Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Дослідження повного циклу розробки генеративних мовних моделей: від підготовки даних до оптимізації інференсу
(тема)

Виконав:
здобувач четвертого року навчання,
групи ІТШІ-22-3

Максим АРГУНОВ
(власне ім'я, прізвище)

Спеціальність 122 Комп'ютерні науки

(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Штучний інтелект

(повна назва освітньої програми)

Керівник ст. викл. Олександр СТЬОПІН
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ _____ Лариса ЧАЛА
(підпис) (власне ім'я, прізвище)

2026 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Штучного інтелекту

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Штучний інтелект
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Аргунову Максиму Костянтиновичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження повного циклу розробки генеративних мовних моделей: від підготовки даних до оптимізації інференсу

затверджена наказом університету від 5 червня 20 26 р. № 488Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 14 липня 20 26 р.

3. Вихідні дані до роботи науково-технічні публікації; відкритий веб-корпус OSCAR-2301 (англійська та німецька підмножини); бібліотеки PyTorch, Hugging Face (datasets, tokenizers, Transformers), NLTK, Gemini API для генерації синтетичного датасету

4. Перелік питань, що потрібно опрацювати в роботі

1) Аналіз предметної галузі

2) Підготовка даних

3) Архітектура моделі

4) Навчання та експерименти

5) Оцінка якості та аналіз результатів

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / термін виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	08.06.2026	виконано
2	Аналіз предметної області	12.06.2026	виконано
3	Постановка задачі	15.06.2026	виконано
4	Практичні експерименти	16.06.2026	виконано
5	Виконання практичної частини	17.06.2026	виконано
6	Оформлення звіту	01.07.2026	виконано
7	Перевірка на академічний плагіат	03.07.2026	виконано
8	Нормоконтроль	06.07.2026	виконано
9	Підготовка презентації та доповіді	08.07.2026	виконано
10	Попередній захист	10.07.2026	виконано
11	Рецензування	10.07.2026	виконано
12	Захист перед ЕК	14.07.2026	

Дата видачі завдання 8 червня 2026 р.

Здобувач _____
(підпис)

Керівник роботи _____ ст. викл. Олександр СТЬОПІН
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 76 с., 1 рис., 5 табл., 3 дод., 26 джерел.

ВЕБ-КОРПУС, МАШИННИЙ ПЕРЕКЛАД, МОВНА МОДЕЛЬ, ПЕРЕДТРЕНУВАННЯ, ФІЛЬТРАЦІЯ ДАНИХ, ЯКІСТЬ ДАНИХ, TRANSFORMER.

Об’єкт дослідження – процес побудови малої мовної моделі на базі архітектури Transformer.

Предмет дослідження – методи підготовки даних, архітектурні рішення та стратегії навчання, що впливають на якість генералізації моделі.

Мета роботи – демонстрація повного шляху побудови мовної моделі від підготовки та фільтрації даних до оцінки на прикладній задачі машинного перекладу (DE→EN).

Методи дослідження – експериментальний метод (побудова та навчання моделі з різними конфігураціями даних), порівняльний аналіз (зіставлення результатів моделей, навчених на фільтрованих та нефільтрованих даних).

Розроблено триетапний конвеєр фільтрації веб-корпусу OSCAR, що відсіює приблизно 90% документів. Реалізовано кастомну encoder-decoder Transformer-архітектуру з приблизно 100M параметрами із сучасними модифікаціями (GQA, RoPE, SwiGLU, Pre-LN). Проведено передтренування та дотренування на задачі машинного перекладу. Емпірично підтверджено, що модель, передтренована на 250M фільтрованих токенів, перевершує модель на 384M нефільтрованих токенів за якістю генералізації, що еквівалентно приблизно 35% економії обчислювальних витрат.

ABSTRACT

Bachelor's thesis contains: 76 pp., 1 fig., 5 tabl., 3 ann., 26 references.

DATA FILTERING, DATA QUALITY, LANGUAGE MODEL, MACHINE TRANSLATION, PRETRAINING, TRANSFORMER, WEB CORPUS.

Object of the study – the process of building a small language model based on the Transformer architecture.

Subject of the study – data preparation methods, architectural decisions, and training strategies affecting the model's generalization quality.

Purpose of the work – to demonstrate the complete path of building a language model, from data preparation and filtering to evaluation on an applied machine translation task (DE→EN).

Research methods – experimental method (building and training the model with different data configurations), comparative analysis (comparing the results of models trained on filtered and unfiltered data).

A three-stage filtering pipeline for the OSCAR web corpus has been developed, discarding approximately 90% of documents. A custom encoder-decoder Transformer architecture with approximately 100M parameters, incorporating modern modifications (GQA, RoPE, SwiGLU, Pre-LN), has been implemented. Pretraining and fine-tuning on the machine translation task have been performed. It has been empirically confirmed that a model pretrained on 250M filtered tokens outperforms a model trained on 384M unfiltered tokens in terms of generalization quality, equivalent to approximately 35% savings in computational cost.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 8
Вступ 9
1 Аналіз предметної галузі 11
1.1 Мовні моделі та їх роль у NLP 11
1.2 Архітектура Transformer 12
1.3 Сучасні модифікації Transformer 14
1.3.1 Pre-Layer Normalization. 14
1.3.2 Grouped Query Attention (GQA) 15
1.3.3 Rotary Positional Embeddings (RoPE) 15
1.3.4 SwiGLU 16
1.3.5 KV Cache 16
1.4 Стратегії навчання мовних моделей 17
1.5 Проблема якості даних для навчання мовних моделей 18
1.6 Висновки до першого розділу 20
2 Підготовка даних 21
2.1 Загальна стратегія та джерела даних 21
2.2 Навчання токенізатора 22
2.3 Багатоетапний конвеєр фільтрації 23
2.3.1 Перший рівень: регулярні вирази та структурні фільтри (49% відсіювання) 23
2.3.2 Другий рівень: статистичні та лексичні евристики (10% відсіювання) 25
2.3.3 Третій рівень: токенайзерні евристики (31% відсіювання) 27
2.4 Генерація синтетичного паралельного датасету 28
2.5 Висновки до другого розділу 29
3 Архітектура моделі 31
3.1 Загальна структура моделі 31
3.2 Grouped Query Attention (GQA) 32

3.3 Rotary Positional Embeddings (RoPE) ... 32
3.4 SwiGLU Feed-Forward Network ... 34
3.5 Pre-Layer Normalization ... 35
3.6 Ініціалізація ваг ... 36
3.7 KV Cache для ефективної генерації ... 36
3.8 Стратегії семплування при генерації ... 37
3.9 Висновки до третього розділу ... 38
4 Навчання та експерименти ... 39
4.1 Конфігурація навчання ... 39
4.2 Передтренування (Pre-training) ... 41
4.3 Дотренування (Fine-tuning) ... 43
4.4 Висновки до четвертого розділу ... 44
5 Оцінка якості та аналіз результатів ... 45
5.1 Парадокс training loss: фільтровані vs нефільтровані дані ... 45
5.2 Fine-tuning: валідація на задачі машинного перекладу ... 46
5.3 Якісний аналіз перекладів ... 47
5.4 KV Cache: кількісна оцінка прискорення ... 49
5.5 Аналіз обчислювальної ефективності ... 50
5.6 Висновки до п’ятого розділу ... 51
Висновки ... 52
Перелік джерел посилання ... 54
Додаток А Лістинги вихідного коду ... 58
Додаток Б Приклади перекладів моделі ... 70
Додаток В Відомість кваліфікаційної роботи ... 76

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

cuDNN – CUDA Deep Neural Network Library;

FFN – Feed-Forward Network – повнозв'язна мережа прямого поширення;

GPU – Graphics Processing Unit – графічний процесор;

GQA – Grouped Query Attention – увага зі згрупованими запитами;

KV Cache – Key-Value Cache – кеш ключів та значень;

LLM – Large Language Model – велика мовна модель;

NLP – Natural Language Processing – обробка природної мови;

Pre-LN – Pre-Layer Normalization – нормалізація перед підшаром;

RoPE – Rotary Positional Embeddings – обертальне позиційне кодування;

SDPA – Scaled Dot-Product Attention – масштабована увага на основі скалярного добутку;

SNR – Signal-to-Noise Ratio – відношення сигнал-шум;

SwiGLU – Swish-Gated Linear Unit – лінійний блок із Swish-гейтуванням;

TTR – Type-Token Ratio – відношення типів до токенів;

VRAM – Video Random Access Memory – відеопам'ять.

ВСТУП

Останнє десятиліття ознаменувалося стрімким розвитком нейронних мовних моделей, які кардинально змінили ландшафт обробки природної мови (NLP). Від перших рекурентних архітектур до сучасних великих мовних моделей (LLM) на основі архітектури Transformer, прогрес у цій галузі був безпрецедентним. Проте за вражаючими результатами комерційних моделей часто залишається непомітним фундаментальний інженерний процес, що стоїть за їхнім створенням: збір та очищення даних, проєктування архітектури, організація навчання та оцінка результатів.

Актуальність роботи зумовлена кількома факторами. По-перше, великі мовні моделі з десятками та сотнями мільярдів параметрів залишаються «чорними скриньками» для більшості практиків: їхній повний цикл створення – від курації даних до фінальної оцінки – є непрозорим та недокументованим. Побудова малої мовної моделі з нуля дозволяє дослідити кожен етап цього процесу на доступному масштабі та сформувати фундаментальне розуміння принципів, що лежать в основі сучасних LLM. По-друге, сучасна індустрія часто зводить роботу з мовними моделями до дотренування або використання API готових рішень, ігноруючи фундаментальне розуміння кожного етапу створення моделі. По-третє, саме якість даних, а не розмір моделі чи обсяг обчислень, є визначальним фактором успіху – і ця теза потребує емпіричного підтвердження.

Метою роботи є демонстрація повного шляху побудови мовної моделі – від підготовки та фільтрації даних до оцінки на прикладній задачі машинного перекладу (DE→EN). Машинний переклад обрано як зручну задачу для валідації: він має чітку метрику оцінки, зрозумілий для інтерпретації результат та достатню складність для перевірки якості мовного моделювання.

Для досягнення поставленої мети було вирішено низку задач. Спочатку проведено аналіз сучасних підходів до побудови мовних моделей,

зокрема розглядено еволюцію від класичних n-грамних і рекурентних моделей до сучасних великих мовних моделей, а також досліджено ключові сучасні модифікації архітектури Transformer – Grouped Query Attention, Rotary Positional Embeddings, SwiGLU та Pre-Layer Normalization, які визначають ефективність і якість сучасних моделей. Далі було розроблено багатоетапний конвеєр фільтрації веб-корпусу OSCAR, що дозволило відсіяти неякісні, шаблонні та шумові документи і сформувати навчальний корпус з високим співвідношенням сигналу до шуму, придатний для якісного передтренування мовної моделі.

На основі відфільтрованого корпусу навчено доменний токенайзер, адаптований саме під білінгвальні (англо-німецькі) дані, що забезпечує ефективнішу сегментацію тексту порівняно з готовими універсальними токенайзерами. Оскільки існуючі відкриті паралельні корпуси (субтитри, парламентські промови, новини) не відповідали за стилем і лексикою корпусу передтренування, окремим завданням було генерація синтетичного паралельного датасету за допомогою великої мовної моделі, що дозволяє провести подальше дотренування без втрати набутої лінгвістичної різноманітності. Ключовим інженерним завданням була реалізація кастомної encoder-decoder архітектури Transformer, що інтегрує зазначені сучасні модифікації для підвищення обчислювальної ефективності та якості генерації. На отриманій архітектурі проведено передтренування з використанням задачі відновлення тексту за принципом T5-style span corruption, що дозволило моделі засвоїти базові мовні закономірності на великому обсязі неанотованих даних. Після цього виконано дотренування моделі на прикладній задачі машинного перекладу з німецької мови на англійську з використанням згенерованого синтетичного датасету. Насамкінець, отримані результати оцінено та проаналізовано, зокрема, порівнено вплив якості навчальних даних на здатність моделі до генералізації через зіставлення показників моделей, передтренованих на фільтрованому та нефільтрованому корпусах.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Мовні моделі та їх роль у NLP

Мовна модель (LM) – це ймовірнісна модель, що визначає розподіл ймовірностей над послідовностями слів або токенів у природній мові. Формально, для послідовності токенів $w_1, w_2, \dots, w_n$ мовна модель оцінює ймовірність $P(w_1, w_2, \dots, w_n)$, яку за ланцюговим правилом можна розкласти як:

$$P(w_1, \dots, w_n) = \prod_{i=1}^{n} P(w_i | w_1, \dots, w_{i-1}). \quad (1.1)$$

Еволюція мовних моделей пройшла кілька ключових етапів. Статистичні n-грамні моделі, що домінували у 1990–2000-х роках, оцінювали ймовірність наступного слова на основі фіксованого вікна з n-1 попередніх слів. Попри свою простоту, вони страждали від проблеми розрідженості даних (data sparsity) та нездатності моделювати довгострокові залежності.

Перший суттєвий прорив відбувся з появою нейронних мовних моделей, зокрема рекурентних нейронних мереж (Recurrent Neural Networks, RNN) та їхніх вдосконалених варіантів – LSTM (Long Short-Term Memory) та GRU (Gated Recurrent Unit). Ці архітектури здатні теоретично моделювати залежності довільної довжини, проте на практиці страждають від проблеми зникаючих градієнтів та послідовної обробки, що унеможливлює ефективну паралелізацію навчання.

Революційним став 2017 рік, коли Vaswani et al. представили архітектуру Transformer [1], яка повністю відмовилася від рекурентності на користь механізму self-attention. Ця архітектура забезпечила паралельну обробку послідовностей, ефективне моделювання довгострокових

залежностей та масштабованість, що відкрило шлях до створення моделей із мільярдами параметрів.

Сучасний ландшафт мовних моделей характеризується чітким поділом на два класи. LLM, такі як GPT, Claude, Gemini, містять десятки та сотні мільярдів параметрів, вимагають колосальних обчислювальних ресурсів та зазвичай доступні через API. На протилежному полюсі знаходяться малі мовні моделі (порядку 100M–70B параметрів), які можуть працювати на споживчому обладнанні та є перспективними для спеціалізованих задач, де важлива автономність, конфіденційність даних або обмежений обчислювальний бюджет.

У цій роботі машинний переклад використовується як прикладна задача для валідації якості побудованої мовної моделі. Переклад вимагає глибокого розуміння семантики обох мов, має чіткі метрики оцінки та дозволяє наочно продемонструвати здатність моделі до генералізації. Актуальність цього напряму підтверджують і вітчизняні дослідження керованості машинного перекладу на основі encoder-decoder архітектур [2].

1.2 Архітектура Transformer

Архітектура Transformer, запропонована у роботі «Attention Is All You Need» [1], стала фундаментом сучасного NLP. На відміну від рекурентних та конволюційних підходів, Transformer базується виключно на механізмі уваги (attention), що дозволяє моделі одночасно аналізувати всі позиції у вхідній послідовності.

Ключовим компонентом архітектури є механізм self-attention, який для кожного елемента послідовності обчислює зважену суму всіх інших елементів. Формально, для вхідної матриці X обчислюються три проєкції: запити $Q = XW_Q$, ключі $K = XW_K$ та значення $V = XW_V$. Функція уваги визначається як:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V, \quad (1.2)$$

де d_k – розмірність ключів.

Масштабування на $\sqrt{d_k}$ запобігає надмірному зростанню скалярного добутку, що могло б призвести до насичення функції softmax та зникнення градієнтів.

Multi-Head Attention (MHA) розширює цей механізм, паралельно обчислюючи кілька «голів» уваги з різними проєкціями. Кожна голова може фокусуватися на різних аспектах вхідних даних – синтаксичних, семантичних чи позиційних. Виходи всіх голів конкатенуються та проєктуються назад у простір моделі:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) \cdot W_O, \quad (1.3)$$

де $\text{head}_i = \text{Attention}(QW_{Q_i}, KW_{K_i}, VW_{V_i})$.

Оригінальний Transformer має encoder-decoder структуру. Encoder складається зі стеку ідентичних шарів, кожен з яких містить два підшари: multi-head self-attention та позиційно-незалежну повнозв'язну мережу (feed-forward network, FFN). Decoder додатково містить третій підшар – cross-attention, який дозволяє декодеру звертатися до виходів енкодера. Крім того, self-attention у декодері використовує маскування, що запобігає доступу до майбутніх позицій під час авторегресійної генерації.

Оригінальна архітектура використовує синусоїдальне позиційне кодування для надання моделі інформації про порядок елементів у послідовності, оскільки механізм самоуваги сам по собі є інваріантним до перестановок. Позиційні вектори додаються до вхідних ембедінгів перед подачею в перший шар.

Кожен підшар оточений залишковим з'єднанням та нормалізацією шару, що стабілізує навчання глибоких мереж.

1.3 Сучасні модифікації Transformer

З моменту публікації оригінальної архітектури Transformer у 2017 році дослідники запропонували численні вдосконалення, спрямовані на покращення стабільності навчання, обчислювальної та пам'яттєвої ефективності, а також якості генерації тексту. Ці модифікації виникли як відповідь на практичні труднощі масштабування глибоких моделей: нестабільність градієнтів при збільшенні кількості шарів, надмірні витрати пам'яті на кешування під час генерації, обмежену здатність до узагальнення на довші послідовності та неоптимальну структуру feed-forward блоків. У цій роботі використано чотири ключові архітектурні модифікації, кожна з яких вирішує окрему проблему оригінальної архітектури.

1.3.1 Pre-Layer Normalization.

У оригінальному Transformer нормалізація шару (LayerNorm) застосовується після залишкового з'єднання (Post-LN). Xiong et al. [3] продемонстрували, що переміщення LayerNorm перед кожним підшаром (Pre-LN) суттєво покращує стабільність навчання глибоких моделей. При Post-LN підході градієнти у нижніх шарах можуть ставати нестабільно великими, що ускладнює оптимізацію та вимагає обережного підбору learning rate та warmup. Pre-LN усуває цю проблему: вихід кожного підшару проходить через нормалізацію перед обчисленнями, а залишкове з'єднання додається після. Формально:

$$x' = x + \mathrm{SubLayer}(\mathrm{LayerNorm}(x)). \tag{1.4}$$

Замість оригінального:

$$x' = \mathrm{LayerNorm}(x + \mathrm{SubLayer}(x)). \tag{1.5}$$

1.3.2 Grouped Query Attention (GQA)

Стандартний Multi-Head Attention вимагає окремих key та value проєкцій для кожної голови уваги, що призводить до значних витрат пам'яті, особливо при кешуванні під час генерації. Ainslie et al. [4] запропонували GQA як компроміс між повним Multi-Head Attention та Multi-Query Attention (MQA, де всі голови поділяють одну пару K/V). У GQA кілька голів запитів спільно використовують одну пару key/value голів. У нашій реалізації 8 query heads розділені на 4 групи по 2, кожна з яких обслуговується однією парою KV heads (ratio 4:1). Це зменшує витрати пам'яті на KV cache приблизно у 4 рази порівняно з MHA при мінімальній втраті якості.

1.3.3 Rotary Positional Embeddings (RoPE)

Замість адитивного позиційного кодування, Su et al. [5] запропонували RoPE – метод кодування позиції через обертання векторів у комплексному просторі. Ключова ідея полягає в тому, що скалярний добуток між обернутими векторами запитів та ключів залежить лише від їхніх відносних позицій, а не абсолютних. Це дозволяє моделі природно узагальнювати на послідовності різної довжини. RoPE застосовує обертання до пар елементів вектора, де кут обертання визначається позицією токена та частотою, що залежить від індексу пари. Математично, для позиції m та пари елементів з індексом i:

$$f(x, m) = \begin{pmatrix} x_{2i}cos(m\theta_i) - x_{2i+1}sin(m\theta_i) \\ x_{2i}sin(m\theta_i) + x_{2i+1}cos(m\theta_i) \end{pmatrix}, \quad (1.6)$$

де $\theta_i = 10000^{-2i/d}$.

1.3.4 SwiGLU

Стандартний FFN блок у Transformer складається з двох лінійних перетворень з нелінійністю ReLU між ними: $\mathrm{FFN}(x) = max(0, xW_1 + b_1)W_2 + b_2$. Shazeer [6] запропонував SwiGLU – варіант Gated Linear Units з функцією активації SiLU (Sigmoid Linear Unit, також відома як Swish). SwiGLU використовує три лінійні проєкції замість двох:

$$\mathrm{SwiGLU}(x) = (\mathrm{SiLU}(xW_{gate}) \odot xW_{up}) \cdot W_{down}, \qquad (1.7)$$

де $\mathrm{SiLU}(x) = x \cdot \sigma(x)$.

Гейтуючий механізм дозволяє мережі адаптивно контролювати потік інформації, що емпірично призводить до кращої якості при порівнянній кількості параметрів.

1.3.5 KV Cache

При авторегресійній генерації декодер послідовно генерує токени один за одним, і на кожному кроці t повинен обчислити attention з усіма попередніми позиціями 1,…,t-1. Без оптимізації це означає, що на кожному новому кроці ключі та значення для всіх раніше згенерованих токенів обчислюються заново, хоча самі ці токени та їхні представлення вже не змінюються – обчислення фактично дублюється. Це призводить до квадратичної обчислювальної складності O(t^2) для генерації послідовності довжиною t, що суттєво уповільнює інференс, особливо для довгих послідовностей. KV Cache вирішує цю проблему, зберігаючи вже обчислені key та value тензори попередніх кроків у пам'яті та додаючи на кожному новому кроці лише key і value нового токена, замість повторного перерахунку всієї історії. Такий підхід зменшує складність до O(t) на

кожному кроці генерації, суттєво прискорюючи інференс ціною додаткових витрат відеопам'яті на зберігання кешу.

1.4 Стратегії навчання мовних моделей

Сучасне навчання мовних моделей базується на парадигмі transfer learning: модель спочатку проходить етап передтренування (pre-training) на великому неанотованому корпусі, а потім дотренування (fine-tuning) на конкретній задачі з анотованими даними.

Ключовим теоретичним орієнтиром для планування навчання є Scaling Laws – степеневі залежності якості моделі від розміру параметрів, обсягу даних та обчислювального бюджету [7]. Hoffmann et al. [8] у роботі «Training Compute-Optimal Large Language Models» (відомій як «закон Chinchilla») продемонстрували, що для оптимального використання фіксованого обчислювального бюджету кількість навчальних токенів повинна масштабуватися пропорційно кількості параметрів моделі з оптимальним співвідношенням приблизно 1:20 (токени:параметри). Подальші дослідження [9] показали, що для моделей, оптимізованих під ефективність інференсу (а не лише навчання), це співвідношення може бути значно більшим – до 1:200 і вище, оскільки «перетренована» мала модель може бути ефективнішою за «оптимально навчану» велику при однакових витратах на інференс. Практичним підтвердженням цієї тенденції є модельне сімейство LLaMA [10], навчене зі значно вищим співвідношенням токенів до параметрів, ніж передбачає класичний Chinchilla-оптимум.

У контексті даної роботи модель з 100M параметрами навчається на 384M токенах, що дає співвідношення приблизно 1:4 – значно нижче оптимуму Chinchilla. Це є свідомим компромісом: мета проєкту полягає у демонстрації повного реального процесу створення мовної моделі, що вимагає архітектури достатнього масштабу для осмислених результатів. Модель із суттєво меншою кількістю параметрів (наприклад, 10–30M) не

здатна продемонструвати характерні закономірності навчання LLM і є нерепрезентативною. Водночас повноцінне навчання 100M-параметрової моделі за оптимумом Chinchilla потребувало б приблизно 2B токенів, що вимагає значно більших обчислювальних ресурсів та часу, ніж було доступно. Тому обрано стратегію свідомо недонавченої, але архітектурно повноцінної моделі. Навіть при такому обмеженому бюджеті різниця між фільтрованими та нефільтрованими даними є статистично значущою, що підкреслює фундаментальність виявленого ефекту.

Для передтренування encoder-decoder моделей ефективним підходом є denoising objective, зокрема text infilling, запропонований у моделі BART [11]. Модель навчається відновлювати оригінальний текст із його зіпсованої (corrupted) версії. Вхідна послідовність розбивається на випадкові спани, кожен з яких замінюється єдиним спеціальним токеном [MASK], а decoder повинен згенерувати повний оригінальний текст на основі цього зіпсованого входу. Довжини спанів визначаються геометричним розподілом. На відміну від T5, де decoder відновлює лише самі замасковані спани (коротший target) [12], BART вимагає повного відновлення послідовності, що привчає decoder генерувати зв'язний текст цілком і краще узгоджується із задачами генерації, хоча й ціною вищих обчислювальних витрат на крок навчання порівняно з T5.

1.5 Проблема якості даних для навчання мовних моделей

Якість навчальних даних є, мабуть, найважливішим і водночас найбільш трудомістким аспектом побудови мовної моделі. Ця теза є центральною для всієї роботи.

Сучасні мовні моделі потребують великих обсягів тексту для навчання. Основним джерелом таких даних є веб-корпуси – колекції текстів, автоматично зібрані з інтернету. Серед найвідоміших – Common Crawl (петабайти сирих HTML-сторінок) та OSCAR [13] (очищена версія

Common Crawl, класифікована за мовами). Ці корпуси надають практично необмежений обсяг тексту, проте їхня якість є вкрай неоднорідною.

Типові забруднення веб-корпусів включають:

– spam та SEO-сміття: тексти, створені виключно для маніпулювання пошуковими системами, з надмірним повторенням ключових слів та безглуздим вмістом;

– boilerplate: повторювані елементи веб-сторінок – навігаційні меню, повідомлення про cookies, політики конфіденційності, форми підписки. Ці тексти ідентичні на тисячах сторінок;

– небажаний контент: азартні ігри, контент для дорослих, фінансове шахрайство;

– низькоентропійний текст: списки, таблиці, згенерований текст з низькою лінгвістичною різноманітністю;

– кодування та мовні артефакти: неправильно розкодовані символи, змішання мов в одному документі.

Для мовних моделей ці забруднення є особливо шкідливими: модель витрачає свою обмежену параметричну ємність на запам'ятовування повторюваного спаму замість вивчення справжніх мовних структур та знань про світ. Як буде продемонстровано у розділах 4 та 5 цієї роботи, модель, навчена на нефільтрованих даних, може демонструвати нижчий training loss, але при цьому мати гіршу здатність до генералізації на downstream задачах – класичний прояв перенавчання на низькоентропійному шумі.

Існує два основні підходи до фільтрації даних. Перший – використання великих мовних моделей як суддів (LLM-as-a-judge), де мовна модель оцінює якість кожного документа. Цей підхід забезпечує високу якість класифікації, проте є непомірно дорогим для обробки мільйонів документів, що робить його непрактичним для команд з обмеженим бюджетом.

Другий підхід – евристичні пайплайни фільтрації, засновані на детермінованих правилах: регулярних виразах, статистичних метриках

тексту та токенайзерних евристиках. Хоча кожне окреме правило може бути менш точним за LLM-суддю, ретельно спроєктована каскадна система евристик забезпечує порівнянну якість при незрівнянно нижчій вартості [14]. Цей підхід покладено в основу низки сучасних відкритих корпусів, зокрема FineWeb [15], що демонструє ефективність масштабованої евристичної фільтрації Common Crawl без використання моделей-суддів. Саме цей підхід обрано у даній роботі, і його ефективність підтверджується емпірично.

1.6 Висновки до першого розділу

У даному розділі проведено аналітичний огляд сучасного стану досліджень у галузі мовних моделей та їхнього навчання. Розглянуто еволюцію мовних моделей від статистичних n-грамних моделей до сучасних Transformer-архітектур. Детально описано архітектуру Transformer та її ключові сучасні модифікації – Pre-Layer Normalization, Grouped Query Attention, Rotary Positional Embeddings та SwiGLU, – які будуть використані при побудові моделі у розділі 3.

Проаналізовано стратегії навчання мовних моделей, зокрема закон Chinchilla та denoising objectives, що визначили стратегію навчання у розділі 4. Особливу увагу приділено проблемі якості даних для навчання мовних моделей – центральній темі всієї роботи. Обґрунтовано вибір евристичного підходу до фільтрації як економічно виправданої альтернативи методу LLM-as-a-judge.

Проведений аналіз підтверджує, що побудова якісної мовної моделі – це насамперед задача підготовки даних. Архітектурні рішення та стратегії навчання є важливими, проте без якісних даних навіть найсучасніша архітектура не зможе досягти задовільної генералізації.

2 ПІДГОТОВКА ДАНИХ

2.1 Загальна стратегія та джерела даних

Джерелом навчальних даних обрано корпус OSCAR-2301 [13] – очищену версію Common Crawl, класифіковану за мовами. Для побудови білінгвальної мовної моделі використано англійську (EN) та німецьку (DE) підмножини корпусу. Вибір саме цієї мовної пари зумовлений кількома факторами: обидві мови добре представлені в корпусі, вони належать до однієї мовної сім'ї (германські мови), проте мають суттєві структурні відмінності (зокрема, композитні слова в німецькій мові), що робить задачу перекладу нетривіальною.

Дані завантажувалися у потоковому режимі (streaming) з Hugging Face Hub з використанням бібліотеки datasets [16]. Англійська та німецька підмножини об'єднувалися через `interleave_datasets` з рівними ймовірностями (0.5/0.5), що забезпечило збалансоване представлення обох мов. Для рандомізації порядку документів використано буфер перемішування розміром 10 000 елементів.

Центральною філософією підготовки даних є пріоритет чистоти над об'ємом. Оскільки для малих моделей веб-корпус є практично нескінченним джерелом тексту, мінімізація False Negatives (допуск низькоякісного тексту у навчальний набір) є значно важливішою за мінімізацію False Positives (випадкове відхилення якісного тексту).

Втрата окремих якісних документів не є критичною, тоді як кожен неякісний документ витрачає обмежену параметричну ємність моделі. Кінцева мета – відсіяти приблизно 90% сирих даних, залишивши лише документи з високим SNR (Signal-to-Noise Ratio, відношення сигнал-шум).

2.2 Навчання токенізатора

Перед основною фільтрацією необхідно навчити токенізатор – компонент, що перетворює текст у послідовність числових ідентифікаторів (токенів), з якими працює модель. Вибір алгоритму токенізації та якість навчальних даних для нього безпосередньо впливають на ефективність моделі.

Обрано алгоритм Unigram [17], реалізований через бібліотеку tokenizers від Hugging Face. Unigram підхід починає з великого початкового словника і ітеративно видаляє найменш корисні токени, оптимізуючи ймовірнісну модель. На відміну від BPE (Byte Pair Encoding), який жадібно об'єднує найчастіші пари, Unigram знаходить глобально оптимальнішу сегментацію, що емпірично показує кращі результати для морфологічно багатих мов, таких як німецька.

Параметри токенайзера:

– розмір словника (vocab_size): 30 000 токенів – компроміс між гранулярністю представлення та розміром embedding-матриці (яка при d_model=512 складає 30000 * 512 = 15.36M параметрів, тобто приблизно 16% від загальної кількості параметрів моделі);

– pre-tokenizer: Metaspace – поділяє текст на підслова з урахуванням пробілів, замінюючи пробіли на спеціальний символ. Це зберігає інформацію про межі слів у токенізованому представленні;

– спеціальні токени: [PAD] (ID=0) для доповнення до фіксованої довжини, [UNK] (ID=1) для невідомих символів, [EN] (ID=2) та [DE] (ID=3) – мовні маркери для encoder, [EOS] (ID=4) – кінець послідовності, [MASK] (ID=5) – маска для denoising objective.

Навчальні дані для токенайзера проходять попередню очистку: нормалізація Unicode-символів, заміна типографських лапок на стандартні, конвертація валютних символів у текстове представлення ($→dollars, euro→euros), заміна переносів рядків на спеціальний токен [NL]. Ця

стандартизація зменшує розмір необхідного словника та покращує узгодженість токенізації.

2.3 Багатоетапний конвеєр фільтрації

Конвеєр фільтрації складається з трьох послідовних рівнів, кожен з яких фокусується на своєму типі забруднень. Документ відхиляється при першому невиконанні будь-якого критерію, що забезпечує ефективну каскадну фільтрацію. Формально, для документа D конвеєр визначається як послідовність трьох предикатів, кожен з яких формалізовано нижче. Усі відсотки відхилень розраховані відносно початкового обсягу сирого корпусу.

2.3.1 Перший рівень: регулярні вирази та структурні фільтри (49% відсіювання)

Перший рівень фільтрації використовує регулярні вирази для виявлення двох категорій проблемного контенту з принципово різними стратегіями обробки.

Формально, рівень 1 визначається предикатом:

$$\exists D' = T(D): P(D) = 0 \wedge Lc(D') \geq \tau c \wedge Lw(D') \geq \tau w, \qquad (2.1)$$

де $P(D)$ – предикат наявності poison-паттернів;

$T(D)$ – функція обрізки шаблонного тексту;

L_c та L_w – довжина в символах та словах відповідно;

$\tau_c = 512$;

$\tau_w = 128$.

Poison-паттерни – це маркери контенту, що повністю дискваліфікує документ. При виявленні будь-якого з цих паттернів у будь-якому місці

документа він негайно відхиляється без можливості відновлення. Категорії poison-паттернів наведено у таблиці 2.1.

Таблиця 2.1 – Категорії poison-паттернів

Категорія	Приклади (EN)	Приклади (DE)
Азартні ігри	casino, gambling, slots, betting	Spielautomaten, Spielbank, Willkommensbonus
Контент для дорослих	porn, escort, hookup, onlyfans	Sexkontakte, Erotik
Фінансове шахрайство	payday loans, binary options, get rich quick	Kredit ohne Schufa, schnelles Geld
Криптовалютний спам	ICO, airdrop, pump and dump	Kryptowährung
SEO-маніпуляції	click here, buy now, limited offer	jetzt kaufen, hier klicken
Академічне шахрайство	essay writing, buy answers	Abitur Lösung, Prüfung Antworten

Boilerplate-паттерни, на відміну від poison, не призводять до негайного відхилення. Вони маркують шаблонний контент, типовий для веб-сторінок, який зазвичай концентрується на початку та в кінці документа. Категорії наведено у таблиці 2.2.

Таблиця 2.2 – Категорії boilerplate-паттернів

Категорія	Приклади (EN)	Приклади (DE)
Навігаційні елементи	skip to content, main navigation	Zur Hauptnavigation, Navigation überspringen

Продовження таблиці 2.2

Категорія	Приклади (EN)	Приклади (DE)
Cookie/GDPR	accept all, configure cookies	Datenschutz, Datenschutzerklärung
Правові застереження	privacy policy, terms of use, all rights reserved	Impressum, AGB, Haftungsausschluss
Електронна комерція	add to cart, shopping cart, checkout	in den Warenkorb, Sortieren nach
Автентифікація	sign in, log in, register	Anmelden, Registrieren, Konto erstellen
Соціальні мережі	tweet share pin, share on Facebook	

При виявленні boilerplate в перших або останніх 200 символах документа застосовується алгоритм trim_text, який намагається відновити документ, обрізавши його до найближчого валідного речення. Алгоритм послідовно перевіряє перші (або останні) 24 рядки, шукаючи рядок, що містить більше 10 слів та хоча б один пунктуаційний знак (. , ! ?). Якщо такий рядок знайдено, текст обрізається до цієї точки. Після обрізки документ повторно перевіряється: якщо boilerplate все ще присутній у нових межах, або якщо довжина тексту впала нижче мінімальних порогів (512 символів та 128 слів), документ остаточно відхиляється.

2.3.2 Другий рівень: статистичні та лексичні евристики (10% відсіювання)

Другий рівень аналізує статистичні характеристики тексту для виявлення документів з аномальною структурою, що не були відловлені паттернами першого рівня. Формально:

$$TTR(D') \geq f(Lw) \wedge Vlex(D') \in [Bmin(L), Bmax(L)], \qquad (2.2)$$

де TTR – відношення типів до токенів;

$f(L_w)$ – динамічний поріг залежно від кількості слів;

V_{lex} – вектор лексичних метрик;

$B^{(L)}$ – мовнозалежні межі.

Type-Token Ratio (TTR) – відношення кількості унікальних слів до загальної кількості слів. Низький TTR вказує на високу повторюваність, характерну для spam-текстів та автоматично згенерованого контенту. Пороги встановлено динамічно залежно від довжини документа: 0.4 для коротких текстів (до 500 слів), 0.35 для середніх (до 2000 слів) та 0.3 для довгих (2000+ слів). Зниження порогу для довших текстів враховує природне повторення слів у великих документах.

Середня довжина слова обмежена діапазоном [3.0, 12.0] символів для англійської та [3.0, 15.0] для німецької мови. Розширений верхній поріг для німецької мови враховує композитну морфологію – здатність утворювати довгі складені слова (наприклад, Bundesverfassungsgericht, Datenschutzgrundverordnung).

Digit ratio (частка цифр у тексті) обмежена значенням 0.18. Документи з надмірною кількістю цифр зазвичай є таблицями даних, фінансовими звітами або технічними специфікаціями, які не несуть корисної лінгвістичної інформації для мовної моделі.

Newline ratio (частка переносів рядків) обмежена значенням 0.02. Висока частка переносів вказує на списки, навігаційні меню або поетичний контент із надмірним форматуванням.

Alpha-word ratio (частка слів, що складаються лише з літер) встановлена на рівні 0.70 для англійської та 0.60 для німецької. Зниження порогу для німецької враховує більш поширене використання дефісів та спеціальних символів (наприклад, умляутів у складених формах).

Uppercase ratio (частка великих літер) обмежена значенням 0.07 для англійської та 0.13 для німецької. Вищий поріг для німецької зумовлений правилом написання іменників з великої літери (Großschreibung), яке є нормою німецької орфографії.

2.3.3 Третій рівень: токенайзерні евристики (31% відсіювання)

Третій рівень використовує навчений токенайзер як інструмент оцінки якості тексту. Цей підхід дозволяє виявити забруднення, невидимі для текстових евристик: математичний мусор, контент на невідомих мовах, тексти з нестандартним кодуванням. Формально, для кожного чанка c документа D':

$$\forall c \in D' : \mu ID(c) \leq \delta \wedge \mu len(c) \in [\epsilon 1, \epsilon 2] \wedge U(c) \leq \zeta, \qquad (2.3)$$

де μ_{ID} – середній ID токена;

$\delta = 7000$; μ_{len} – середня довжина токена;

$\epsilon_1 = 3.2$, $\epsilon_2 = 5.0$; U – кількість невідомих токенів;

$\zeta = 1$.

Перед токенізацією текст проходить стандартизацію через словник TEXT_REPLACEMENTS: типографські лапки замінюються на стандартні, символи валют – на текстові еквіваленти, переноси рядків – на токен [NL]. Це забезпечує узгодженість з даними, на яких навчався токенайзер.

Документ розбивається на чанки (фрагменти) по 128 токенів (з урахуванням 2 спеціальних токенів, максимум 126 контентних токенів на чанк). Кожен чанк оцінюється незалежно за трьома критеріями:

Максимальний середній ID токена (`max_avg_token_id` <= 7000). Токени у словнику Unigram впорядковані за частотою: часті слова мають низькі ID, рідкісні – високі. Чанк з високим середнім ID містить переважно рідкісні токени, що вказує на екзотичний або зашумлений текст. Поріг 7000

при словнику 30 000 означає, що текст повинен переважно складатися з верхніх 23% найчастіших токенів.

Середня довжина токена в символах (`avg_token_len` in [3.2, 5.0]). Занадто короткі токени (< 3.2 символи) вказують на посимвольну токенізацію, характерну для нерозпізнаних скриптів або кодування. Занадто довгі токени (> 5.0 символів) вказують на незвичний текст, з яким токенайзер не справляється.

Максимальна кількість невідомих токенів (`max_unk_count` = 1 на чанк). Кожний [UNK] токен означає, що токенайзер зустрів символ або послідовність, яких не було в навчальних даних. Більше одного [UNK] на 128 токенів вказує на суттєві проблеми з кодуванням або мовою тексту.

Максимум 16 чанків зберігається з одного документа, що обмежує вклад надмірно довгих документів та забезпечує різноманітність навчальних даних.

2.4 Генерація синтетичного паралельного датасету

Для fine-tuning моделі на задачі машинного перекладу необхідний паралельний корпус – набір пар речень на вихідній та цільовій мовах. Однак існуючі відкриті паралельні датасети виявилися непридатними для цілей проєкту.

OpenSubtitles – один з найбільших відкритих паралельних корпусів [18] – складається переважно з дуже коротких пар (1–2 речення), які є субтитрами до фільмів. Такий формат не відображає повноцінної мовної моделі: речення часто фрагментарні, контекстно залежні та стилістично обмежені розмовним жанром.

Europarl [19] та WMT News Commentary [20] мають протилежну проблему: вони складаються з офіційних парламентських промов та новинних текстів відповідно. Використання таких даних для доналаштування ризикує «вбити» лінгвістичну різноманітність, набуту

моделлю під час pre-training на OSCAR, замінивши її вузьким офіційним стилем.

Ідеальний датасет для доналаштування повинен відповідати розподілу pre-training даних за стилем, лексикою та довжиною речень, щоб зберегти широту мовних знань моделі. Оскільки такий датасет знайти не вдалось, було прийнято рішення згенерувати його синтетично.

Пайплайн генерації синтетичного датасету складається з наступних етапів:

– відбір вихідних речень: з відфільтрованого OSCAR-корпусу на обох мовах відбираються якісні тексти, які проходять сегментацію на речення за допомогою бібліотеки NLTK [21];

– формування фрагментів: речення випадково групуються у фрагменти від 1 до 7 речень. Така варіативність забезпечує різноманітність довжин у фінальному датасеті;

– пакетний переклад: фрагменти перекладаються з німецької на англійську за допомогою Gemini API (модель gemini-2.5-flash-lite) [22]. Промпт інструктує модель зберігати стиль та регістр оригіналу, повертаючи лише переклад без пояснень;

– збирання результатів: перекладені пари збираються у структурований датасет з полями source (DE) та target (EN).

Фінальний датасет містить приблизно 80 000 пар DE-EN та опублікований на Hugging Face Hub для відтворюваності дослідження.

Оскільки в кожну з вибірок мов попадають як згенеровані, так і оригінальні тексти, ми також уникаємо упередженості до стилю мовлення обраної моделі перекладача.

2.5 Висновки до другого розділу

У даному розділі детально описано процес підготовки даних для побудови мовної моделі – етап, що виявився найбільш трудомістким та водночас найважливішим у всьому пайплайні.

Розроблено та реалізовано багатоетапний конвеєр фільтрації веб-корпусу OSCAR, що складається з трьох рівнів: регулярних виразів та структурних фільтрів (49% відсіювання), статистичних та лексичних евристик (10% відсіювання) та токенайзерних евристик (31% відсіювання). Сукупно конвеєр відсіює приблизно 90% сирих документів, максимізуючи Signal-to-Noise Ratio навчальних даних.

Навчено доменний Unigram токенайзер зі словником 30 000 токенів, адаптований під відфільтрований білінгвальний корпус. Згенеровано синтетичний паралельний датасет з приблизно 80 000 пар DE-EN за допомогою Gemini API, що зберігає лінгвістичну різноманітність pre-training корпусу та є доступним для відтворення через Hugging Face Hub.

Емпіричне підтвердження ефективності конвеєра фільтрації буде представлено у розділах 4 та 5, де порівнюються результати моделей, навчених на фільтрованих та нефільтрованих даних.

3 АРХІТЕКТУРА МОДЕЛІ

3.1 Загальна структура моделі

Для вирішення задач мовного моделювання та машинного перекладу реалізовано кастомну encoder-decoder архітектуру ModernEncoderDecoder на базі фреймворку PyTorch [23]. Модель містить приблизно 100M параметрів (98 276 352), що відносить її до класу малих мовних моделей. Основні гіперпараметри архітектури наведено у таблиці 3.1.

Таблиця 3.1 – Гіперпараметри архітектури моделі

Параметр	Значення
Розмірність моделі (d_{model})	512
Кількість голів уваги (n_{head})	8
Кількість KV голів (n_{kv})	2
Кількість шарів encoder	9
Кількість шарів decoder	11
Розмір словника	30 000
Максимальна довжина послідовності	1024
Dropout	0.1

Асиметрична конфігурація (9 encoder, 11 decoder шарів) обрана свідомо. Decoder виконує складнішу задачу – авторегресійну генерацію тексту, що вимагає більшої глибини для моделювання складних

залежностей. Encoder, натомість, обробляє вхідну послідовність цілком з двобічним контекстом, що дещо компенсує меншу кількість шарів.

Обидві частини моделі використовують спільну embedding-матрицю розміром $30000 \times 512 = 15.36\text{M}$ параметрів. Вихідний лінійний шар (`fc_out`) має окрему матрицю ваг тієї ж розмірності, що забезпечує незалежне навчання вхідних та вихідних представлень токенів.

3.2 Grouped Query Attention (GQA)

Реалізація GQA у моделі використовує 8 query голів та 2 KV голови, що дає співвідношення 4:1. Кожна пара з чотирьох query голів спільно використовує одну пару key/value проєкцій. Це означає, що при `d_model=512` та `head_dim=64` (512/8) проєкції Q мають розмірність 512x512, тоді як проєкції K та V – лише 512x128 (512 x 2*64).

У PyTorch реалізації використано параметр `enable_gqa=True` у функції `F.scaled_dot_product_attention`, що забезпечує нативну підтримку GQA на рівні оптимізованого CUDA-ядра. Це потенційно ефективніше за ручну реалізацію через repeat_interleave, яка явно дублює KV тензори для кожної групи query голів. У коді збережено закоментовану ручну реалізацію як альтернативний підхід для розуміння механіки GQA:

```
# n_repeats = self.n_head // self.kv_head
# keys = xk.repeat_interleave(n_repeats, dim=2)
```

Економія пам'яті від GQA є особливо відчутною при використанні KV Cache під час генерації: кешувати потрібно лише 2 KV голови замість 8, що зменшує витрати пам'яті на кеш у 4 рази.

3.3 Rotary Positional Embeddings (RoPE)

Реалізація RoPE у моделі базується на попередньому обчисленні матриць cos та sin для всіх можливих позицій (до `max_seq_len=4096`). Для

кожної позиції m обчислюються частоти, де `dim` – розмірність голови уваги (`head_dim = d_model / n_head = 64`):

```
inv_freq = 1 / (10000 ^ (2i / dim))
freqs = outer_product(positions, inv_freq)
```

Обчислені `freqs_cos` та freqs_sin зберігаються як буфери моделі (`register_buffer`), що означає їхню присутність у `state_dict` без участі в оптимізації градієнтів.

Важливою технічною особливістю є відмова від використання комплексних чисел PyTorch (`torch.complex64`). Стандартна реалізація RoPE через комплексну арифметику (torch.polar, множення комплексних тензорів) не підтримується повною мірою у трасуванні обчислювального графу через torch.compile та SafeTensors. Тому використано реалізацію через пари дійсних чисел: вхідний тензор розбивається на пари (`x_r`, `x_i`) як дійсна та уявна компоненти, і обертання виконується явно:

```
out_r = x_r * cos - x_i * sin
out_i = x_r * sin + x_i * cos
```

Математично це еквівалентно множенню комплексного числа (`x_r` + `jx_i`) на одиничний вектор обертання (`cos(theta)` + `jsin(theta)`), де `theta = m * 10000^(-2i/d)`. Результат у матричній формі для кожної пари елементів:

```
[out_r]   [cos(theta)  -sin(theta)] [x_r]
[out_i] = [sin(theta)   cos(theta)] [x_i]
```

Цей підхід зберігає математичну коректність обертання у комплексному просторі, але записаний виключно через операції над дійсними числами, що забезпечує повну сумісність з компіляцією моделі.

RoPE застосовується виключно у self-attention шарах – як encoder, так і decoder. У cross-attention шарах decoder, де query походять від decoder, а key/value – від encoder, RoPE не використовується. Це пояснюється тим, що позиційна інформація encoder-послідовності вже закодована у виходах encoder через self-attention з RoPE. Додаткове застосування RoPE у cross-attention було б семантично некоректним, оскільки позиції query та key

належать до різних послідовностей і їхнє відносне положення не несе позиційної інформації.

У реалізації ця відмінність досягається передачею `rope_kwargs=None` у конструктор `GroupedQueryAttention` для cross-attention:

```
self.self_attn = GroupedQueryAttention(...,
rope_kwargs=rope_kwargs)
self.cross_attn = GroupedQueryAttention(...,
rope_kwargs=None)
```

3.4 SwiGLU Feed-Forward Network

Формально, SwiGLU обчислюється як:

$$\text{SwiGLU}(x) = (\text{SiLU}(xW_{gate}) \odot xW_{up}) \cdot W_{down}, \qquad (3.1)$$

де $\text{SiLU}(x) = x \cdot \sigma(x)$.

Три лінійні проєкції W_{gate} (w1), W_{up} (w3) та W_{down} (w2) реалізовані без bias (зсуву – окремого навчального параметра, що додається до результату множення на матрицю ваг і не залежить від вхідних даних). Відсутність bias є стандартною практикою для сучасних Transformer-моделей: вона зменшує кількість параметрів без суттєвої втрати виразності, оскільки LayerNorm перед кожним підшаром вже виконує нормалізацію та зсув через власні параметри γ та β.

Розмірність прихованого шару (`hidden_dim`) обчислюється автоматично:

```
hidden_dim = 4 * d_model  # початкова оцінка
hidden_dim = int(2 * hidden_dim / 3)  # корекція для
SwiGLU
hidden_dim = multiple_of * ceil(hidden_dim / multiple_of)
# вирівнювання
```

Множник 2/3 компенсує додаткові параметри третьої проєкції (w3), зберігаючи загальну кількість параметрів FFN порівнянною з класичним ReLU FFN. Вирівнювання до кратного 256 (`multiple_of=256`) забезпечує оптимальне використання тензорних ядер GPU, які працюють найефективніше з розмірностями, кратними степеням двійки або їхнім множникам.

Для `d_model=512` ефективна розмірність `hidden_dim` після обчислень становить 1536 (512 * 4 = 2048, 2048 * 2/3 = 1365, вирівнювання до 1536). Кожен SwiGLU блок містить три матриці 512x1536 (w1, w3) та одну 1536x512 (w2), що дає приблизно 2.36M параметрів на блок.

3.5 Pre-Layer Normalization

Кожен шар encoder містить два підшари (self-attention та FFN), кожен з яких оточений Pre-LN паттерном:

```
x = x + dropout(self_attn(LayerNorm(x)))
x = x + dropout(ffn(LayerNorm(x)))
```

Кожен шар decoder містить три підшари з аналогічною структурою:

```
x = x + dropout(self_attn(LayerNorm(x)))
x = x + dropout(cross_attn(LayerNorm(x)))
x = x + dropout(ffn(LayerNorm(x)))
```

Фінальний LayerNorm (`final_norm`) застосовується після останнього шару decoder перед проєкцією у простір словника. Це стандартна практика для Pre-LN архітектур, де останній підшар може мати нестабільну масштабність виходів.

Dropout з ймовірністю 0.1 застосовується після кожного підшару як регуляризація. Окремі dropout-модулі використовуються для self-attention, cross-attention та FFN, що дозволяє потенційно налаштовувати їх незалежно.

3.6 Ініціалізація ваг

Коректна ініціалізація ваг є критичною для стабільного навчання глибоких моделей. У реалізації використано наступну стратегію:

Лінійні шари (`nn.Linear`) та embedding (`nn.Embedding`) ініціалізуються нормальним розподілом N(0, 0.02). Стандартне відхилення 0.02 емпірично показало хорошу збіжність для Transformer-моделей розміром 100M-1B параметрів.

Для output проєкцій attention (wo) та FFN (w2) застосовується attenuating initialization за підходом GPT-2 [24]: стандартне відхилення масштабується множником 1/sqrt(2*N), де N – загальна кількість residual шарів. Це зменшує початкову амплітуду виходів підшарів, запобігаючи експоненційному зростанню активацій при проходженні через глибокий стек residual з'єднань.

LayerNorm ініціалізується стандартно: ваги (gamma) – одиницями, зміщення (beta) – нулями, що забезпечує ідентичне перетворення на початку навчання.

3.7 KV Cache для ефективної генерації

При авторегресійній генерації модель на кожному кроці t генерує один токен, використовуючи всі попередньо згенеровані токени як контекст. Без KV Cache кожен крок вимагає повного прямого проходу через всі шари декодера для всіх t токенів, причому key та value тензори перераховуються для кожної позиції.

Реалізований KV Cache зберігає key та value тензори кожного шару decoder після обчислення. На наступному кроці генерації:

– через шари decoder проходить лише новий токен (`seq_len_q=1`);

– його key та value обчислюються та конкатенуються з кешованими: `xk = cat([cached_k, new_k], dim=1);`

– attention обчислюється між одним query та всіма накопиченими keys/values;

– оновлений кеш передається на наступний крок.

Позиція нового токена для RoPE визначається як `start_pos = cached_k.shape[1],` що забезпечує коректне позиційне кодування навіть при покроковій генерації.

Кількісна оцінка прискорення від KV Cache наведена у розділі 5.4.

3.8 Стратегії семплування при генерації

Метод `generate()` моделі підтримує кілька стратегій семплування для контролю якості та різноманітності генерованого тексту:

Temperature – параметр, що масштабує логіти перед softmax: `logits = logits / temperature`. При temperature < 1.0 розподіл стає більш «гострим», збільшуючи ймовірність найвірогідніших токенів. При temperature > 1.0 розподіл стає більш рівномірним, збільшуючи різноманітність генерації.

Top-K sampling – обмеження вибірки до K найвірогідніших токенів. Логіти решти токенів встановлюються у `-inf`. Це запобігає вибору маловірогідних токенів з «хвоста» розподілу, які можуть бути безглуздими.

Top-P (nucleus) sampling – динамічне обмеження, де вибірка обмежується найменшою множиною токенів, кумулятивна ймовірність яких перевищує поріг P. На відміну від Top-K, цей метод адаптується до форми розподілу: при впевненості моделі в наступному токені множина буде малою, при невизначеності – більшою.

У реалізації ці стратегії можуть комбінуватися: спочатку застосовується temperature, потім Top-K, потім Top-P.

3.9 Висновки до третього розділу

У даному розділі детально описано архітектуру кастомної encoder-decoder моделі з приблизно 100M параметрами. Модель інтегрує чотири ключові сучасні модифікації Transformer: Pre-Layer Normalization для стабільності навчання, Grouped Query Attention для ефективності обчислень та пам'яті, Rotary Positional Embeddings для кодування відносних позицій (з workaround для сумісності з torch.compile), та SwiGLU для покращення feed-forward мережі.

Реалізовано KV Cache для прискорення авторегресійної генерації та декілька стратегій семплування (Temperature, Top-K, Top-P). Описано стратегію ініціалізації ваг з attenuating initialization для output проєкцій.

Архітектурні рішення обрані з урахуванням балансу між якістю моделювання, обчислювальною ефективністю та сумісністю з інструментами оптимізації (torch.compile, cuDNN SDPA). Усі компоненти реалізовані як модулі PyTorch, що забезпечує гнучкість конфігурації та можливість відтворення.

4 НАВЧАННЯ ТА ЕКСПЕРИМЕНТИ

4.1 Конфігурація навчання

Навчання проводилося на GPU NVIDIA RTX 3090 (24 GB VRAM) з використанням бібліотеки Hugging Face Trainer [25], яка забезпечує стандартний тренувальний цикл з автоматичним логуванням, чекпоінтингом та підтримкою навчання в змішанній точності.

Вибір HF Trainer замість кастомного тренувального циклу зумовлений прагматичними міркуваннями: для даного масштабу моделі та обсягу даних стандартна інфраструктура Trainer є цілком достатньою, а додатковий контроль кастомного циклу не виправдовує витрат часу на його реалізацію та відлагодження.

Для прискорення навчання використано декілька оптимізацій:

– `torch.compile` з бекендом `aot_eager` – компіляція моделі через Ahead-of-Time eager трасування. Спроби використати більш агресивний бекенд inductor (який генерує оптимізовані Triton/C++ ядра) не вдалися через проблеми з C++ компілятором у середовищі виконання. Бекенд aot_eager забезпечує помірне прискорення через оптимізацію графу обчислень без генерації нативного коду;

– bfloat16 (Brain Floating Point) – формат зі зниженою точністю, що зберігає динамічний діапазон float32 (8 біт експоненти) при зменшеному розрізненні мантиси (7 біт замість 23). Для задач NLP bfloat16 є кращим вибором порівняно з float16, оскільки не потребує loss scaling завдяки достатньому динамічному діапазону, що спрощує тренувальний пайплайн. Використання bfloat16 замість float32 зменшує витрати пам'яті на зберігання параметрів та активацій удвічі, що дозволяє збільшити batch size або довжину послідовності при тому ж обсязі VRAM;

– cuDNN SDPA (Scaled Dot-Product Attention) – використання оптимізованого cuDNN ядра для обчислення attention через PyTorch backend

`torch.backends.cuda.enable_cudnn_sdp(True)`. Це забезпечує апаратно-оптимізовану реалізацію attention, що суттєво швидша за наївну реалізацію. Основні гіперпараметри навчання наведено у таблиці 4.1.

Таблиця 4.1 – Гіперпараметри навчання

Параметр	Значення
Оптимізатор	AdamW [26]
Weight decay	0.05
Learning rate	4×10^{-4}
Scheduler	Cosine annealing
Warmup steps	200
Загальна кількість кроків	5000
Batch size (ефективний)	600
Label smoothing (ε)	0.08

Label smoothing зі значенням 0.08 застосовується до cross-entropy loss. Це «розмазує» частину ймовірнісної маси з ground truth токена на решту словника, що запобігає надмірній впевненості моделі та покращує генералізацію. Формально, замість one-hot розподілу $q(k) = 1$ для правильного класу k, використовується:

$$q'(k) = (1 - \varepsilon) \cdot q(k) + \frac{\varepsilon}{V}, \tag{4.1}$$

де $\varepsilon = 0.08$;

$V = 30000$.

4.2 Передтренування (Pre-training)

Передтренування використовує BART span corruption як denoising objective. Реалізація виконана через кастомний `DataCollator`, який на льоту перетворює вхідні послідовності токенів у пари (`corrupted_input, target`):

– вхідна послідовність довжиною 128 токенів розбивається на випадкові спани (лістинг 4.1). Довжина кожного спану визначається геометричним розподілом із середньою довжиною 6 токенів;

– приблизно 15% токенів (`noise_density = 0.15`) замінюються спеціальним токеном [MASK]. Кожен спан замінюється одним [MASK] токеном, незалежно від довжини спану (лістинг 4.2);

– вхід encoder містить оригінальний текст зі вставленими [MASK] токенами замість спанів;

– вхід decoder містить повну оригінальну послідовність токенів без масок обрізану до останнього токена;

– ціль decoder містить повну оригінальну послідовність токенів зсунуту на один токен.

Лістинг 4.1 – Генерація довжин спанів за геометричним розподілом

```
def generate_span_lengths(num_spans: int, p: float,
min_len: int, max_len: int) -> torch.Tensor:
    lengths =
torch.distributions.Geometric(torch.tensor([p])) \
.sample((num_spans,)).squeeze(-1) + 1
return torch.clamp(lengths, min_len, max_len).long()

    self.p = 1.0 / 6.0
    span_lengths  =  generate_span_lengths(int(num_spans),
self.p, 2, 20)
```

Лістинг 4.2 – Усунення послідовних маскувальних токенів

```
is_mask = raw_ids == self.mask_token
if is_mask.any():
    shifted = torch.roll(is_mask, shifts=1)
    shifted[0] = False
    is_first_mask = is_mask & ~shifted
    collapsed_ids = raw_ids[~is_mask | is_first_mask]
else:
    collapsed_ids = raw_ids
```

Наприклад, для вхідної послідовності токенів:

[EN] The quick brown fox jumps over the lazy dog near the river [EOS]

Після span corruption вхід encoder може мати вигляд:

[EN] The quick [MASK] jumps over [MASK] near the river [EOS]

Вхід decoder:

[EN] The quick brown fox jumps over the lazy dog near the river

А ціль decoder:

The quick brown fox jumps over the lazy dog near the river [EOS]

Модель навчається як відновлювати замасковані фрагменти, так і точно відтворювати решту послідовності, що вимагає розуміння контексту навколо маски та генерації зв'язного тексту в цілому.

Хоча на вхід decoder подається повна послідовність одразу, а не по одному токену, як під час генерації, модель фізично не має доступу до майбутніх токенів.

Це можливо завдяки causal-маскуванню в self-attention decoder: вхід зсунутий на один токен відносно цілі, а нижньотрикутна маска забороняє кожній позиції і звертатися до позицій i+1 і далі. Тому передбачення на позиції і ніколи не залежить від самого y_i чи пізніших токенів – обчислення лише розпаралелені (teacher forcing), а результат еквівалентний послідовній авторегресивній генерації. Наприклад, у наведеному прикладі позиція, що передбачає токен «fox», має доступ лише до «[EN] The quick brown», а не до решти речення. Такий підхід дозволяє обчислити втрати одразу для всієї

послідовності за один прохід замість повільної покрокової генерації, що суттєво прискорює навчання.

Навчання проводилося на двох окремих версіях корпусу:

– фільтрований корпус: дані, отримані після повного конвеєра фільтрації (10% від сирого корпусу). Модель побачила приблизно 384M токенів у потоковому режимі. Додатково відстежувався проміжний чекпоінт після 250M токенів для порівняння ефективності;

– нефільтрований корпус: дані із сирого OSCAR без фільтрації, також у потоковому режимі. Модель побачила аналогічні 384M токенів для контрольного порівняння.

Обидві моделі навчалися з ідентичними гіперпараметрами протягом 5000 кроків, що при batch size 600 та довжині послідовності 128 дає приблизно 5000 * 600 * 128 = 384M токенів за прогін. Оскільки дані подавалися у потоковому режимі (streaming), поняття «епохи» у класичному сенсі не застосовується – модель послідовно обробляла нові документи з корпусу протягом усіх 5000 кроків.

4.3 Дотренування (Fine-tuning)

Fine-tuning проводилося на синтетичному паралельному датасеті DE-EN (приблизно 80 000 пар), описаному у розділі 2.4. Формат введення для fine-tuning:

– вхід encoder: токен мовного маркера [DE] + токенізоване німецьке речення + [EOS];

– ціль decoder: токен мовного маркера [EN] + токенізований англійський переклад + [EOS].

Гіперпараметри fine-tuning відрізнялися від pre-training зменшеним learning rate (5e-5) для збереження набутих під час pre-training мовних знань, та зменшеною кількістю кроків.

Для порівняння навчено три варіанти:

– baseline (без pre-training): модель ініціалізована випадковими вагами, fine-tuned безпосередньо на задачі перекладу;

– pre-trained на нефільтрованих даних (384M токенів): fine-tuned на тому ж датасеті перекладу;

– pre-trained на фільтрованих даних (384M токенів): fine-tuned на тому ж датасеті перекладу. Додатково перевірено проміжний чекпоінт (250M токенів).

4.4 Висновки до четвертого розділу

У даному розділі описано процес навчання моделі, що складається з двох етапів: передтренування на веб-корпусі OSCAR та дотренування на синтетичному датасеті машинного перекладу. Детально описано конфігурацію навчання, включаючи оптимізації (torch.compile, bfloat16, cuDNN SDPA), гіперпараметри та реалізацію span corruption objective.

Навчання проведено на двох версіях корпусу – фільтрованому та нефільтрованому – з ідентичними гіперпараметрами, що забезпечує контрольоване порівняння. Результати навчання та їхній аналіз представлені у наступному розділі.

5 ОЦІНКА ЯКОСТІ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

5.1 Парадокс training loss: фільтровані vs нефільтровані дані

Одним із найцікавіших результатів експерименту є парадокс pre-training loss. Модель, навчена на нефільтрованих даних, досягла дещо нижчого training loss (~1.43) порівняно з моделлю на фільтрованих даних (~1.47) (рисунок 5.1). На перший погляд, це може свідчити про перевагу нефільтрованих даних. Однак детальний аналіз розкриває протилежний висновок.

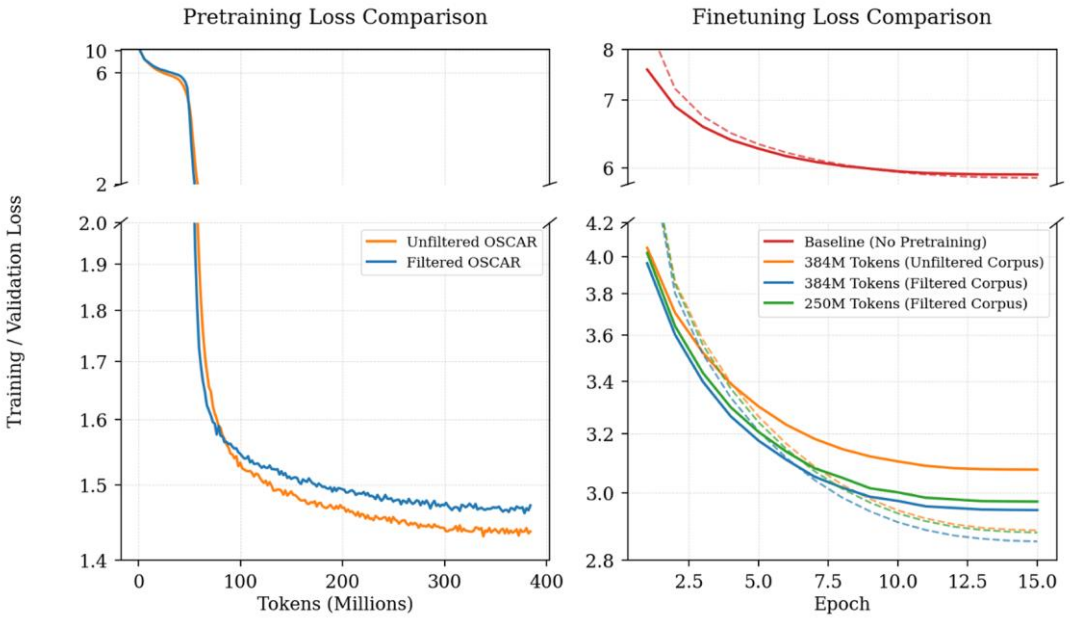


Рисунок 5.1 – Порівняння validation loss при передтренуванні і дотренуванні

Нефільтрований корпус містить значну частку низькоентропійного тексту: повторювані boilerplate-фрагменти, шаблонні конструкції, SEO-spam з обмеженим словниковим запасом. Такий текст має низьку перплексію «з коробки» – модель може досягти низького loss просто запам'ятовуючи часто повторювані паттерни, без справжнього розуміння

мовних структур. Це класичний приклад перенавчання на шумі: модель оптимізує свою функцію втрат за рахунок запам'ятовування тривіальних закономірностей.

Фільтрований корпус, навпаки, містить виключно лінгвістично різноманітний текст із високим TTR та складною структурою. Передбачення наступного токена для такого тексту об'єктивно складніше, що закономірно призводить до вищого training loss. Однак саме ця складність змушує модель вивчати справжні мовні закономірності – синтаксичні структури, семантичні зв'язки та прагматичні патерни, – які потім переносяться на downstream задачі.

5.2 Fine-tuning: валідація на задачі машинного перекладу

Справжню якість мовного моделювання розкриває fine-tuning на downstream задачі – машинному перекладі DE→EN. Саме тут парадокс training loss розв'язується:

Baseline (без pre-training) показав validation loss ~5.9 (рисунок 5.1), що підтверджує принципову необхідність pre-training. Модель з випадковою ініціалізацією не може компенсувати відсутність мовних знань лише на 80 000 парах перекладу.

Pre-training на 384M нефільтрованих токенів дав validation loss ~3.07 – суттєве покращення над baseline, що демонструє цінність pre-training навіть на зашумлених даних.

Pre-training на 384M фільтрованих токенів досяг validation loss ~2.94 – значно кращий результат, що підтверджує перевагу якісних даних.

Найважливіший результат: проміжний чекпоінт фільтрованої моделі на 250M токенів показав validation loss ~2.97 – краще, ніж повний тренінг на 384M нефільтрованих токенів. Це означає, що фільтрована модель перевершила нефільтровану, використавши приблизно на 35% менше токенів.

Цей результат має фундаментальне практичне значення: агресивна евристична фільтрація не лише покращує якість моделі, а й суттєво зменшує обчислювальні витрати на навчання. Для команд з обмеженим бюджетом це означає можливість досягти кращих результатів при менших витратах часу та ресурсів GPU.

5.3 Якісний аналіз перекладів

Для якісної оцінки моделі проведено аналіз перекладів на шести прикладах різного рівня складності – від найпростіших речень (рівень A1) до складних академічних конструкцій (рівень C1) та розмовної мови зі сленгом. Повні тексти вхідних послідовностей та відповідних перекладів наведено у додатку Б (таблиця Б.1). Переклади виконано моделлю, передтренованою на фільтрованому корпусі (384M токенів), з параметрами генерації: `temperature=0.8, top_k=40, top_p=0.8`.

Рівень A1 (прості фактичні речення). Навіть для найпростішого введення модель демонструє нестабільність: плутає лексеми (fahre → drive/ride/walk), генерує галюцинації та дублює речення. Водночас модель коректно передає окремі елементи – "Berlin's big" (Berlin ist groß), "Germany is large" (семантично близьке узагальнення). Це свідчить про часткове засвоєння базової лексики, хоча без стабільної здатності до послідовного перекладу.

Рівень A2 (прості складні речення). Загальний сенс частково вловлений: модель правильно передає ключові іменники та фрази («my name is Anna», «hotel room», «sights», «restaurant»). Це підтверджує, що передтренування на фільтрованому корпусі забезпечило засвоєння базового білінгвального словника. Основна проблема – надмірна генерація: модель не зупиняється після перекладу вхідного тексту, що є характерною ознакою недонавчених авторегресійних моделей.

Рівень B1 (конструкції з підрядними реченнями). Модель демонструє помітний прогрес у обробці граматичних структур: підрядне з «Obwohl» коректно передане як «Although», пасивна конструкція «Mir wurde gesagt» – як «It was said» (прийнятна спрощена форма). Ці результати свідчать про засвоєння базових граматичних відповідностей між мовами. Лексичні помилки (parks → parking) є очікуваними для моделі з обмеженим словниковим охопленням.

Рівень B2 (абстрактні поняття). Модель зберігає розуміння загальної теми («planning my Berlin travel», «tourist attractions», «authentic impression»), демонструючи здатність до семантичного узагальнення. Проте на цьому рівні з'являються грубі галюцинації (підміна Berlin на London), що вказує на межу параметричної ємності моделі для зберігання точних асоціацій.

Рівень C1 (академічний стиль). Попри значні семантичні втрати, модель зберігає академічний регістр тексту та намагається відтворити складну структуру речень. Буквальний переклад «Spannungsfeld» як «tension field» є прийнятним калькуванням. Термін «subkulturellen Strömungen» частково збережений як «subcultural». Це свідчить про часткове засвоєння стилістичних маркерів.

Розмовна мова зі сленгом. Примітно, що сленг «mega krass» передано як «super cool» – одна з найвдаліших передач тону у всій вибірці, яка демонструє здатність моделі до стилістичної адаптації. Проте загалом розмовний регістр не розпізнається, а фраза «was geht ab» перекладена описово замість ідіоматичного «what's up?».

Попри очевидні обмеження, аналіз виявив ряд значущих позитивних закономірностей, які підтверджують ефективність обраного підходу:

– часткове засвоєння білінгвального словника. Модель коректно передає базову лексику (іменники, дієслова повсякденного вжитку) на всіх рівнях складності, що свідчить про ефективність передтренування;

– розуміння граматичних відповідностей. Підрядні речення (Obwohl → Although), пасивні конструкції (Mir wurde gesagt → It was said) та умовний спосіб (частково) передаються, демонструючи засвоєння структурних зв'язків між мовами;

– збереження регістру. Модель відрізняє академічний стиль від розмовного та адаптує лексику відповідно (super cool для сленгу, formal vocabulary для C1);

– семантичне узагальнення. На рівнях B1-B2 модель передає загальну тему тексту, навіть коли точний переклад окремих фраз втрачається.

Основними проблемами залишаються: надмірна генерація, лексичні галюцинації та втрата складних граматичних конструкцій. Якість деградує нелінійно зі зростанням складності: перехід від B1 до B2 супроводжується різким збільшенням помилок. Ці результати є очікуваними для свідомо недонавченої моделі зі 100M параметрами і підтверджують, що навіть при обмеженому обчислювальному бюджеті модель засвоює базові мовні структури, які потенційно можуть бути покращені збільшенням обсягу навчання та параметричної ємності.

5.4 KV Cache: кількісна оцінка прискорення

Для оцінки ефективності реалізованого KV Cache проведено бенчмарк генерації на GPU. Вимірювався час генерації послідовності з 200 токенів:

– без KV Cache: загальний час приблизно 9.0 секунд на 200 токенів (~45 мс/токен у середньому). Час кожного кроку зростає з довжиною згенерованої послідовності, оскільки decoder повторно обчислює key та value тензори для всіх попередніх позицій;

– з KV Cache: загальний час приблизно 4.5 секунди на 200 токенів (~22.5 мс/токен). Час кожного кроку залишається приблизно постійним завдяки кешуванню.

Прискорення у 2 рази є помірним порівняно з великими моделями, де KV Cache забезпечує прискорення у 3-10 разів. Для малих моделей (100M параметрів) накладні витрати на управління кешем (конкатенація тензорів, додаткове копіювання пам'яті через `clone().detach()`) займають відчутнішу частку загального часу обчислення. Проте навіть для моделі даного масштабу прискорення є суттєвим та зростає з довжиною генерованої послідовності.

Додаткову перевагу GQA при використанні KV Cache забезпечує зменшення обсягу кешу: з 2 KV головами замість 8 потрібно зберігати у 4 рази менше тензорів, що зменшує як витрати пам'яті, так і час конкатенації.

5.5 Аналіз обчислювальної ефективності

Загальний час навчання на одній GPU RTX 3090 для 5000 кроків pre-training складав приблизно 8-10 годин на кожну конфігурацію даних. Fine-tuning займав приблизно 2-3 години. Сумарно весь експеримент (pre-training двох моделей + fine-tuning трьох варіантів) вимагав приблизно 30 годин GPU-часу, що є доступним для академічного дослідження на споживчому обладнанні.

Використання bfloat16 зменшило вимоги до VRAM приблизно вдвічі, що дозволило використовувати ефективний розмір батчу 600 на одній GPU з 24 GB. Без змішаної точності максимальний розмір батчу був би приблизно 300, що потребувало б приблизно вдвічі більше кроків для досягнення тієї ж кількості оброблених токенів.

5.6 Висновки до п’ятого розділу

Експериментальні результати підтверджують центральну тезу роботи: якість даних є визначальним фактором генералізації мовної моделі. Ілюзія нижчого training loss (нижчий training loss на нефільтрованих даних) розсіюється при оцінці на downstream задачі, де фільтрована модель демонструє суттєву перевагу.

Ключовий результат: модель, предтренована на 250M фільтрованих токенів, перевершує модель на 384M нефільтрованих токенів за validation loss на задачі перекладу (2.97 vs 3.07). Це еквівалентно приблизно 35% економії обчислювальних витрат при кращій якості.

Реалізований KV Cache забезпечує прискорення генерації приблизно у 2 рази для моделі даного розміру, а GQA зменшує витрати пам'яті на кеш у 4 рази.

ВИСНОВКИ

У даній кваліфікаційній роботі продемонстровано повний шлях побудови малої мовної моделі з нуля – від підготовки та фільтрації даних до оцінки на задачі машинного перекладу. Усі поставлені задачі виконано у повному обсязі.

Проведено аналітичний огляд актуальних підходів до побудови мовних моделей, включаючи архітектуру Transformer та її сучасні модифікації (GQA, RoPE, SwiGLU, Pre-LN), стратегії навчання (Scaling Laws, denoising objectives) та підходи до фільтрації даних. Розроблено та реалізовано триетапний конвеєр фільтрації веб-корпусу OSCAR, що відсіює приблизно 90% документів через послідовне застосування регулярних виразів (49%), статистичних евристик (10%) та токенайзерних евристик (31%), забезпечуючи максимізацію Signal-to-Noise Ratio при нульових витратах на зовнішні API.

Навчено доменний Unigram токенайзер зі словником 30 000 токенів, адаптований під відфільтрований білінгвальний корпус EN-DE. Згенеровано синтетичний паралельний датасет з приблизно 80 000 пар DE-EN за допомогою Gemini API, що зберігає лінгвістичну різноманітність корпусу передтренування та опублікований на Hugging Face Hub для відтворюваності.

Реалізовано кастомну encoder-decoder Transformer архітектуру з приблизно 100M параметрами, що інтегрує сучасні модифікації: Pre-LN для стабільності навчання, GQA (ratio 4:1) для ефективності пам'яті, RoPE для кодування відносних позицій (з реалізацією через дійсні числа для сумісності з torch.compile), SwiGLU для покращення feed-forward мережі, KV Cache для прискорення генерації та стратегії семплування (Temperature, Top-K, Top-P).

Проведено передтренування з використанням T5-style span corruption на двох версіях корпусу (фільтрованому та нефільтрованому) з ідентичними

гіперпараметрами для контрольованого порівняння. Виконано дотренування на задачі машинного перекладу DE→EN із порівнянням трьох конфігурацій: baseline без передтренування, передтренування на нефільтрованих даних та передтренування на фільтрованих даних. Результати підтверджують центральну тезу роботи: модель, передтренована на 250M фільтрованих токенів, перевершує модель на 384M нефільтрованих токенів (validation loss 2.97 vs 3.07), що еквівалентно приблизно 35% економії обчислювальних витрат.

Якісний аналіз перекладів на прикладах різної складності (від A1 до C1) виявив характерні закономірності недонавченої моделі: часткове розуміння семантики навіть складних текстів, проте з системними проблемами надмірної генерації, лексичних галюцинацій та втрати складних граматичних конструкцій. KV Cache забезпечив прискорення генерації приблизно у 2 рази.

Головний висновок роботи полягає у тому, що для побудови якісної мовної моделі, особливо при обмежених ресурсах, підготовка даних є найбільш трудомістким та водночас найважливішим етапом. Архітектурні інновації та стратегії навчання важливі, проте їхній вплив є вторинним відносно якості навчальних даних.

Для команд, що будують спеціалізовані моделі на обмежених бюджетах, евристична фільтрація є не компромісом, а стратегічним імперативом. Ретельно спроєктований конвеєр детермінованих правил забезпечує якість, порівнянну з LLM-as-a-judge, при незрівнянно нижчій вартості, демократизуючи доступ до якісного передтренування.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Attention is all you need / A. Vaswani et al. *arXiv*. 2023. URL: https://doi.org/10.48550/arXiv.1706.03762 (date of access: 15.06.2026).

2. Controllable machine translation based on semantic descriptors for transformer models / O. Borovik et al. *Bionics of intelligence*. 2026. Vol. 2, no. 105. P. 87–99. URL: https://doi.org/10.30837/bi.2026.2(105).09 (date of access: 09.07.2026).

3. On layer normalization in the transformer architecture / R. Xiong et al. *arXiv*. 2020. URL: https://doi.org/10.48550/arXiv.2002.04745 (date of access: 09.07.2026).

4. GQA: training generalized multi-query transformer models from multi-head checkpoints / J. Ainslie et al. *Proceedings of the 2023 conference on empirical methods in natural language processing*, Singapore. Stroudsburg, PA, USA, 2023. URL: https://doi.org/10.18653/v1/2023.emnlp-main.298 (date of access: 09.07.2026).

5. RoFormer: enhanced transformer with rotary position embedding / J. Su et al. *Neurocomputing*. 2023. P. 127063. URL: https://doi.org/10.1016/j.neucom.2023.127063 (date of access: 09.07.2026).

6. Shazeer N. GLU variants improve transformer. *arXiv*. 2020. URL: https://doi.org/10.48550/arXiv.2002.05202 (date of access: 09.07.2026).

7. Scaling laws for neural language models / J. Kaplan et al. *arXiv*. 2020. URL: https://doi.org/10.48550/arXiv.2001.08361 (date of access: 09.07.2026).

8. Training compute-optimal large language models / J. Hoffmann et al. *arXiv*. 2022. URL: https://doi.org/10.48550/arXiv.2203.15556 (date of access: 09.07.2026).

9. Beyond chinchilla-optimal: accounting for inference in language model scaling laws / N. Sardana et al. *arXiv*. 2024. URL: https://doi.org/10.48550/arXiv.2401.00448 (date of access: 09.07.2026).

10. LLaMA: open and efficient foundation language models / H. Touvron et al. *arXiv*. 2023. URL: https://doi.org/10.48550/arXiv.2302.13971 (date of access: 09.07.2026).

11. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension / M. Lewis et al. *Proceedings of the 58th annual meeting of the association for computational linguistics*, Online. Stroudsburg, PA, USA, 2020. URL: https://doi.org/10.18653/v1/2020.acl-main.703 (date of access: 09.07.2026).

12. Exploring the limits of transfer learning with a unified text-to-text transformer / C. Raffel et al. *arXiv*. 2020. URL: https://doi.org/10.48550/arXiv.1910.10683 (date of access: 09.07.2026).

13. Ortiz Suárez P. J., Romary L., Sagot B. A monolingual approach to contextualized word embeddings for mid-resource languages. *Proceedings of the 58th annual meeting of the association for computational linguistics*, Online. Stroudsburg, PA, USA, 2020. URL: https://doi.org/10.18653/v1/2020.acl-main.156 (date of access: 09.07.2026).

14. The refinedweb dataset for falcon LLM: outperforming curated corpora with web data, and web data only / G. Penedo et al. *arXiv*. 2023. URL: https://doi.org/10.48550/arXiv.2306.01116 (date of access: 09.07.2026).

15. The fineweb datasets: decanting the web for the finest text data at scale / L. Allal et al. *Advances in neural information processing systems 37*, Vancouver, BC, Canada, 10–15 December 2024. San Diego, California, USA, 2024. P. 30811–30849. URL: https://doi.org/10.52202/079017-0970 (date of access: 09.07.2026).

16. Datasets: a community library for natural language processing / Q. Lhoest et al. *arXiv*. 2021. URL: https://doi.org/10.48550/arXiv.2109.02846 (date of access: 09.07.2026).

17. Kudo T. Subword regularization: improving neural network translation models with multiple subword candidates. *Proceedings of the 56th annual*

meeting of the association for computational linguistics (volume 1: long papers), Melbourne, Australia. Stroudsburg, PA, USA, 2018. URL: https://doi.org/10.18653/v1/p18-1007 (date of access: 09.07.2026).

18. Lison P., Tiedemann J. OpenSubtitles2016: extracting large parallel corpora from movie and TV subtitles. Proceedings of *the Tenth International Conference on Language Resources and Evaluation* (LREC'16). 2016. P. 923–929. URL: https://aclanthology.org/L16-1147/ (date of access: 09.07.2026).

19. Koehn P. Europarl: a parallel corpus for statistical machine translation. Proceedings of *Machine Translation Summit X: Papers*. 2005. P. 79–86. URL: https://aclanthology.org/2005.mtsummit-papers.11/ (date of access: 09.07.2026).

20. Findings of the 2016 conference on machine translation / O. Bojar et al. *Proceedings of the first conference on machine translation: volume 2, shared task papers*, Berlin, Germany. Stroudsburg, PA, USA, 2016. URL: https://doi.org/10.18653/v1/w16-2301 (date of access: 09.07.2026).

21. Bird S. NLTK. *The COLING/ACL*, Sydney, Australia, 17–18 July 2006. Morristown, NJ, USA, 2006. URL: https://doi.org/10.3115/1225403.1225421 (date of access: 09.07.2026).

22. Gemini Team Google. Gemini: a family of highly capable multimodal models. *arXiv.* 2025. URL: https://doi.org/10.48550/arXiv.2312.11805 (date of access: 09.07.2026).

23. PyTorch: an imperative style, high-performance deep learning library / A. Paszke et al. *arXiv*. 2019. URL: https://doi.org/10.48550/arXiv.1912.01703 (date of access: 04.07.2026).

24. Language models are unsupervised multitask learners / A. Radford et al. OpenAI Blog. 2019. Vol. 1. No. 8. P. 9.

25. Transformers: state-of-the-art natural language processing / T. Wolf et al. *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, Online. Stroudsburg, PA, USA, 2020.

URL: https://doi.org/10.18653/v1/2020.emnlp-demos.6 (date of access: 09.07.2026).

26. Loshchilov I., Hutter F. Decoupled weight decay regularization. *arXiv*. 2019. URL: https://doi.org/10.48550/arXiv.1711.05101 (date of access: 09.07.2026).