera, d^h – wymiar ukryty, d^f – rozmiar warstwy



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

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 $d^q = d^k = d^v = d^h/h$.

$\text{BERT}_{\text{SMALL}}$ We wszystkich eksperymentach wykorzystano architekturę $\text{BERT}_{\text{SMALL}}$, wprowadzoną w artykule Turc i in. [21]. Konfiguracja: $l^t = 4$, $d^h = 512$, $d^f = 2048$, $h = 8$, $d^{q|k|v} = 512$.

Liczba parametrów W modelu $\text{BERT}_{\text{SMALL}}$ 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). Należy zaznaczyć, że ostatnia warstwa liniowa w głowicy MLM współdzieli wagi z warstwą osadzeń. Całkowitą liczbę parametrów modelu wyrażamy wzorem:

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

gdzie P_{head} to liczba parametrów trenowalnych danej głowicy. Sumaryczna liczba parametrów dla $\text{BERT}_{\text{SMALL}}$ wynosi zatem ok. 28.5 mln (zarówno dla MLM, jak i klasyfikacji).

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	2×10^{-4}	0.5	512	64
	Hyperpartisan	6			4096	8
	Arxiv	2			16384	2
Dostrajanie (klasyfikacja)	IMDb	8	3×10^{-5}	0.2	512	64
	Hyperpartisan	7	2×10^{-5}		4096	64*
	Arxiv	4	3×10^{-4}		16384	16*

* Z wykorzystaniem mechanizmu akumulacji gradientów (8 kroków).

4.2.3. Hiperparametry

Hiperparametry dobrano na podstawie literatury oraz wstępnych eksperymentów. Wartości specyficzne dla zbiorów danych, użyte w poszczególnych etapach treningu, przedstawiono w tab. 4.2, natomiast parametry wspólne dla wszystkich etapów i zbiorów – w tab. 4.3. Szczegółowe opisy poszczególnych parametrów znajdują się w tabelach: A.1, A.2, A.3, A.4 i A.6.

4.2.4. Badane konfiguracje mechanizmów uwagi

Dla mechanizmów uwagi przybliżonej (LSH, FAVOR+) zdefiniowano przestrzeń hiperparametrów specyficznych, przedstawioną w tab. 4.4. Standardowa uwaga SDPA stanowi konfigurację bazową bez dodatkowych hiperparametrów. Każda z wynikowych 9 konfiguracji (1 dla SDPA, 4 dla LSH, 4 dla FAVOR+) przeszła pełny proces uczenia.

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

Jako punkt odniesienia dla eksperymentów z modelami Transformer wytrenowaliśmy na zbiorach treningowych 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. B.1, dla arXiv w tab. B.2, dla Hyperpartisan w tab. B.3.

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
Pretrening	`tie_mlm_weights`	true
	`mask_p`	0.15
	`mask_token_p`	0.8
	`random_token_p`	0.1
Dostrajanie	`pooler_type`	bert
	`head_lr_mult`	1.0
	`backbone_lr_mult`	1.0
Trening	`warmup_ratio`	0.1
	`weight_decay`	0.01
	`max_grad_norm`	1.0
	`loss`	cross_entropy

Tabela 4.4: Konfiguracje hiperparametrów dla badanych mechanizmów uwagi

Mechanizm	**Hiperparametr**	**Badane wartości**
LSH	Liczba haszy (N_h)	$\{2, 4\}$
	Wielkość bloku (C)	$\{64, 128\}$
FAVOR+	Liczba losowych cech (N_f)	$\{0.125, 0.25, 0.5, 1.0\} \times d^{q\|k\|v}$

4.2.6. Środowisko eksperymentalne i zużyte zasoby

Wszystkie eksperymenty przeprowadzono w środowisku Google Colab z wykorzystaniem karty NVIDIA A100 (40 GB pamięci VRAM). Pełny cykl eksperymentów obejmujący wszystkie etapy (4.3, 4.4, 4.5), zajął łącznie 169.48 godzin czasu GPU. Przekłada się to na zużycie około 910 jednostek obliczeniowych (ang. *compute units*) Google Colab, przy koszcie 5.37 jednostek obliczeniowych za godzinę.

Tabela 4.5: Porównanie mechanizmów uwagi w pretreningu na zbiorze Wikipedia: zużycie VRAM, średni czas jednej epoki oraz minimalna wartość funkcji straty (średnia z epoki) na zbiorze treningowym, wyznaczona jako minimum po wszystkich epokach.

Konfiguracja	**Max VRAM [GB]**	**Czas/epoka [min]**	**Min. loss**
SDPA	14.44	7.88	2.157
LSH			
N_h=2, C=64	19.09	12.75	2.345
N_h=2, C=128	21.34	18.15	2.278
N_h=4, C=64	24.38	17.94	2.286
N_h=4, C=128	28.88	28.39	2.248
FAVOR+			
N_f=0.125	16.62	10.99	3.069
N_f=0.25	18.40	12.12	2.694
N_f=0.5	21.97	15.29	2.666
N_f=1.0	29.10	21.73	2.579

4.3. Etap 1: Wstępny pretrening

Celem pierwszego etapu jest nauczenie modelu ogólnych struktur językowych i semantyki poprzez zadanie MLM na korpusie Wikipedii (patrz sek. 4.1.2). Trenowano wszystkie 9 konfiguracji modelu $\text{BERT}_{\text{SMALL}}$ opisanych w sek. 4.2.4. Hiperparametry treningu przedstawiono w tab. 4.2 i tab. 4.3.

4.3.1. Wyniki

Tabela 4.5 przedstawia wyniki wstępnego pretreningu, uwzględniając minimalną wartość funkcji straty na zbiorze treningowym z całej epoki, czas trwania epoki oraz maksymalne zużycie pamięci VRAM dla poszczególnych konfiguracji.

4.3.2. Podsumowanie wyników

- **SDPA** osiągnęła najlepsze wyniki we wszystkich trzech metrykach: najniższą wartość funkcji straty (2.157), najkrótszy czas epoki (7.88 min) oraz najmniejsze zużycie VRAM (14.44 GB).
- **LSH** – wartość funkcji straty zbliżona do SDPA (2.25–2.35), jednak kosztem znacznie większych zasobów: zużycie VRAM wzrosło o 32–100%, a czas treningu o 62–260%. Zwiększanie

parametrów N_h i C obniża wartość funkcji straty, ale zwiększa koszt obliczeniowy.

- **FAVOR+** – zwiększanie liczby losowych cech obniża wartość funkcji straty, ale nawet najlepsza konfiguracja (N_f=1.0) osiąga stratę ok. 20% wyższą niż SDPA, a zużycie zasobów jest porównywalne z najcięższą konfiguracją LSH.

4.4. Etap 2: Adaptacja domenowa

Celem drugiego etapu jest adaptacja domenowa (TAPT) poprzez kontynuację treningu MLM na nieetykietowanych danych docelowych: IMDb, Hyperpartisan oraz ArXiv (patrz sek. 4.1.2). Każdy z 9 punktów kontrolnych z Etapu 1 został wykorzystany jako punkt startowy dla 3 niezależnych procesów TAPT – po jednym dla każdego zbioru danych – co daje łącznie 27 modeli. Hiperparametry treningu przedstawiono w tab. 4.2 i tab. 4.3.

4.4.1. Wyniki

Tabela 4.6 przedstawia wyniki adaptacji domenowej, uwzględniając minimalną wartość funkcji straty na zbiorze treningowym z całej epoki, czas trwania epoki oraz maksymalne zużycie pamięci VRAM dla poszczególnych konfiguracji i zbiorów danych.

4.4.2. Podsumowanie wyników

- **SDPA** ponownie osiągnęła najniższą wartość funkcji straty na wszystkich trzech zbiorach danych.

- **LSH** – na zbiorze ArXiv (długość sekwencji – 16384 tokenów) osiągnął krótszy czas epoki niż SDPA – najlepsza konfiguracja (N_h=2, C=64) trenowała się o 46% szybciej (38.15 vs 70.10 min). Na krótszych sekwencjach (IMDb, Hyperpartisan) LSH pozostaje wolniejszy od SDPA.

- **FAVOR+** – wartość funkcji straty na ArXiv jest znacznie wyższa niż dla SDPA i LSH. Jednocześnie FAVOR+ oferuje najkrótsze czasy treningu na ArXiv (ale niewiele niższe niż LSH).

Tabela 4.6: Porównanie mechanizmów uwagi w adaptacji domenowej na poszczególnych zbiorach danych: zużycie VRAM, średni czas jednej epoki oraz minimalna wartość funkcji straty (średnia z epoki) na zbiorze treningowym, wyznaczona jako minimum po wszystkich epokach.

	VRAM [GB]			**Czas [min]**			**Min loss**		
Model	**IMDb**	**Hyper.**	**Arxiv**	**IMDb**	**Hyper.**	**Arxiv**	**IMDb**	**Hyper.**	**Arxiv**
SDPA	14.44	14.44	14.45	1.24	16.65	70.10	2.388	1.887	1.672
LSH									
N_h=2, C=64	19.09	19.09	19.09	1.97	18.60	38.15	2.603	2.557	2.300
N_h=2, C=128	21.34	21.34	21.34	2.28	21.57	43.59	2.502	2.440	2.224
N_h=4, C=64	24.38	24.38	24.39	2.80	26.82	55.01	2.535	2.351	2.072
N_h=4, C=128	28.88	28.88	28.89	3.41	32.80	65.84	2.478	2.263	2.024
FAVOR+									
N_f=0.125	16.62	16.59	16.59	1.52	15.51	34.51	3.200	4.505	5.033
N_f=0.25	18.40	18.35	18.34	1.71	17.60	38.75	2.866	3.914	4.797
N_f=0.5	21.96	21.85	21.85	2.16	22.62	47.39	2.834	3.665	4.632
N_f=1.0	29.09	28.87	28.85	3.37	32.04	64.92	2.762	3.268	4.316

4.5. Etap 3: Dostrajanie

Trzeci, ostatni etap procesu polega na uczeniu nadzorowanym (klasyfikacji) na etykietowanych zbiorach danych. Modele, po wstępnym pretreningu i adaptacji domenowej, są dostrajane do realizacji konkretnych zadań klasyfikacji (sentyment, stronniczość polityczna, kategoria tematyczna).

Hiperparametry treningu dla tego etapu znajdują się w tab. 4.2 i tab. 4.3. Przed właściwym treningiem wszystkich wariantów przeprowadzono jednak procedurę doboru parametrów specyficznych dla dostrajania.

4.5.1. Dobieranie parametrów dostrajania

W tej fazie wyznaczamy optymalne hiperparametry procesu dostrajania dla architektury $\text{BERT}_{\text{SMALL}}$, z wykorzystaniem mechanizmu uwagi SDPA. Przetestowano wpływ trzech czynników:

1. **Liczba zamrożonych warstw enkodera**: Odnosi się do dolnych bloków enkodera. Dla wartości > 0 wskazane warstwy oraz warstwa osadzeń pozostają zamrożone przez pierwsze $\lfloor \text{liczba epok}/2 \rfloor$ epok dostrajania. Dla wartości 0 wszystkie parametry są aktualizo-

Tabela 4.7: Przestrzeń poszukiwań hiperparametrów etapu dostrajania

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

wane od początku. Dotyczy to parametrów: `freeze`, `freeze_n_layers`, `freeze_epochs`, `freeze_embeddings` w tab. A.4.

2. **Dropout w głowicy klasyfikacyjnej**: Wartość dropoutu przed ostatnią projekcją na liczbę klas – parametr `classifier_dropout` w tab. A.6.

3. **Metoda agregacji (Pooling)**: Sposób tworzenia reprezentacji całego tekstu – parametr `pooling` w tab. A.6.

Przestrzeń poszukiwań zamieszczono w tab. 4.7.

Decyzję o wyborze optymalnej konfiguracji podjęto na podstawie metryki F1-macro na zbiorze walidacyjnym, niezależnie dla każdego zbioru danych. Wyniki wszystkich rozważanych konfiguracji zostały przedstawione w tab. 4.8.

Wyłonione w ten sposób optymalne parametry (zestawione w tab. 4.9) zostały wykorzystane w głównych eksperymentach klasyfikacji dla wszystkich badanych mechanizmów uwagi.

4.5.2. Wyniki

W tab. 4.10 przedstawiono wyniki klasyfikacji dla wszystkich 27 modeli, uwzględniając metrykę F1-macro [%] na zbiorze testowym, średni czas trwania epoki oraz maksymalne zużycie pamięci VRAM. Jako punkt odniesienia zamieszczono również wyniki klasyfikatora bazowego TF-IDF + regresja logistyczna.

4.5.3. Podsumowanie wyników

- **Wszystkie modele Transformer** znacząco przewyższyły baseline TF-IDF, szczególnie na zbiorze Hyperpartisan (poprawa o 13–23 pp F1-macro).

- **SDPA** osiągnęło najwyższy wynik F1 na zbiorze ArXiv (88.38%), przy najniższym zużyciu VRAM (3.32–3.44 GB). Na zbiorach IMDb i Hyperpartisan nieznacznie ustąpiło konfigu-

Tabela 4.8: Wyniki F1-macro [%] na zbiorze walidacyjnym dla różnych konfiguracji hiperparametrów.

Model	**IMDb**	**Hyperpartisan**	**Arxiv**
Agregacja CLS			
$N_{\text{freeze}} = 0$, $P_{\text{drop}} = 0.1$	93.48	64.06	88.06
$N_{\text{freeze}} = 0$, $P_{\text{drop}} = 0.2$	93.44	64.34	88.12
$N_{\text{freeze}} = 1$, $P_{\text{drop}} = 0.1$	93.68	64.06	88.24
$N_{\text{freeze}} = 1$, $P_{\text{drop}} = 0.2$	93.70	64.74	88.40
$N_{\text{freeze}} = 2$, $P_{\text{drop}} = 0.1$	93.44	60.35	88.26
$N_{\text{freeze}} = 2$, $P_{\text{drop}} = 0.2$	93.46	60.20	88.54
Agregacja Mean			
$N_{\text{freeze}} = 0$, $P_{\text{drop}} = 0.1$	93.80	65.34	88.23
$N_{\text{freeze}} = 0$, $P_{\text{drop}} = 0.2$	93.72	65.38	88.14
$N_{\text{freeze}} = 1$, $P_{\text{drop}} = 0.1$	93.84	65.44	89.16
$N_{\text{freeze}} = 1$, $P_{\text{drop}} = 0.2$	93.80	64.52	89.32
$N_{\text{freeze}} = 2$, $P_{\text{drop}} = 0.1$	93.86	64.48	88.90
$N_{\text{freeze}} = 2$, $P_{\text{drop}} = 0.2$	93.88	63.78	88.87

Tabela 4.9: Optymalne hiperparametry procesu dostrajania.

Zbiór danych	$\boldsymbol{N_{freeze}}$	$\boldsymbol{P_{drop}}$	**Agregacja**
IMDb	2	0.2	Mean
Hyperpartisan	1	0.1	Mean
ArXiv	1	0.2	Mean

racji LSH (N_h=4, C=128) dla IMDb (92.98% vs 92.86%) oraz LSH (N_h=2, C=64) dla Hyperpartisan (65.37% vs 64.55%).

- **LSH** na zbiorze ArXiv oferuje istotną przewagę czasową – najszybsza konfiguracja (N_h=2, C=64) trenowała się około 2.1× szybciej niż SDPA (25.93 vs 55.15 min/epoka), jednocześnie tracąc 1.5 pp F1. Wyniki F1-macro na IMDb są porównywalne z SDPA (92.5–93.0%).

- **FAVOR+** uzyskał najsłabsze wyniki F1 na wszystkich zbiorach danych. Na IMDb i ArXiv strata względem SDPA wynosi odpowiednio 1.1–1.9 pp oraz 1.9–2.8 pp. Szczególnie słabe wyniki FAVOR+ osiągnął na zbiorze Hyperpartisan – strata względem SDPA sięga 5.3–11.1 pp. Jednocześnie FAVOR+ oferuje przewagę czasową dla długich sekwencji (ArXiv: 22–52 min vs 55 min dla SDPA).

Tabela 4.10: Porównanie mechanizmów uwagi przy dostrajaniu: VRAM, czas na epokę i F1 macro [%] dla różnych zbiorów.

	VRAM [GB]			**Czas [min]**			**F1 [%]**		
Model	**IMDb**	**Hyper.**	**Arxiv**	**IMDb**	**Hyper.**	**Arxiv**	**IMDb**	**Hyper.**	**Arxiv**
TF-IDF + LR	–	–	–	–	–	–	89.50	42.23	83.62
SDPA	3.32	3.44	3.44	0.64	10.33	55.15	92.86	64.55	88.38
LSH									
N_h=2, C=64	9.04	9.16	9.17	1.35	12.61	25.93	92.78	65.37	86.90
N_h=2, C=128	11.76	11.88	11.89	1.65	15.64	31.35	92.66	61.25	87.30
N_h=4, C=64	15.89	16.01	16.02	2.13	20.88	43.07	92.50	62.47	86.68
N_h=4, C=128	21.33	21.45	21.46	2.70	26.94	53.94	92.98	55.68	87.40
FAVOR+									
N_f=0.125	5.50	5.59	5.59	0.94	9.37	22.14	91.16	53.41	86.02
N_f=0.25	7.28	7.34	7.34	1.12	11.48	26.22	90.98	56.99	86.52
N_f=0.5	10.85	10.85	10.84	1.52	16.40	34.66	91.64	59.30	86.53
N_f=1.0	18.40	18.36	18.34	2.57	25.62	51.61	91.74	55.02	85.57

4.6. Analiza i porównanie wyników

4.6.1. Analiza

W niniejszej sekcji przedstawiono zbiorczą analizę wyników ze wszystkich etapów eksperymentu. Tabele 4.12 i 4.13 zestawiają względne różnice w zużyciu pamięci VRAM oraz czasie treningu dla mechanizmów przybliżonych względem SDPA. Różnice procentowe w czasie treningu obliczono na podstawie całkowitego czasu obu etapów (suma: liczba epok TAPT × czas epoki TAPT + liczba epok dostrajania × czas epoki dostrajania), a nie pojedynczych epok. Tabela 4.14 podsumowuje wyniki klasyfikacji F1-macro w odniesieniu do baseline'u oraz SDPA.

Aby zidentyfikować konfiguracje oferujące najlepszy kompromis między jakością a kosztem obliczeniowym, przeprowadziliśmy analizę Pareto. Konfiguracja należy do frontu Pareto, jeśli żadna inna konfiguracja nie jest od niej jednocześnie lepsza pod względem wszystkich trzech kryteriów: F1-macro, czasu treningu oraz zużycia VRAM. Analizę przeprowadzono dla etapów TAPT i dostrajania łącznie, osobno dla każdego zbioru danych. Wyniki przedstawiono w tab. 4.11 Ze względu na brak replikacji eksperymentów, analiza Pareto ma charakter orientacyjny – konfiguracje o zbliżonych wartościach F1-macro (różnice < 1 pp) mogłyby zmienić pozycję względem frontu przy powtórzeniu z inną inicjalizacją.

Tabela 4.11: Analiza Pareto dla etapów TAPT + dostrajanie. Symbol ✓ oznacza przynależność do frontu Pareto (optymalizacja: max F1, min czas, min VRAM).

Konfiguracja	**IMDb**	**Hyperpartisan**	**ArXiv**
SDPA	✓	✓	✓
LSH			
N_h=2, C=64	–	✓	✓
N_h=2, C=128	–	–	✓
N_h=4, C=64	–	–	–
N_h=4, C=128	✓	–	✓
FAVOR+			
N_f=0.125	–	✓	✓
N_f=0.25	–	–	✓
N_f=0.5	–	–	–
N_f=1.0	–	–	–

Tabela 4.12: Porównanie maksymalnego zużycia pamięci VRAM: pretrening na Wikipedii oraz TAPT+dostrajanie na zbiorach docelowych. Wartości pokazują różnicę procentową względem SDPA.

	pretrening	**TAPT + dostrajanie**		
Model	**Wikipedia**	**IMDb**	**Hyper.**	**Arxiv**
LSH				
N_h=2, C=64	+32.2%	+32.2%	+32.2%	+32.2%
N_h=2, C=128	+47.8%	+47.8%	+47.8%	+47.7%
N_h=4, C=64	+68.9%	+68.9%	+68.9%	+68.8%
N_h=4, C=128	+100.0%	+100.0%	+100.0%	+100.0%
FAVOR+				
N_f=0.125	+15.1%	+15.1%	+14.9%	+14.9%
N_f=0.25	+27.4%	+27.4%	+27.1%	+27.0%
N_f=0.5	+52.1%	+52.1%	+51.3%	+51.2%
N_f=1.0	+101.5%	+101.5%	+99.9%	+99.7%

Tabela 4.13: Porównanie czasu treningu: pretrening na Wikipedii oraz całkowity czas etapów TAPT i dostrajanie na zbiorach docelowych. Wartości pokazują różnicę procentową względem SDPA.

	pretrening	**TAPT + dostrajanie**		
Model	**Wikipedia**	**IMDb**	**Hyper.**	**Arxiv**
LSH				
N_h=2, C=64	+61.8%	+70.2%	+16.1%	-50.1%
N_h=2, C=128	+130.3%	+100.0%	+38.7%	-41.1%
N_h=4, C=64	+127.7%	+149.1%	+78.3%	-21.8%
N_h=4, C=128	+260.3%	+207.2%	+123.8%	-3.7%
FAVOR+				
N_f=0.125	+39.5%	+27.8%	-7.9%	-56.3%
N_f=0.25	+53.8%	+46.0%	+8.0%	-49.5%
N_f=0.5	+94.0%	+87.9%	+45.5%	-35.3%
N_f=1.0	+175.8%	+199.8%	+115.8%	-6.8%

Tabela 4.14: F1 macro [%] - porównanie mechanizmów uwagi z baseline TF-IDF+LR i SDPA. Dla TF-IDF+LR i SDPA podano wartości bezwzględne, dla LSH i FAVOR+ podano różnicę w punktach procentowych (pp) względem SDPA i względem TF-IDF+LR.

Model	**IMDb**	**Hyper.**	**Arxiv**
TF-IDF + LR	89.50	42.23	83.62
SDPA (vs TF-IDF+LR)	92.86 (+3.4)	64.55 (+22.3)	88.38 (+4.8)
LSH (vs SDPA / vs TF-IDF+LR)			
N_h=2, C=64	-0.1 / +3.3	+0.8 / +23.1	-1.5 / +3.3
N_h=2, C=128	-0.2 / +3.2	-3.3 / +19.0	-1.1 / +3.7
N_h=4, C=64	-0.4 / +3.0	-2.1 / +20.2	-1.7 / +3.1
N_h=4, C=128	+0.1 / +3.5	-8.9 / +13.4	-1.0 / +3.8
FAVOR+ (vs SDPA / vs TF-IDF+LR)			
N_f=0.125	-1.7 / +1.7	-11.1 / +11.2	-2.4 / +2.4
N_f=0.25	-1.9 / +1.5	-7.6 / +14.8	-1.9 / +2.9
N_f=0.5	-1.2 / +2.1	-5.2 / +17.1	-1.9 / +2.9
N_f=1.0	-1.1 / +2.2	-9.5 / +12.8	-2.8 / +1.9

4.6.2. Wnioski

Poniżej granicy 4000 tokenów SDPA dominuje we wszystkich trzech wymiarach – jest szybsza, zużywa mniej pamięci i osiąga konkurencyjną jakość.

Mechanizmy przybliżone zwiększają zużycie pamięci VRAM o $15-100\%$ w stosunku do SDPA. Ich przewagą jest przyspieszenie dla długich sekwencji, przy czym LSH oferuje lepszy stosunek jakości do czasu niż FAVOR+. Dla sekwencji długości 16384 tokenów (ArXiv), LSH w konfiguracji (N_h=2, C=64) oferuje redukcję czasu o 50% przy niewielkiej stracie jakości (1.5 pp F1-macro) względem SDPA. Metoda FAVOR+, pomimo osiągania najkrótszych czasów treningu, wykazuje się niższą jakością, zauważalną już na etapie pretreningu modelu.

4.7. Eksperymenty dodatkowe

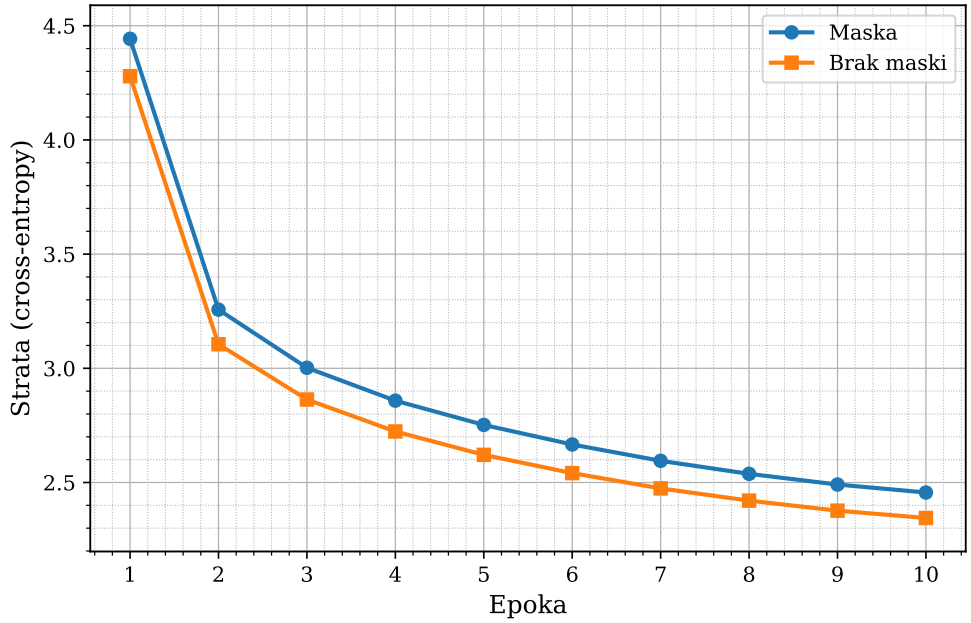
4.7.1. Hybrydowe podejście do treningu

Architektura FAVOR+ umożliwia zamianę mechanizmu uwagi bez konieczności ponownego pretreningu, ponieważ projekcje Q, K, V pozostają zgodne ze standardową uwagą. Zbadano podejście hybrydowe: wykorzystanie zapisanego stanu modelu SDPA z pretreningu na Wikipedii, a następnie zamiana mechanizmu na FAVOR+ (N_f=0.25) wyłącznie dla etapów TAPT i dostrajania na zbiorze ArXiv. Wyniki przedstawiono w tab. 4.15.

Tabela 4.15: Porównanie podejścia hybrydowego (SDPA → FAVOR+) z pełnym treningiem FAVOR+ i SDPA na zbiorze ArXiv.

Konfiguracja	**Czas pretrening [min]**	**Czas TAPT [min]**	**Czas dostrajania [min]**	**F1 [%]**
SDPA	78.82	140.20	220.58	88.40
FAVOR+	121.20	77.52	104.87	86.52
SDPA → FAVOR+	78.82	77.90	104.65	86.23

Podejście hybrydowe pozwoliło uzyskać całkowity czas treningu wynoszący 261.37 min ($78.82+77.90+104.65$), co stanowi redukcję o 40.5% względem SDPA (439.60 min) oraz o 13.9% względem pełnego FAVOR+ (303.59 min). Strata jakości względem SDPA wynosi 2.17 pp F1-macro, natomiast względem pełnego FAVOR+ model hybrydowy osiąga porównywalny wynik (86.23% vs 86.52%, różnica 0.29 pp). Podejście to może być wykorzystane w scenariuszu, gdy mamy dostęp do zapisanego stanu modelu SDPA i priorytetem jest minimalizacja czasu adaptacji domenowej oraz dostrajania.



Rysunek 4.2: Porównanie pretreningu LSH na zbiorze Wikipedia w zależności od maskowania wewnątrz koszyków. Wykres przedstawia przebieg straty cross-entropy w kolejnych epokach dla dwóch wariantów: `mask_withing_chunks=true` (*maska*) oraz `mask_withing_chunks=false` (*brak maski*).

4.7.2. Maskowanie wewnątrz bloku LSH

Przetestowaliśmy modyfikację opisaną w sek. 3.2.3. Rysunek 4.2 przedstawia porównanie pretreningu LSH – na zbiorze Wikipedia w konfiguracji `num_hashes = 2, chunk_size = 64` – w zależności od tego, czy maskujemy uwagę wewnątrz koszyków. Wersja bez maski uczy się szybciej – niższa wartość funkcji straty w poszczególnych epokach (czas treningu różni się w stopniu znikomym).

Interpretacja Dla elementów j znajdujących się w tym samym (lub sąsiednim) bloku, lecz w innym koszyku co token i rozważmy dwa scenariusze: (1) Jeśli j nie jest podobny do i ($q_i \cdot k_j$ jest małe), to jego wkład do Softmax jest niewielki, włączenie takich elementów można traktować jako formę regularizacji, ograniczając nadmierne „skupienie" uwagi wyłącznie na elementach z własnego koszyka. (2) Jeśli natomiast j jest podobny do i ($q_i \cdot k_j$ jest duże), ale trafił do innego koszyka, to usunięcie maski pozwala nadal uwzględnić go w uwadze, co redukuje negatywne skutki błędów haszowania.

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
Core			
`__init__`	5	0	100%
Logger			
`__init__`	1	0	100%
`wandb_logger`	98	16	84%
Tokenizer			
`__init__`	1	0	100%
`wordpiece_tokenizer_wrapper`	117	8	93%
Training			
`__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%
Modele – ogólne i bloki			
`__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%
Mechanizmy uwagi			
`multihead_sdp_self_attention`	79	4	95%
`multihead_favor_self_attention`	180	13	93%
`multihead_lsh_self_attention`	145	3	98%
Embeddingi			
`positional_encodings`	24	0	100%
`rotary`	27	0	100%
`text_embeddings`	57	1	98%
Pooling i Głowice			
`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**
WNF-1 – Wydajność i efektywność zasobowa		
Środowisko GPU (Colab)	Spełnione	Wykorzystano GPU *A100 40GB*
Wymóg kosztowy (Performer)	Częściowo spełnione	Zgodnie z tab. 4.14, 4.13 i 4.12
Wymóg kosztowy (Reformer)	Częściowo spełnione	Zgodnie z tab. 4.14, 4.13 i 4.12
Techniki optymalizacji	Spełnione	Zgodnie z sek. 2.3.2
WNF-2 – Jakość, niezawodność i testowalność		
Jakość (SDPA)	Spełnione	Zgodnie z tab. 4.14
Jakość (Performer)	Częściowo spełnione	Zgodnie z tab. 4.14
Jakość (Reformer)	Spełnione	Zgodnie z tab. 4.14
Stabilność	Spełnione	Zgodnie z tab. C.1
Testy komponentów	Spełnione	Zgodnie z sek. 5.1
WNF-3 – Użyteczność i utrzymanie		
Dokumentacja	Spełnione	Zgodnie z sek. 2.7
Struktura katalogów	Spełnione	Zgodnie z sek. 2
Zgodność z PEP-8	Spełnione	Kod źródłowy w załączniku D
Wersjonowanie	Spełnione	Kod był wersjonowany z wykorzystaniem Git.
Rozszerzalność mechanizmów uwagi	Spełnione	Zgodnie z sek. 2.1.2
Konfiguracja YAML	Spełnione	Zgodnie z sek. 2.5
WNF-4 – Przenośność i kompatybilność		
Kompatybilność Python	Spełnione	Zgodnie z sek. 2.6
Biblioteki	Spełnione	Zgodnie z sek. 2.6
WNF-5 – Monitorowanie i obserwowalność		
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**
WF-1 – Pretrening i dostrajanie		
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
WF-2 — Wymienne mechanizmy uwagi		
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
WF-3 — Obsługa długich sekwencji		
Kodowanie pozycyjne	Spełnione	Zgodnie z sek. 2.1.4
Obsługa sekwencji > 512	Spełnione	Parametr `max_length` przyjmuje dowolną wartość (tab. A.1).
WF-4 — Pipeline danych		
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
WF-5 — Konfiguracja		
Generator eksperymentów	Spełnione	Zgodnie z sek. 2.5
Separacja pretreningu i dostrajania	Spełnione	Zgodnie z sek. 2.5

6. Podsumowanie

6.1. Doświadczenie projektowe

Realizacja niniejszego projektu stanowiła kompleksowe przedsięwzięcie inżynierskie, łączące teorię głębokiego uczenia ze standardami wytwarzania oprogramowania. Poniżej przedstawiono kluczowe obszary kompetencji oraz wnioski płynące z realizacji prac.

6.1.1. Głębokie Uczenie i NLP

Najistotniejszym elementem projektu była implementacja architektury Transformer, bez polegania na gotowych abstrakcjach modelowych.

- **Zrozumienie mechanizmu uwagi:** Implementacja klasycznego *Scaled Dot-Product Attention* oraz jego wariantów: *LSH* (Reformer) i *FAVOR+* (Performer).
- **Pełny cykl treningowy:** Praktyczne opanowanie wieloetapowego procesu uczenia modeli NLP, obejmującego pretrening na korpusie ogólnym (MLM), adaptację do domeny (TAPT) oraz końcowe dostrajanie na zadaniu klasyfikacji.

6.1.2. Inżynieria Oprogramowania

Implementacja systemu opiera się na nowoczesnych praktykach inżynierii oprogramowania:

- **Architektura modułowa:** Obsługa wielu wariantów mechanizmu uwagi zapewnia łatwość rozszerzania systemu o nowe komponenty.
- **Weryfikacja jakości:** Poprawność kodu potwierdzono poprzez testy jednostkowe zaimplementowane w środowisku `pytest`, pokrywające kluczowe moduły aplikacji.

6.1.3. Zarządzanie Eksperymentami

Istotnym aspektem pracy było stworzenie środowiska badawczego:

- **Zarządzanie konfiguracją:** Wykorzystanie szablonów YAML i skryptów generujących (`generate_*_experiment.py`) do automatyzacji tworzenia struktury eksperymentów.

- **Śledzenie eksperymentów:** Integracja z platformą *Weights & Biases* (W&B) umożliwiająca monitorowanie metryk eksperymentów, zużycia zasobów systemowych oraz porównywanie wielu przebiegów.

Podsumowując, projekt ten pozwolił na zdobycie praktycznego doświadczenia w pełnym cyklu badania modelu uczenia maszynowego: od implementacji algorytmów, przez inżynierię oprogramowania, aż po zarządzanie procesem badawczym i ewaluację wyników.

6.2. Ograniczenia

Ze względu na koszty obliczeniowe pretrening naszego modelu był znacznie krótszy niż w oryginalnej pracy o BERT [7]. Podczas pretreningu na Wikipedii model przetworzył około 1.3 mld tokenów (10 epok $\times$ (450k $\times$ 128 + 150k $\times$ 512)), podczas gdy w oryginalnym BERT było to około 42.6 mld tokenów. Co więcej, nasz korpus tekstowy obejmował zaledwie 600 tys. artykułów Wikipedii, czyli około 9% całego korpusu (6.4 mln artykułów). Dodatkowo w BERT wykorzystano nie tylko Wikipedię, ale także BooksCorpus. W rezultacie nasze modele nie pokazują pełnego potencjału, jaki można uzyskać dzięki dłuższemu pretreningowi na większym i bardziej zróżnicowanym zbiorze danych, a następnie przenieść na zadania docelowe. Z powodu kosztów obliczeniowych, każdą konfigurację trenowano jednokrotnie, co oznacza, że różnice rzędu dziesiątych części punktu procentowego mogą wynikać z losowości procesu uczenia. Zbadano tylko trzy długości sekwencji (512, 4096, 16384 tokenów). Eksperymenty przeprowadzono wyłącznie na architekturze $\text{BERT}_{\text{SMALL}}$ – zachowanie mechanizmów uwagi może różnić się dla większych modeli. Zbadano trzy zbiory danych o zróżnicowanej charakterystyce, jednak wyniki mogą nie generalizować się na inne zbiory danych oraz inne zadania NLP. Dodatkowo, badane mechanizmy uwagi przybliżonej nie były w żaden sposób niskopoziomowo optymalizowane, w przeciwieństwie do natywnej implementacji SDPA w PyTorch. Ponadto, proces optymalizacji hiperparametrów dostrajania modeli był prowadzony z wykorzystaniem mechanizmu SDPA, co potencjalnie mogło faworyzować go względem mechanizmów FAVOR+ i LSH.

6.3. Kierunki dalszych prac

6.3.1. Rozszerzenie zbioru mechanizmów uwagi systemu

System został zaprojektowany w sposób umożliwiający łatwe dodawanie nowych wariantów warstwy uwagi. Wartościowym rozszerzeniem byłaby implementacja mechanizmów wykorzystu-

jących rzadkie wzorce uwagi (ang. *sparse attention*), znanych m.in. z BigBird, a także innych nowoczesnych architektur, np. DeepSeek.

Interesującą alternatywą dla mechanizmu FAVOR+ może być implementacja metod aproksymacji uwagi opartej na innej funkcji mapującej $\phi(\cdot)$ oraz metod opartych na projekcjach niskowymiarowych, takich jak Linformer.

6.3.2. Niskopoziomowa optymalizacja mechanizmów przybliżonych

Jak pokazują nasze eksperymenty, *scaled dot-product attention* (SDPA) osiąga znaczną przewagę czasową wtedy, gdy może zostać wykonana przez wyspecjalizowane backendy SDPA typu FlashAttention. Z kolei mechanizmy przybliżone, takie jak uwaga oparta o LSH oraz FAVOR+, mimo korzystniejszej złożoności asymptotycznej, w implementacji opartej wyłącznie o standardowe operatory PyTorch, uzyskują znaczne przyspieszenie dopiero przy bardzo długich sekwencjach. Ogranicza je przede wszystkim narzut wielu uruchomień jąder oraz koszty transferów i materializacji pośrednich wyników w pamięci globalnej GPU.

Obiecującym kierunkiem jest zaprojektowanie wariantów liczenia uwagi, wykorzystujących mechanizmy stosowane w FlashAttention, które minimalizują ruch danych między pamięcią globalną (HBM) a pamięcią on-chip.

6.3.3. Hybrydowe podejście do przetwarzania sekwencji

Sekcja 4.7.1 przedstawiona wstępne wyniki podejścia hybrydowego polegającego na przełączaniu mechanizmu uwagi pomiędzy etapami treningu. Dalsze prace mogłyby pogłębić tę koncepcję poprzez analizę innych rodzin mechanizmów uwagi.

6.3.4. Walidacja statystyczna

Ze względu na losowość procesu uczenia pojedynczy przebieg treningu może nie być reprezentatywny. W przyszłości warto przeprowadzić wielokrotne treningi dla każdej konfiguracji, z wykorzystaniem różnych ziaren losowości, raportować średnią oraz odchylenie standardowe metryk, a także zastosować testy istotności statystycznej w celu sprawdzenia, czy niewielkie różnice pomiędzy wynikami poszczególnych konfiguracji są statystycznie istotne, czy też wynikają z przypadku.

6.3.5. Analiza interpretowalności

Ponieważ rozważane mechanizmy różnią się sposobem dystrybucji uwagi, wartościowe byłoby porównanie wzorców uwagi pomiędzy modelami – wizualizacje macierzy uwagi dla wybranych

przykładów, analiza koncentracji na tokenach specjalnych oraz preferencji lokalnych vs dalekozasięgowych.

Dodatkowej analizy wymaga mechanizm FAVOR+, którego wyniki klasyfikacji okazały się słabsze, zwłaszcza na zbiorze Hyperpartisan. Co więcej, na Hyperpartisan wyniki wszystkich modeli cechowała duża wariancja, dlatego warto sprawdzić możliwe przyczyny – np. czy modele (lub mechanizmy uwagi) mają tendencję do zapamiętywania słów kluczowych zamiast uczenia się ogólnych reprezentacji języka.

6.3.6. Porównanie z gotowymi modelami

Aby oszacować, jak dużo tracimy przez skrócony pretrening i ograniczony korpus, warto porównać nasze modele z dostępnymi checkpointami, np. BERT-small z biblioteki Hugging Face na zbiorze IMDB (gotowy model obsługuje sekwencje do 512 tokenów) oraz z innymi modelami na dłuższych zbiorach danych – np. Longformer – w celach wyłącznie porównawczych.

Bibliografia

[1] Marah Abdin i in. *Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone*. 2024. arXiv: 2404.14219 [cs.CL]. URL: https://arxiv.org/abs/2404.14219.

[2] Iz Beltagy, Matthew E Peters i Arman Cohan. "Longformer: The long-document transformer". W: *arXiv preprint arXiv:2004.05150* (2020).

[3] Rewon Child. "Generating long sequences with sparse transformers". W: *arXiv preprint arXiv:1904.10509* (2019).

[4] Krzysztof Marcin Choromanski i in. "Rethinking Attention with Performers". W: *International Conference on Learning Representations*. 2021.

[5] Tri Dao. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning". W: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=mZn2Xyh9Ec.

[6] 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.

[7] 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.

[8] Albert Gu i Tri Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. 2024. arXiv: 2312.00752 [cs.LG]. URL: https://arxiv.org/abs/2312.00752.

[9] 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.

[10] Jordan Hoffmann i in. *Training Compute-Optimal Large Language Models*. 2022. arXiv: 2203.15556 [cs.CL]. URL: https://arxiv.org/abs/2203.15556.

[11] Albert Q. Jiang i in. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL]. URL: https://arxiv.org/abs/2310.06825.

[12] Jared Kaplan i in. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361.

[13] Nikita Kitaev, Lukasz Kaiser i Anselm Levskaya. "Reformer: The Efficient Transformer". W: *International Conference on Learning Representations*. 2020.

[14] Aixin Liu i in. "Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model". W: *arXiv preprint arXiv:2405.04434* (2024).

[15] Aixin Liu i in. "Deepseek-v3. 2: Pushing the frontier of open large language models". W: *arXiv preprint arXiv:2512.02556* (2025).

[16] Yinhan Liu i in. "Roberta: A robustly optimized bert pretraining approach". W: *arXiv preprint arXiv:1907.11692* (2019).

[17] Zhen Qin i in. "cosformer: Rethinking softmax in attention". W: *arXiv preprint arXiv:2202.08791* (2022).

[18] Jay Shah i in. "FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision". W: *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. 2024. URL: https://openreview.net/forum?id=tVConYid20.

[19] 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.

[20] Gemma Team i in. *Gemma 2: Improving Open Language Models at a Practical Size*. 2024. arXiv: 2408.00118 [cs.CL]. URL: https://arxiv.org/abs/2408.00118.

[21] Iulia Turc i in. "Well-read students learn better: On the importance of pre-training compact models". W: *arXiv preprint arXiv:1908.08962* (2019).

[22] Gorka Urbizu i in. "Scaling Laws for BERT in Low-Resource Settings". W: *Findings of the Association for Computational Linguistics: ACL 2023*. Red. Anna Rogers, Jordan Boyd-Graber i Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, lip. 2023, s. 7771–7789. DOI: 10.18653/v1/2023.findings-acl.492. URL: https://aclanthology.org/2023.findings-acl.492/.

[23] Ashish Vaswani i in. "Attention is all you need". W: *Advances in neural information processing systems* 30 (2017).

[24] Sinong Wang i in. “Linformer: Self-attention with linear complexity”. W: *arXiv preprint arXiv:2006.04768* (2020).

[25] Manzil Zaheer i in. “Big bird: Transformers for longer sequences”. W: *Advances in neural information processing systems* 33 (2020), s. 17283–17297.

Spis rysunków

Spis tabel

Spis załączników

Załącznik

A. Szczegółowy opis parametrów

Poniższe tabele zawierają pełny opis parametrów modeli oraz eksperymentów możliwych do konfiguracji.

Tabela A.1: Parametry eksperymentu, logowania i tokenizacji

Sekcja	Parametr	Przykład	Opis
`experiment`	`name`	*run_v1*	Unikalna nazwa eksperymentu.
	`kind`	`finetuning`	Typ: `pretraining` lub `finetuning`.
	`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`	*run_v1*	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 A.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: `learned`, `sinusoidal`, `rope`.
	`rope.rope_base`	`10000.0`	Podstawa częstotliwości θ dla RoPE.
	`rope.rope_scale`	`1.0`	Skalowanie częstotliwości RoPE.

Tabela A.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 Softmax).
`attention.mha`	`use_native_sdpa`	`true`	Czy użyć natywnej implementacji `scaled_dot_product_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 A.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 A.5: Parametry kontynuacji treningu

Sekcja	**Parametr**	**Przykład**	**Opis**
`pretrained_exp`	`name`	*pre_v1*	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`	*pre_v1*	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 A.6: Parametry głowic zadaniowych

Sekcja	Parametr	Przykład	Opis
`mlm_head`	`tie_mlm_weights`	`true`	Czy współdzielić wagi dekodera i osadzeń.
(Pretrening)	`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`	`num_labels`	`2`	Liczba klas wyjściowych.
(Dostrajanie)	`pooling`	`cls`	Agregacja: `cls`, `mean`, `max`, `min`.
	`pooler_type`	`bert`	Warstwa pośrednia: `bert`, `roberta`, `null`.
	`classifier_dropout`	`0.1`	Dropout przed klasyfikatorem.

Tabela A.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.

B. Szczegółowe wyniki klasyfikatora bazowego

Poniższe tabele przedstawiają szczegółowe wyniki klasyfikacji dla modelu bazowego TF-IDF + regresja logistyczna na zbiorach testowych.

Tabela B.1: Wyniki klasyfikatora TF-IDF + LR dla zbioru IMDb

Klasa	**Precision**	**Recall**	**F1-score**
negative	89.77	89.16	89.46
positive	89.23	89.84	89.54
Accuracy		89.50	
Macro	89.50	89.50	89.50

Tabela B.2: Wyniki klasyfikatora TF-IDF + LR dla zbioru arXiv

Klasa	**Precision**	**Recall**	**F1-score**
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
Accuracy		84.36	
Macro	83.68	83.72	83.62

Tabela B.3: Wyniki klasyfikatora TF-IDF + LR dla zbioru Hyperpartisan

Klasa	**Precision**	**Recall**	**F1-score**
not_hyperpartisan	48.60	13.88	21.59
hyperpartisan	49.77	85.32	62.87
Accuracy		49.60	
Macro	49.18	49.60	42.23

C. Stabilność uczenia

Tabela C.1: Stabilność wyników F1-macro [%] na zbiorze testowym dla modelu SDPA na zbiorze IMDb (3 uruchomienia z różnymi ziarnami losowości)

Konfiguracja	**Agregacja**	**Max F1**	**Min F1**	**Różnica [p.p.]**
f=0, d=0.1	CLS	93.58	93.46	0.12
f=0, d=0.1	Mean	93.88	93.80	0.08
f=0, d=0.2	CLS	93.52	93.44	0.08
f=0, d=0.2	Mean	93.86	93.72	0.14
f=1, d=0.1	CLS	93.68	93.54	0.14
f=1, d=0.1	Mean	94.12	93.80	0.32
f=1, d=0.2	CLS	93.70	93.52	0.18
f=1, d=0.2	Mean	93.98	93.80	0.18
f=2, d=0.1	CLS	93.66	93.44	0.22
f=2, d=0.1	Mean	93.86	93.64	0.22
f=2, d=0.2	CLS	93.64	93.46	0.18
f=2, d=0.2	Mean	93.88	93.62	0.26

D. Kod źródłowy

Kod źródłowy aplikacji wraz z notatnikiem pozwalającym na odtworzenie eksperymentów został dołączony osobno w pliku ZIP wraz z pracą dyplomową.