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

Dokumentacja projektu – część III

Bartłomiej Borycki, Michał Iwaniuk

27 listopada 2025

1 Podsumowanie

W ramach niniejszego etapu prac zrealizowano kompletną, w pełni modularną implementację architektury Transformera, stanowiącą fundament dla planowanych badań porównawczych. Głównym osiągnięciem jest opracowanie szkieletu modelu, który pozwala na swobodną wymianę kluczowych komponentów obliczeniowych tj. mechanizmów atencji (klasyczne MHA, aproksymacyjne LSH i FAVOR) - bez konieczności modyfikacji pozostałej części systemu.

Stworzono środowisko eksperymentalne, zapewniające pełną powtarzalność i konfigurowalność procesów treningowych. Zaimplementowana infrastruktura integruje techniki optymalizacji uczenia (m.in. mieszana precyzja, akumulacja gradientów) oraz rozbudowany system monitorowania metryk, co umożliwia efektywne przeprowadzenie i analizę zaplanowanych eksperymentów w zadaniach pretreningu oraz klasyfikacji tekstu.

2 Tokenizer

W projekcie wykorzystano algorytm tokenizacji WordPiece (standard w modelach z rodziny BERT). Aby uprościć procesy trenowania, przetwarzania danych, zaimplementowano klasę pomocniczą `WordPieceTokenizerWrapper`.

2.1 Wrapper tokenizera

Klasa `WordPieceTokenizerWrapper` stanowi nakładkę na bibliotekę `tokenizers` oraz `transformers` Hugging Face. Jej głównym celem jest abstrakcja operacji niskopoziomowych i dostarczenia API do przygotowywania danych dla modelu.

2.1.1 Inicjalizacja i trening

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 tokeny specjalne `[CLS]` na początku i `[SEP]` na końcu sekwencji, co jest wymagane przez architekturę BERT.
- 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.

2.1.2 Przetwarzanie danych (Encoding)

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

1. Tokenizację tekstu na podjednostki.
2. Obcięcie sekwencji do maksymalnej długości (`max_length`) lub dopełnienie (padding) tokenem `[PAD]` do tej długości.
3. Generowanie maski atencji (`attention_mask`), gdzie wartość `True` oznacza tokeny paddingu, które powinny być ignorowane przez mechanizm atencji (odwrotnie niż w implementacji Hugging Face).
4. Opcjonalne dołączenie etykiet.

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

2.1.3 Maskowanie dla MLM

Dla potrzeb uczenia nienadzorowanego (Masked Language Modeling), zaimplementowano metodę `mask_input_for_mlm`. Realizuje ona dynamiczne maskowanie tokenów zgodnie ze strategią opisaną w oryginalnej pracy BERT:

- 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 (w celu zachowania spójności reprezentacji, (omyślnie 10%).

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

3 Architektura transformera

3.1 Klasa transformera

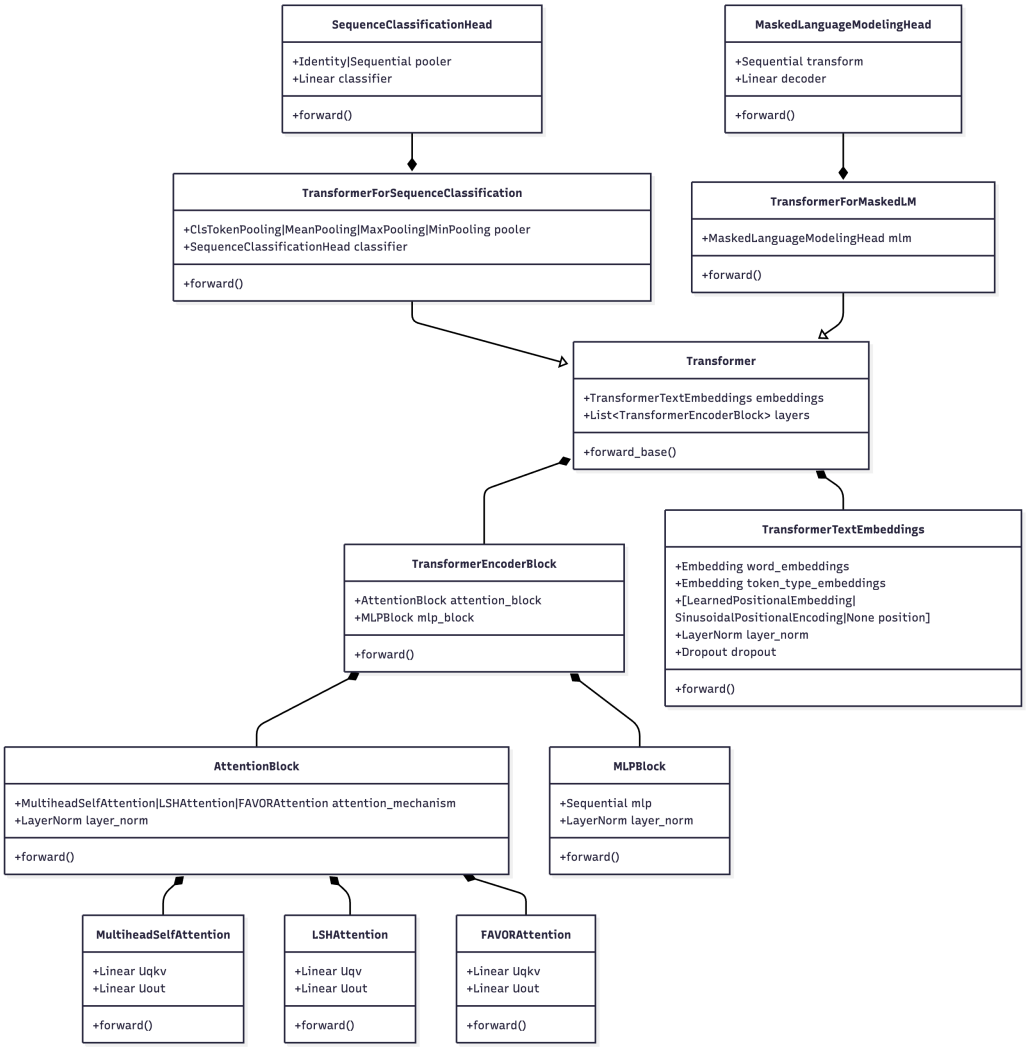
Implementacja architektury opiera się na modułowej hierarchii klas, której fundamentem jest klasa bazowa `Transformer`, a jej specjalizacje (`TransformerForSequenceClassification` oraz `TransformerForMaskedLM`) dostosowują model do konkretnych zadań uczenia maszynowego.

3.1.1 Model bazowy: Transformer

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

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

Kluczowym elementem implementacji jest metoda `forward_base`. Realizuje ona 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, a N to długość sekwencji. Klasa Transformer zarządza również dynamicznym obliczaniem i buforowaniem wartości trygonometrycznych dla kodowania pozycyjnego RoPE (Rotary Positional Embeddings), jeśli zostało ono wybrane w konfiguracji (funkcja `build_rope_cache`).



Rysunek 1: Diagram klas dla Architektury Transformera

3.1.2 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 poolingu.
3. Rzutowanie wektora na przestrzeń etykiet za pomocą głowicy klasyfikacyjnej.

Model zwraca słownik zawierający zarówno pełną sekwencję wyjściową, wektor po poolingu, jak i logity klasyfikacji $(B, \texttt{num\_labels})$.

3.1.3 Model Masked Language Modeling (MLM)

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

Implementacja obsługuje parametr `tie_mlm_weights`, ustawiony na `True`, wagi warstwy wyjściowej (dekodującej) są współdzielone z wagami macierzy osadzeń wejściowych, co jest standardową praktyką w modelach typu BERT zmniejszającą liczbę parametrów i zapobiegającą przeuczeniu.

3.2 Mechanizmy atencji

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

3.2.1 Abstrakcja bloku atencji

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

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

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

3.2.2 Standardowa atencja wielogłowicowa (MHA)

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

1. Projekcja wejścia na macierze zapytań (Q), kluczy (K) i wartości (V).
2. Podział na H niezależnych głowic o wymiarze $d_k = D/H$.
3. Opcjonalne zaaplikowanie rotacyjnego kodowania pozycyjnego (RoPE) na tensory Q i K.
4. Obliczenie macierzy atencji:

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

   gdzie M to addytywna maska (wartości $-\infty$ dla tokenów paddingu).
5. Scalenie wyników ze wszystkich głowic i projekcja liniowa wyjścia.

3.2.3 Atencja liniowa FAVOR+ (Performer)

Klasa `FAVORAttention` implementuje mechanizm atencji o złożoności czasowej oraz pamięciowej $O(N)$ wykorzystujący estymację jądra (Kernel Trick). Zamiast obliczać pełną macierz atencji $N \times N$, model aproksymuje funkcję softmax za pomocą mapowania cech $\phi(\cdot)$.

Kluczowe elementy implementacji:

- **Ortogonalne cechy losowe (GORF):** Metoda `_gaussian_orthogonal_random_matrix` generuje bloki ortogonalnych wektorów losowych poprzez dekompozycję QR macierzy losowej, co zapewnia lepsze pokrycie przestrzeni cech przy mniejszej liczbie próbek (M).
- **Mapowanie cech (ϕ):** Metoda `_phi` dysponuje wariantem transformacji `phi_kind`:
  - `"exp"` (metoda `_phi_exp`): Realizuje pozytywne cechy losowe aproksymujące jądro softmax. Wykorzystuje bufor `_omega` przechowujący macierz projekcji losowych
- **Liczenie uwagi:** Obliczane w metodzie `forward`:

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

  gdzie $Q' = \phi(Q)$, $K' = \phi(K)$, D to czynnik normalizacyjny obliczany jako $D = Q'(K'^T \mathbf{1}_N)$.
- **Zarządzanie cechami losowymi:** Implementacja wspiera 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.

3.2.4 Atencja LSH (Reformer)

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

Charakterystyka implementacji:

- **Współdzielone Q i K:** Zgodnie z architekturą Reformera, projekcje zapytań i kluczy są tożsame ($Q = K$), co jest wymogiem poprawnego działania LSH w tym kontekście.
- **Haszowanie i sortowanie:** Wykorzystano losowe rotacje do przypisania tokenów do kubełków (buckets). Następnie sekwencja jest sortowana według indeksów kubełków.
- **Przetwarzanie w blokach (Chunking):** Posortowana sekwencja jest dzielona na bloki o stałej długości (`chunk_size`). Atencja jest obliczana lokalnie, przy czym każdy blok ma dostęp do kontekstu bloku poprzedniego i następnego.
- **Maskowanie:** Zaimplementowano możliwość wyboru czy tokeny mają zwracać uwagę na tokeny z tego samego bloku ale innego kubełka (`mask_within_chunks`).

3.3 Kodowanie pozycyjne

Implementacja w klasie `TransformerTextEmbeddings` wspiera trzy podejścia do tego problemu, sterowane parametrem konfiguracyjnym `pos_encoding`.

3.3.1 TransformerTextEmbeddings

Klasa ta pełni rolę agregatora, łącząc:

- **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), dodawane addytywnie.
- **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`).

3.3.2 Kodowanie sinusoidalne (Sinusoidal)

Klasa `SinusoidalPositionalEncoding` implementuje deterministyczny schemat kodowania absolutnego, zgodny z pierwotną architekturą Transformera. 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:

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

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

3.3.3 Wyuczone kodowanie absolutne (Learned)

Klasa `LearnedPositionalEmbedding` realizuje podejście charakterystyczne dla modeli z rodziny BERT. Pozycje modelowane są jako wyuczalne wektory wagi macierzy o wymiarach (N_{max}, D). Każdemu indeksowi pozycji przyporządkowany jest unikalny wektor, który jest optymalizowany w procesie uczenia wstecznego.

3.3.4 Rotacyjne kodowanie pozycyjne (RoPE)

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

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

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

$$\theta_i = 10000^{-\frac{2i}{D}}, \qquad i = 0, 1, \ldots, \frac{D}{2} - 1.$$

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

$$\cos(m\theta_i), \qquad \sin(m\theta_i).$$

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

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

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

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

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

3.4 Pooling

Celem warstwy poolingu jest redukcja wymiarowości sekwencji stanów ukrytych $H \in R^{B\times N\times D}$ (zwracanej przez koder) do pojedynczego wektora reprezentacji całego tekstu $h_{pooled} \in R^{B\times D}$. Jest to operacja niezbędna w zadaniach klasyfikacji sekwencji. Zaimplementowano cztery strategie agregacji:

3.4.1 Pooling tokenu CLS (ClsTokenPooling)

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

$$h_{pooled} = H_{:,0,:}$$

3.4.2 Pooling uśredniający (MeanPooling)

Strategia polegająca na obliczeniu średniej arytmetycznej z wektorów wszystkich tokenów w sekwencji. Kluczowym elementem implementacji jest obsługa maski paddingu (M), aby tokeny wypełniające (PAD) nie wpływały na wartość średniej. Dla każdego przykładu w wsadzie:

$$h_{pooled} = \frac{\sum_{i=1}^{N}(H_{:,i,:} \cdot 1_{M_i=1})}{\max(1, \sum_{i=1}^{N} 1_{M_i=1})}$$

Dzielnik (mianownik) jest ograniczany od dołu wartością 1 (`clamp`), aby uniknąć dzielenia przez zero w przypadku całkowicie puste sekwencji (choć w praktyce rzadkiej).

3.4.3 Pooling maksymalny i minimalny (MaxPooling, MinPooling)

Strategie wybierające odpowiednio największą lub najmniejszą wartość cechy wzdłuż wymiaru sekwencji. W celu poprawnej obsługi paddingu, przed wykonaniem redukcji, pozycje zamaskowane są zastępowane wartościami "wartownikami":

- Dla **Max Pooling**: pozycje PAD wypełniane są wartością najmniejszą reprezentowalną dla danego typu danych (np. $-\infty$).
- Dla **Min Pooling**: pozycje PAD wypełniane są wartością największą (np. $+\infty$).

Gwarantuje to, że tokeny techniczne nie zostaną wybrane jako ekstrema.

3.5 Głowice zadaniowe

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

3.5.1 Głowica do klasyfikacji sekwencji (SequenceClassificationHead)

Moduł ten przyjmuje na wejściu wektor po poolingu (B, D) i rzutuje go na przestrzeń etykiet $(B, \text{num\_labels})$. Implementacja wspiera różne architektury "poolera"(warstwy pośredniej), sterowane parametrem `pooler_type`:

- **Styl BERT:** Składa się z warstwy gęstej zachowującej wymiarowość, funkcji aktywacji Tanh, a następnie warstwy wyjściowej.
$$y = \text{Linear}_{out}(\text{Dropout}(\text{Tanh}(\text{Linear}_{in}(x))))$$
- **Styl RoBERTa:** Charakteryzuje się dodatkowym Dropoutem na wejściu oraz inną kolejnością operacji.
$$y = \text{Linear}_{out}(\text{Dropout}(\text{Tanh}(\text{Linear}_{in}(\text{Dropout}(x)))))$$
- **Brak (None):** Bezpośrednie rzutowanie wejścia na wyjście (z uwzględnieniem Dropoutu).

3.5.2 Głowica modelowania języka (MaskedLanguageModelingHead)

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

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

4 Środowisko treningowe

Środowisko treningowe zostało zaprojektowane z myślą o powtarzalności badań i minimalizacji błędów konfiguracyjnych. System opiera się na plikach konfiguracyjnych w formacie YAML oraz dedykowanych skryptach generujących, które automatyzują tworzenie struktury katalogów i inicjalizację parametrów.

4.1 Konfiguracja eksperymentów

Zarządzanie eksperymentami odbywa się poprzez dedykowane skrypty pomocnicze, które automatyzują tworzenie spójnej struktury katalogów (`experiments/pretraining` oraz `experiments/finetuning`) i inicjalizację plików konfiguracyjnych. Każde uruchomienie jest w pełni determinowane przez plik `config.yaml` znajdujący się w katalogu danego eksperymentu.

4.1.1 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 przestrzeni `experiments/pretraining`, a następnie tworzy dedykowany katalog.
2. **Templating i wznawianie:**
   - W trybie standardowym: wczytywany jest szablon bazowy z `config_templates/pretraining.yaml`.
   - W trybie wznawiania (flaga `-rp`): konfiguracja jest kopiowana z istniejącego eksperymentu, a sekcja `training.resume` jest automatycznie uzupełniana o ścieżkę do ostatniego checkpointu (`model.ckpt`) oraz flagę `is_resume=True`.
3. **Konfiguracja ścieżek:** Automatyczne wyznaczenie ścieżki wyjściowej (`output_dir`) relatywnej do korzenia projektu (ROOT), co zapewnia przenośność środowiska między różnymi maszynami.
4. **Integracja z W&B:** Automatyczne przypisanie nazwy uruchomienia (`run_name`) dla systemu logowania WandB.

4.1.2 Inicjalizacja finetuningu i dziedziczenie

Skrypt `generate_finetuning_experiment.py` realizuje logikę niezbędną do przeprowadzenia douczania modelu na zadaniu docelowym. Kluczowym mechanizmem jest tutaj **dziedziczenie konfiguracji architektonicznej**.

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

- **Weryfikację źródła:** Sprawdzenie istnienia katalogu i pliku konfiguracyjnego eksperymentu bazowego.
- **Kopiowanie architektury:** Sekcje `architecture` oraz `tokenizer` są kopiowane bezpośrednio z konfiguracji pretreningu do konfiguracji finetuningu. Gwarantuje to, że model docelowy będzie miał identyczne wymiary warstw, liczbę głowic oraz słownik jak model bazowy, co jest warunkiem koniecznym poprawnego załadowania wag.
- **Relatywizację ścieżek:** Ścieżka do eksperymentu bazowego jest zapisywana w sekcji `pretrained_experiment.path` jako ścieżka względna względem korzenia projektu. Umożliwia to skryptowi treningowemu jednoznaczne zlokalizowanie checkpointu pretreningowego niezależnie od środowiska uruchomieniowego.

Dzięki temu podejściu użytkownik nie musi ręcznie przepisywać hiperparametrów architektury, co eliminuje ryzyko pomyłki przy definiowaniu modelu pochodnego.

4.2 Skrypt treningowy

System treningowy został zaprojektowany w architekturze składającej się ze skryptu treningowego (punkt wejścia) oraz klasy `TrainingLoop`, która zawiera właściwą logikę optymalizacji modelu. Taka separacja zapewnia przejrzystość kodu i łatwość adaptacji do różnych zadań (pretrening vs finetuning).

4.2.1 Punkt wejścia i inicjalizacja środowiska

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

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

4.2.2 Logika pętli treningowej (TrainingLoop)

Zainicjowany model trafia do obiektu `TrainingLoop`, który zarządza cyklem życia procesu uczenia. Implementacja ta integruje szereg nowoczesnych technik optymalizacyjnych:

- **Mieszana precyzja (AMP):** Wykorzystanie `torch.amp.GradScaler` pozwala na wykonywanie części obliczeń w precyzji FP16/BF16, co przyspiesza trening i redukuje zużycie pamięci VRAM, przy zachowaniu stabilności numerycznej.
- **Akumulacja gradientów:** Parametr `grad_accum_steps` umożliwia odseparowanie logicznego rozmiaru wsadu od fizycznych ograniczeń pamięci GPU poprzez sumowanie gradientów z wielu kroków przed wykonaniem aktualizacji wag optymalizatorem.
- **Stabilizacja (Clipping):** Zastosowano przycinanie normy gradientów (`clip_grad_norm_`) w celu zabezpieczenia przed zjawiskiem eksplodujących gradientów.
- **Harmonogram uczenia (Scheduler):** Zaimplementowano harmonogram typu *cosine decay* z fazą liniowej rozgrzewki (warmup).

Przepływ danych w pojedynczym kroku treningowym (`_train_step`) obejmuje przygotowanie wsadu, przejście w przód w kontekście `autocast`, obliczenie straty, skalowanie gradientów oraz warunkową aktualizację wag (w zależności od kroku akumulacji).

4.2.3 Dynamiczna optymalizacja wsadów

W celu zwiększenia wydajności przetwarzania sekwencji o zróżnicowanej długości, zaimplementowano niestandardową funkcję kolacjonującą `make_collate_trim_to_longest`. W przeciwieństwie do statycznego dopełniania (padding) do maksymalnej długości modelu (np. 512), funkcja ta analizuje każdą paczkę danych (batch) i przycina tensory wejściowe do długości najdłuższego rzeczywistego przykładu w danej paczce. Pozwala to na ograniczenie zbędnych obliczeń na tokenach `[PAD]`, co jest szczególnie istotne przy wykorzystaniu mechanizmów atencji o złożoności kwadratowej $O(N^2)$.

4.2.4 Ewaluacja i metryki

System ewaluacji automatycznie dostosowuje obliczane metryki do rodzaju zadania:

- **Dla MLM:** Podstawową metryką jest perpleksja (perplexity), wyliczana jako e^{loss}.
- **Dla klasyfikacji:** Wykorzystano bibliotekę `scikit-learn` do obliczania szerokiego spektrum metryk, w tym: *Accuracy*, *Balanced Accuracy*, *F1 Score* (warianty macro i micro). Dodatkowo monitorowana jest entropia predykcji, służąca jako miara pewności modelu.

4.2.5 Zarządzanie stanem (Checkpointing)

Skrypt obsługuje zaawansowane 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, scheduler, skaler) do pliku `best-model.ckpt`.
- **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.

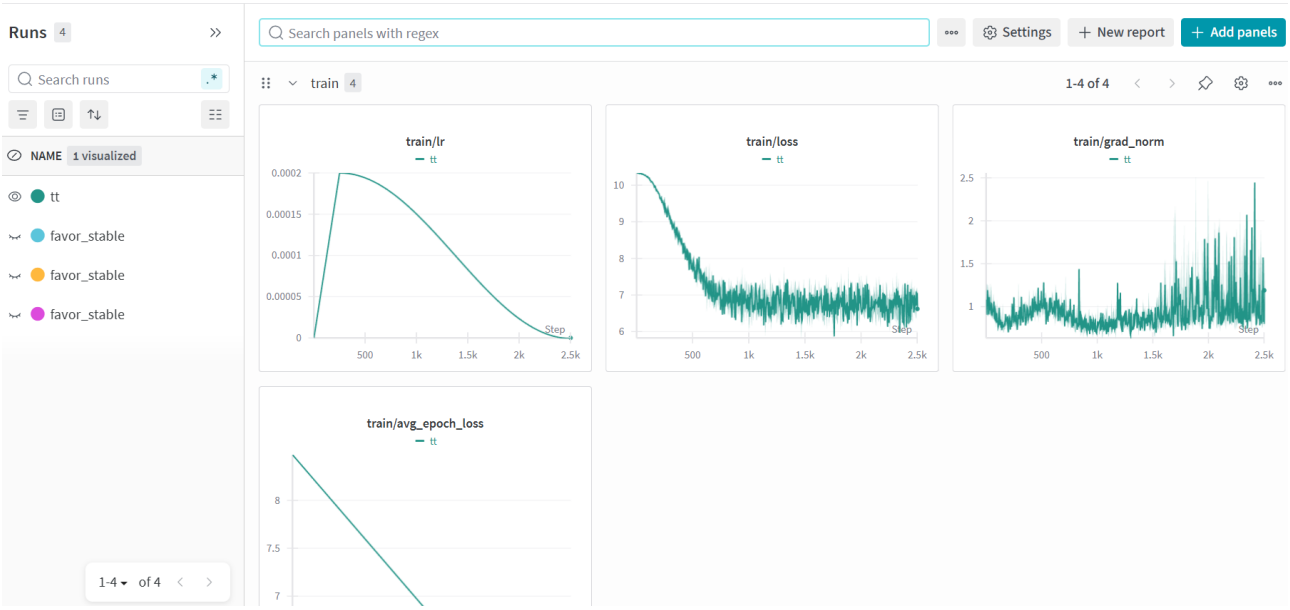
4.3 Logowanie przebiegu treningu

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

4.3.1 Integracja z Weights & Biases

Głównym kanałem zbierania metryk jest serwis W&B. W ramach projektu utworzono dedykowany zespół `praca-inzynierska`, gdzie agregowane są wyniki wszystkich eksperymentów. Klasa `WandbRun` odpowiada za:

- **Inicjalizację sesji:** Metoda `__init__` nawiązuje połączenie z projektem określonym w konfiguracji, przesyłając jednocześnie pełny słownik hiperparametrów (`config`).
- **Organizacja metryk:** Metody `log_train` oraz `log_eval` automatycznie dodają odpowiednie prefiksy (`train/`, `eval/`, `test/`) do nazw zmiennych. Pozwala to na przejrzyste grupowanie wykresów.



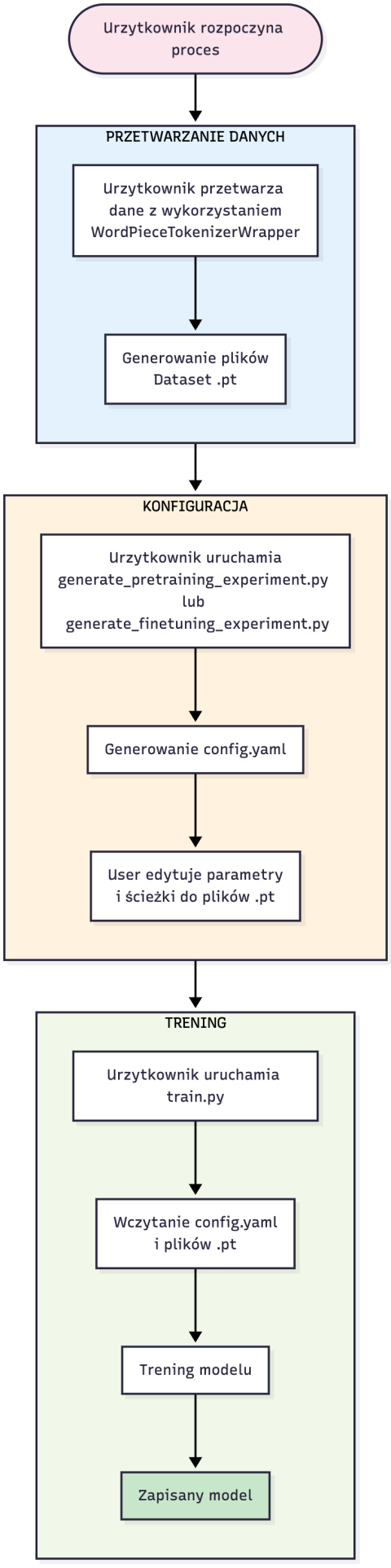
Rysunek 2: Przykładowe metryki treningu zalogowane w Weights and Biases

4.3.2 Lokalny zapis danych (CSV)

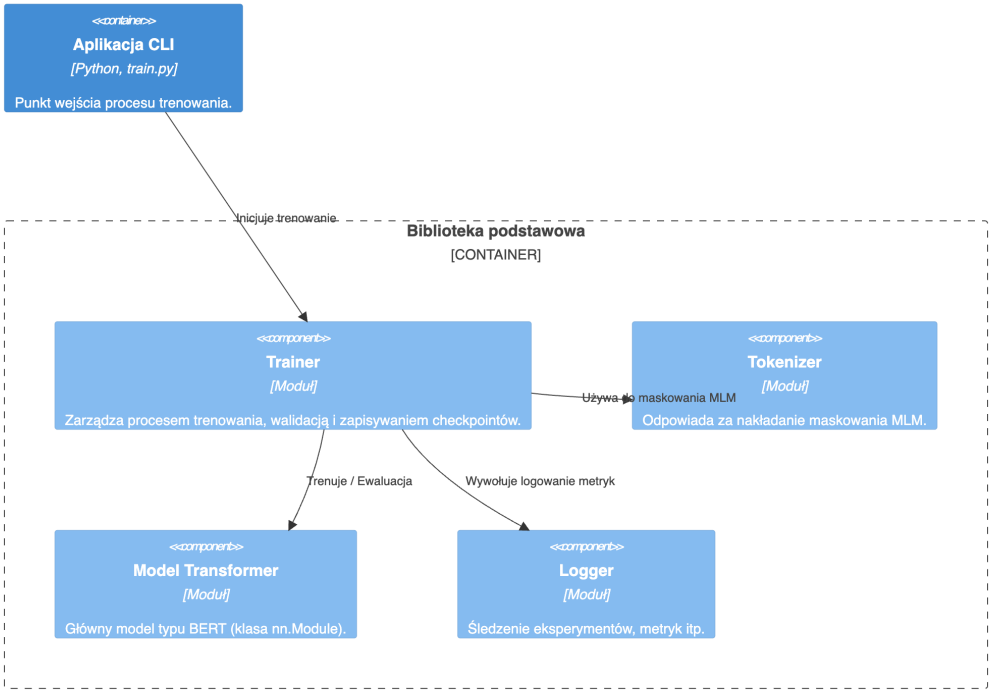
W przypadku ustawienia w konfiguracji `log_metrics_csv = True` logger utrzymuje lokalną kopię wszystkich metryk. Dane są zapisywane w plikach:

- `metrics/train/metrics.csv` – dla kroków treningowych.
- `metrics/eval/metrics.csv` – dla walidacji i testów.

Metoda pomocnicza `_write_csv` zarządza otwieraniem plików w trybie dopisywania (*append*) oraz tworzeniem nagłówków kolumn przy pierwszym zapisie. Mechanizm ten gwarantuje dostęp do surowych danych numerycznych nawet w przypadku awarii serwisu zewnętrznego lub konieczności wykonania analiz offline.



Rysunek 3: Diagram przepływu dla procesu treningu



Rysunek 4: Diagram komponentów systemu

5 Parametry modeli i treningu

Konfiguracja eksperymentów odbywa się za pomocą plików w formacie YAML, które definiują pełny stan hiperparametrów modelu, środowiska treningowego oraz przetwarzania danych. Poniższe tabele prezentują szczegółowy opis wszystkich dostępnych parametrów konfiguracyjnych, podzielonych na sekcje funkcjonalne, wraz z przykładowymi wartościami domyślnymi.

Tabela 1: Parametry eksperymentu, logowania i tokenizacji

Sekcja	**Parametr**	**Przykładowa wartość**	**Opis**
`experiment`	`name`	*run_v1*	Unikalna nazwa eksperymentu (ID w W&B).
	`kind`	`pretraining`	Typ zadania: `pretraining` lub `finetuning`.
	`output_dir`	`experiments/...`	Ścieżka katalogu wyjściowego.
	`seed`	420	Ziarno losowości (Python, NumPy, PyTorch).
`logging`	`use_wandb`	`true`	Logowanie do Weights & Biases.
	`log_eval_metrics`	`true`	Czy logować metryki walidacyjne.
	`log_metrics_csv`	`false`	Równoległy zapis metryk do CSV.
	`csv_..._path`	`metrics/train/...`	Ścieżki do plików CSV (trening/eval).
`tokenizer`	`max_length`	512	Maksymalna długość sekwencji (N).
	`vocab_dir`	`.../BERT_original`	Katalog ze słownikiem tokenizera.

Tabela 2: Hiperparametry architektury Transformera

Sekcja	Parametr	Przykład	Opis
`architecture`	`embedding_dim`	`32`	Wymiar osadzeń i stanów ukrytych (D).
	`num_layers`	`4`	Liczba bloków kodera.
	`mlp_size`	`128`	Rozmiar warstwy ukrytej w MLP.
	`pos_encoding`	`rope`	Typ pozycji: `learned`, `sinusoidal`, `rope`.
	`rope_base`	`10000.0`	Podstawa częstotliwości θ dla RoPE.
	`rope_scale`	`1.0`	Skalowanie częstotliwości RoPE.
`dropout`	`mlp_dropout`	`0.1`	Dropout w bloku MLP.
	`embedding_dropout`	`0.1`	Dropout na sumie osadzeń wejściowych.

Tabela 3: Parametry mechanizmu atencji

Wariant	Parametr	Przykład	Opis
Wspólne	`kind`	`lsh`	Typ: `mha`, `lsh`, `favor`.
	`num_heads`	`4`	Liczba głowic atencji (H).
	`projection_bias`	`true`	Bias w projekcjach Q/K/V/Out.
	`attn_out_dropout`	`0.1`	Dropout na wyjściu bloku atencji.
	`attn_dropout`	`0.0`	Dropout na macierzy atencji (Softmax).
	`attention_emb...`	`null`	Opcjonalny wymiar projekcji atencji.
LSH	`num_hashes`	`4`	Liczba rund haszowania.
	`chunk_size`	`64`	Rozmiar bloku lokalnej atencji.
	`mask_within_...`	`true`	Maskowanie między kubełkami w bloku.
Favor	`nb_features`	`256`	Liczba cech losowych (M).
	`ortho_features`	`true`	Użycie ortogonalnych cech (GORF).
	`phi`	`exp`	Funkcja jądra: `exp`, `elu`, `relu2`.
	`stabilize`	`true`	Stabilizacja numeryczna (odejmowanie max).
	`eps`	`1e-6`	Epsilon dla mianownika.

Tabela 4: Hiperparametry procesu treningowego

Sekcja	Parametr	Przykład	Opis
`training`	`batch_size`	`6`	Rozmiar wsadu.
	`epochs`	`3`	Liczba epok treningowych.
	`learning_rate`	`2e-5`	Początkowy współczynnik uczenia.
	`warmup_ratio`	`0.1`	Procent kroków rozgrzewki.
	`weight_decay`	`0.01`	Współczynnik zaniku wag (L2).
	`grad_accum_steps`	`1`	Kroki akumulacji gradientu.
	`max_grad_norm`	`1.0`	Próg przycinania gradientów.
	`use_amp`	`true`	Mieszana precyzja (Automatic Mixed Precision).
	`loss`	`cross_entropy`	Funkcja kosztu.
	`device`	`auto`	Urządzenie: `auto`, `cuda`, `cpu`.
`resume`	`is_resume`	`false`	Czy wznawiać trening (tylko pretrening).
(Opcjonalne)	`strict`	`true`	Czy wymagać pełnej zgodności wag.

Tabela 5: Parametry głowic zadaniowych

Typ głowicy	**Parametr**	**Przykład**	**Opis**
MLM Head (Pretrening)	`tie_mlm_weights`	`true`	Współdzielenie wag dekodera i osadzeń.
	`mask_p`	`0.15`	Prawdopodobieństwo wyboru tokenu.
	`mask_token_p`	`0.8`	Szansa na zastąpienie przez `[MASK]`.
	`random_token_p`	`0.1`	Szansa na zastąpienie losowym słowem.
Class. Head (Finetuning)	`num_labels`	`2`	Liczba klas wyjściowych.
	`pooling`	`cls`	Agregacja: `cls`, `mean`, `max`, `min`.
	`pooler_type`	`bert`	Warstwa pośrednia: `bert`, `roberta`, `null`.
	`classifier_dropout`	`0.1`	Dropout przed klasyfikatorem.

6 Środowisko programistyczne

Eksperymenty przeprowadzono w wyizolowanym środowisku wirtualnym. Poniżej przedstawiono wersje interpretera Python oraz kluczowych bibliotek wykorzystanych w implementacji projektu:

- **Python:** 3.12.2
- **PyTorch:** 2.8.0
- **Transformers:** 4.56.2
- **Tokenizers:** 0.22.1
- **WandB:** 0.22.1
- **Scikit-learn:** 1.6.1
- **NumPy:** 2.1.3
- **Pandas:** 2.2.3
- **PyYAML:** 6.0.2