Міністерство освіти і науки України

Львівський національний університет імені Івана Франка

Факультет електроніки та комп’ютерних технологій

Курсова робота

на тему:

«Лінгвістичний переклад за допомогою рекурентних нейронних мереж»

Виконав:

студент групи ФеІм-51

Туз Р. _____

Науковий керівник

доц. Рабик В.Г._____

Львів 2023

Анотація

курсової роботи студента 5-го курсу групи ФеІм-11с спеціальності 122 «Комп’ютерні науки» Туз Руслан Андрійовича на тему "Лінгвістичний переклад за допомогою рекурентних нейронних мереж".

Курсова робота присвячена розгляду та реалізації підходів до машинного перекладу за допомогою рекурентних нейронних мереж, а також, інструментів, що використовуються для їх побудови. Реалізовано приклад використання нейронних мереж для додатку "Neural Translator ".

Ключові слова: RNN, LSTM, рекурентні нейронні мережі, лінгвістичний переклад.

Стор. - 57, рис. - 28, табл. - 4, бібл. - 13.

Abstract

course work of a student of 5th course of the FeIm-11s group of specialty 122 “Computer science” Tuz Ruslan on the topic «Linguistic translation using recurrent neural networks».

Course work devoted to overview and implementation of machine translation using recurrent neural networks, and tools which used to build them. Implementation of neural network for application “Neural Translator”.

Keywords: RNN, LSTM, recurrent neural networks, linguistic neural networks.

P. - 57, fig. - 28, tabl. - 4, bibl. - 13.

ЗМІСТ

ВСТУП **4**

РОЗДІЛ 1. ІСТОРІЯ МАШИННОГО ПЕРЕКЛАДУ **5**

1.1. Історія машинного перекладу 5

1.2. Програми машинного перекладу 9

1.2.1. Перекладач Prompt *9*

1.2.2. Перекладач Google *10*

1.2.3. Перекладач DeepL *12*

2. РЕКУРЕНТНІ НЕЙРОННІ МЕРЕЖІ **14**

2.1. Основні типи рекурентних нейронних мереж 14

2.2. Нейронні мережі LSTM *22*

3. ОПИС ПРОГРАМНОГО ДОДАТКУ NEURAL TRANSLATOR **27**

3.1. Попередня обробка даних 27

3.2. Основні компоненти 28

3.2.1. Вхідний шар *28*

3.2.2. Архітектура Transformer *29*

3.2.3. Двонаправлений рекурентний шар *30*

3.3. Опис додатку Neural Translator та результати досліджень 31

3.3.1. Підготовка даних *32*

3.3.2. Компоненти для навчання та візуалізації результатів *32*

3.3.3. Результати навчання моделей *39*

ВИСНОВКИ **47**

СПИСОК ЛІТЕРАТУРИ **48**

ДОДАТОК 1. ЛІСТІНГ ДОДАТКУ «NEURAL TRANSLATOR» **50**

ВСТУП

Лінгвістичний переклад - це процес передачі значення, структури та інших лінгвістичних аспектів тексту з однієї мови на іншу. Це складна задача, яка вимагає розуміння обох мов, культурних контекстів, стилістичних особливостей і виразності.

Машинний переклад є однією з галузей машинного навчання, яка активно розвивається. Машинний переклад може використовуватись як сам по собі, так і як допоміжний інструмент для перекладу людиною.

Рекурентні нейронні мережі (RNN) є одним з найпотужніших інструментів у цій області, оскільки вони здатні зберігати контекст у процесі перекладу. Метою дослідження є дослідження ефективності та точності RNN в контексті перекладу.

Суттєвим привілеєм нейронних мереж над системами перекладу, заснованими на правилах, є більша гнучкість у виборі семантичних об'єктів для навчання. Для систем заснованих на правилах існує необхідність вручну позначати частини мови, інші граматичні правила, які властиві певній мові - очевидно, що граматичні норми однієї мови можуть суттєво розходитись з іншою.

Для досягнення поставленої мети в курсовій роботі будуть проведені експерименти з використанням рекурентних нейронних мереж. Будуть зібрані дані та підготовлені для тренування та тестування моделей. Оцінка результатів буде здійснюватися за допомогою метрик точності, що дозволять зробити висновки про ефективність RNN в контексті лінгвістичного перекладу.

РОЗДІЛ 1. ІСТОРІЯ МАШИННОГО ПЕРЕКЛАДУ

1.1. Історія машинного перекладу

Використовуючи комп'ютер під час перекладу розрізняють автоматизований переклад (Computer – Aided Translation / Computer – Assisted Translation, CAT), покликаний допомогти перекладачеві виконати свою роботу швидше, та машинний переклад (Machine Translation, MT), який визначають як процес перекладу деякого тексту однієї природної мови іншою, реалізований комп'ютером повністю або майже повністю, а у широкому сенсі машинний переклад (МП) трактують як напрям наукових досліджень, що перебуває на перетині лінгвістики, математики, кібернетики, маючи на меті побудову систем, що реалізують машинний переклад у вузькому сенсі [1].

Історія МП розпочалася ще у середині минулого століття, коли народилася ідея автоматичного перекладу текстів однієї мови іншою без допомоги людини. Незважаючи на те, що це завдання й досі не має задовільного вирішення, останнім часом у цій сфері досягнуто суттєвого прогресу, а сама постановка проблеми стала поштовхом для розвитку нових лінгвістичних теорій.

Уперше концепція МП була сформульована у 1947 р. Уорреном Вівером – директором відділення природничих наук Рокфеллерівського фонду, яку він (дещо пізніше) розвинув у своєму меморандумі, адресованому Фонду. Концепція базувалася на інформаційній теорії, успіхах у зламуванні кодів протягом другої світової війни та обговореннях універсальних та основних принципів мов. За кілька років після опублікування меморандуму розпочалися серйозні дослідження у багатьох університетах Сполучених Штатів. 7 січня 1954 у Нью-Йорку в головному офісі IBM було вперше проведено публічну демонстрацію системи машинного перекладу (МП). Про демонстрацію повідомили в газетах, тож подія отримала широкий розголос. Попри те, що сама система мала лише 250 слів та 49 перекладених на англійську російськомовних речень (головним чином у області хімії) і була доволі

примітивною, вона продемонструвала перспективи машинного перекладу, стимулювавши фінансування цього дослідження не тільки у США, а й у всьому світі. Експеримент було визнано успішним, що сповістило про початок ери вагомих капіталовкладень у дослідження машинного перекладу. Автори стверджували, що за кілька років машинний переклад буде повністю втілено в життя [2].

МП пройшов бурхливий період розвитку і зазнав значних змін, еволюціонувавши від максимально спрощених версій («лексиконних») до версій, «заглиблених» у зміст. Цей розвиток чітко простежується у почерговій появі систем МП.

Перше покоління систем, яке прийнято датувати до середини 1960-х рр., наближене за своєю технікою до послівного перекладу (word-for-word translation): до кожного слова або мовного звороту у вхідному тексті підбирається еквівалент у вихідному тексті, знайдений у словнику. Системи першого покоління не здатні вирішувати проблем багатозначності, не проводять жодного лінгвістичного аналізу, з огляду на що якість перекладу досить низька.

Наступний період розвитку систем МП (середина 1960–1970-х рр.) називають другим поколінням. Їх внутрішній устрій трохи складніший, ніж у першого покоління, такі системи МП базуються на морфологічних кореляціях між вхідною та вихідною мовами («морфологічні системи»), що є ефективними для організації перекладу в межах споріднених мов.

Структурно-граматичні системи МП, що ґрунтуються на синтаксичних кореляціях вхідної та вихідної мов – «синтаксичні системи». Ключовою процедурою цих систем є синтаксичний аналіз вхідної фрази із подальшою її трансформацією в структурно-синтаксичний каркас вихідної фрази.

Структурно-семантичні системи МП оперують глибинними структурами вхідного та вихідного контекстів. У таких системах передбачається багаторівневий аналіз та опрацювання мовного матеріалу.

Інтерактивні системи МП, у яких на різних стадіях перекладу залучається людина-перекладач.

Можна виокремити такі форми взаємодії перекладача та інтерактивних систем МП:

- постредагування (перекладач редагує перекладений машиною текст);
- попереднє редагування (перекладач редагує вхідний текст, пристосовуючи його для якіснішої обробки);
- інтерредагування (перекладач бере участь у роботі програми, вносячи необхідні корективи у процесі перекладу);
- змішані системи (із застосуванням попереднього і постредагування).

Початок 1990-х рр. став переломним моментом. По-перше, група з IBM опублікувала результати експериментів над системою (Candide), заснованих виключно на статистичних методах. По-друге, деякі японські групи почали використовувати методи, засновані на сукупності прикладів перекладу, тобто використовуючи підхід, який зараз називається перекладом на основі прикладів. В обох підходах відмінною рисою було те, що при аналізі текстів і підборі лексичних еквівалентів не використовуються синтаксичні чи семантичні правила; обидва підходи відрізнялися від попередніх методів на основі правил у використанні великих текстових корпусів [2].

На цьому етапі розвитку наявні системи МП можна розділити на дві основні групи: системи, засновані на граматичних правилах, та статистичні системи МП.

Системи, засновані на граматичних правилах (Rule-Based Machine Translation, RBMT), виконують аналіз тексту і переклад за вбудованими словниками і набором правил. Ефективність роботи таких систем безпосередньо залежить від якості двомовних словників і точності формулювання правил. У системах, заснованих на граматичних правилах, можна виокремити такі дві основні підгрупи, як трансферні та інтерлінгви. Більш широкого застосування отримали системи трансферного типу, які

працюють за таким принципом: аналізується морфологічний, семантико-синтаксичний і лексичний рівні вхідної мови, потім формується семантико-синтаксична форма вихідної мови, після чого відбувається «трансфер» речення відповідно до структурних вимог вихідної мови. Робота систем-інтерлінгв базується на теорії універсальної метамови, тобто вхідний текст трансформується у смисл метамовою, а потім синтезується у вихідний текст [3].

Статистичні системи (Statistical Machine Translation, SMT) здатні до навчання. Створення таких систем МП базується на теорії інформації і характеризується використанням паралельних корпусів (корпусів текстів вхідної і вихідної мов). За результатами аналізу корпусів складається статистика щодо слововживання, міжмовних відповідностей тощо. Таким чином, програма визначає більш імовірний результат перекладу, ґрунтуючись на даних, отриманих під час оброблення корпусів. Наприкінці програма оцінює якість отриманого перекладу, керуючись нормами і правилами вихідної мови [3].

У 2014 році виходить стаття з коротким описом ідеї застосування нейромереж глибокого навчання до машинного перекладу. В Інтернеті її взагалі ніхто не помітив, а в лабораторіях Google почали активно працювати. Через два роки, в листопаді 2016 року, в блозі Google з'являється анонс, який і перевернув наступну сторінку машинного перекладу.

Якщо представити вихідний текст як набір таких же характерних властивостей, тобто закодувати його так, щоб потім інша нейромережа - декодер, розшифрувала їх назад в текст, але вже на іншій мові. Декодер спеціально навчається знати тільки одну мову. Він і гадки не має звідки характеристики взялися, але вміє висловити їх, скажімо, на іспанській мові.

Перша нейромережа вміє тільки кодувати речення в набір цифр-характеристик, а друга тільки декодувати їх назад в текст. Обидві не знають про існування одна про одну, кожна знає тільки свою мову. Текст, мова і музика завжди послідовні. Для їх обробки краще підходять рекурентні нейромережі

(RNN), оскільки вони пам'ятають попередній результат. В цьому випадку це попередні слова в реченні.

Рекурентні мережі зараз застосовують багато де: розпізнавання мови в Siri (парсування послідовності звуків, де кожен залежить від попереднього), підказки тексту за допомогою клавіатури (запам'ятовуються попередні і вгадуються наступні), генерація музики, навіть чатботи.

Спочатку дослідники використовували звичайні рекурентні мережі, потім перейшли на двохскеровані - перекладач враховував не тільки слова до, а й після потрібного слова. Так стало значно ефективніше. Потім взагалі почали використовувати багатошарові рекурентні мережі з LSTM-комірками (Long Short Term Memory) для довгого зберігання контексту перекладу [4].

1.2. Програми машинного перекладу

1.2.1. Перекладач Prompt

Перекладач PROMT - програмне забезпечення для автоматизованого перекладу текстів. Найбільш популярними є переклади з української мови російською та навпаки.

У 2016 році програма була побудована на власній системі «PROMT Hybrid». Система «PROMT Hybrid» базується на трьох компонентах: модулі RBMT, постпроцесорі RBMT і модулі статистичного постредагування (SPE). Модуль RBMT перекладає вихідний текст і виводить складну структуру, що містить переклад і його лінгвістичні особливості (морфологічну та синтаксичну інформацію, витягнуті іменовані сутності тощо). Постпроцесор RBMT генерує XML на основі вихідних даних модуля RBMT. Нарешті, XML передається в модуль SPE, який генерує вихідний переклад. Модуль SPE в основному є системою SMT, побудованою на паралельному корпусі перекладів RBMT та їхнім людським відповідникам. Техніка SPE дозволяє обробляти систематичні помилки RBMT, з якими важко впоратися алгоритмічно та швидко й ефективно адаптувати систему перекладу до конкретної області [5].

На конференції Sixth Conference on Machine Translation у 2021 році компанія представила напрацювання з використанням нейронної мережі. Системи SmartND (Smart Neural Dictionary) працюють на основі технології PROMT RBMT. Технологія не потребує спеціального попереднього навчання чи тонкого налаштування. Весь процес можна розділити на три етапи: створення словника, термінологічний пошук і модифікація вихідних даних.

Спочатку створюється словник PROMT у спеціальному форматі. Словник оптимізований для швидкодії та містить інформацію для кожної лексеми разом із повною флективною парадигмою. Якщо термін присутній у будь-якому з існуючих словників, інформація копіюється звідти. В інших випадках програма намагаємося вгадати слово і можливу парадигму на основі того, як закінчується термін. Якщо термін має кілька перекладів, то або вибирається один або повністю опускається термін. Також видаляються будь-які загальні терміни, які, ймовірно, будуть правильно перекладені моделями NMT. Це гарантує, що SmartND не заважатиме ідеально правильному виходу NMT.

Процес перекладу організований таким чином. Якщо термін присутній у вхідному тексті, програма шукає у вихідних даних NMT очікуваний переклад терміну. Якщо цей переклад відсутній у вихідних даних NMT, системи RBMT аналізують його та визначають граматичну інформацію (регістр і число) слова, яке модель NMT використовувала для перекладу терміну. Потім це слово замінюється на правильний переклад, узятий зі словника RBMT, використовуючи той самий регістр і число. Весь процес залежить від моделі NMT, яка забезпечує якісне вирівнювання на рівні слів [6].

1.2.2. Перекладач Google

У вересні 2016 року дослідницька група Google оголосила про розробку системи перекладу GNMT, і до листопада Google Перекладач почав використовувати нейронний машинний переклад (NMT) замість колишніх

статистичних методів (SMT), які використовувалися з жовтня 2007 року всередині власної закритої SMT-системи.

Система NMT всередині Google Перекладача використовує велику штучну нейронну мережу, придатну для глибинного навчання. Вивчаючи мільйони прикладів, GNMT покращує якість перекладу, використовуючи ширший контекст для виведення найякіснішого перекладу. Потім результат перебудовується й пристосовується для відповідності граматиці людської мови.

У цій реалізації рекурентні мережі є RNN з довгою короткочасною пам'яттю (LSTM). Їхні LSTM RNN мають 8 шарів із залишковими з'єднаннями між шарами для сприяння градієнтному потоку (рис. 1.1).

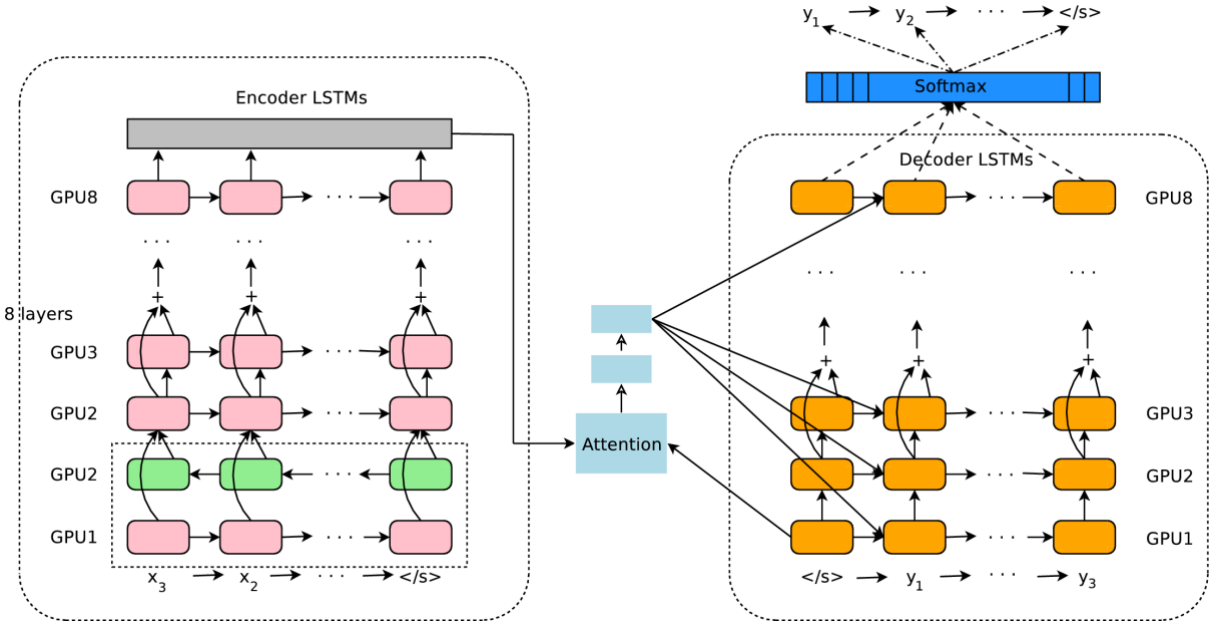

Рис. 1.1. Модель архітектури GNMT

Зліва (рис. 1.1) мережа кодера, праворуч мережа декодера, посередині модуль уваги. Нижній рівень кодера є двонаправленим: рожеві вузли збирають інформацію зліва направо, тоді як зелені вузли збирають інформацію справа наліво. Інші шари кодера є односпрямованими. Залишкові з'єднання починаються з третього знизу рівня в кодері та декодері. Для прискорення навчання модель розділена на кілька GPU. У цій структурі є 8 шарів LSTM кодера (1 двонаправлений шар і 7 односпрямованих шарів) і 8 шарів декодера. За допомогою цього параметра одна репліка моделі розділена на 8 сторін і

розміщується на 8 різних графічних процесорах, які зазвичай належать одній головній машині. Під час навчання спочатку паралельно обчислюються нижні шари двонаправленого кодера. Після того, як обидва завершать, односпрямовані шари кодера можуть почати обчислення, кожен на окремому GPU. Щоб зберегти якомога більший паралелізм під час роботи рівнів декодера, використовується вихід нижнього рівня декодера лише для отримання повторюваного контексту уваги, який надсилається безпосередньо на всі інші рівні декодера. Рівень softmax також розділений і розміщений на кількох графічних процесорах. Залежно від розміру вихідного словника або вони запускаються на тих самих графічних процесорах, що й мережі кодера та декодера, або на окремому наборі виділених графічних процесорів [7].

1.2.3. Перекладач DeepL

Обидва Google Translate та DeepL є NMT системами, які використовуються для перекладу текстів з одної мови на іншу. Але системи є різними в тому, які технології вони використовують та треновані на різних даних. Більш докладніше, DeepL використовує згорткові нейронні мережі CNN (convolutional neural network) та тренована на базі даних Linguee bilingual corpora database, коли Google Translate використовує RNN та тренована на великій кількості цифрових ресурсів на багатьох мовах.

Для оцінки якості перекладу існує декілька автоматичних метрик, найбільш популярними є BLEU та METEOR. Основна одиниця для оцінки в них - речення, алгоритм спочатку проводить вирівнювання тексту між двома реченням, рядком еталонного перекладу та рядком вхідного тексту для оцінювання.

Відповідно до метрик, DeepL показав кращий результат порівняно з Google Translate (рис. 1.2). Зокрема, переклад медичного тексту отримав найкращий результат в обох метриках а літературних текст найнижчий. Однак Google Translate отримав кращий результат в літературних та маркетингових

текстах відповідно до метрики BLEU, та в медичних текстах відповідно до метрики METEOR [8].

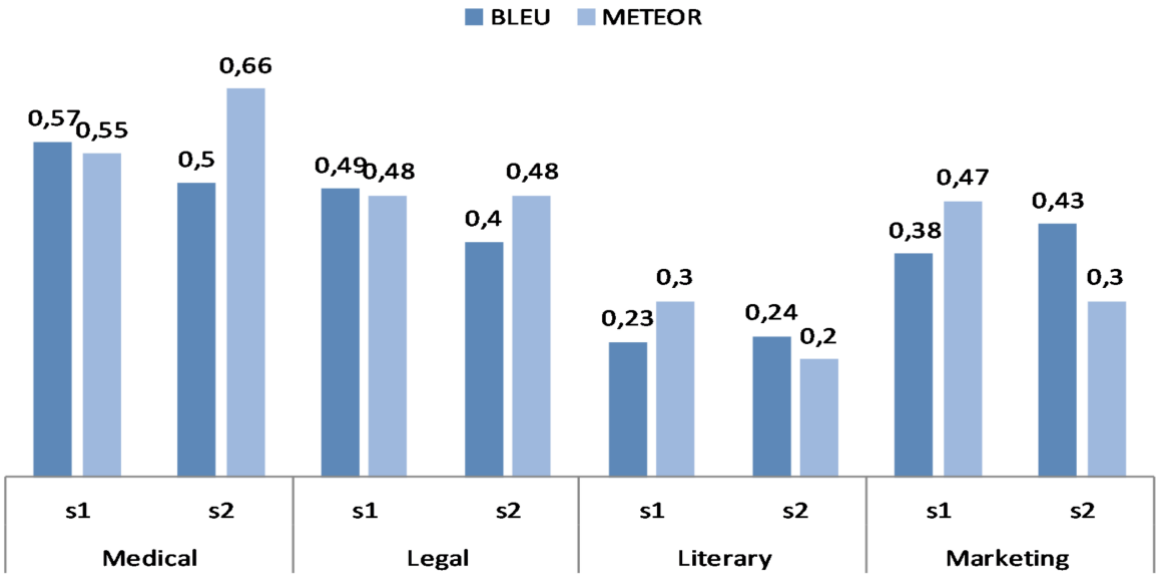

Рис. 2. Автоматичні метрики BLEU та METEOR для оцінки якості перекладу DeepL (s1) та Google Translate (s2)

2. РЕКУРЕНТНІ НЕЙРОННІ МЕРЕЖІ

Нейронні мережі є одним із найпопулярніших алгоритмів машинного навчання, який перевершує інші алгоритми як за точністю, так і за швидкістю. Тому важливо мати глибоке розуміння того, що таке нейронна мережа, яка її архітектура, які основні типи нейронних мереж та їх основні застосування.

Багатошарові нейронні мережі прямого поширення (MLNN) використовується найчастіше для задач регресії та класифікації. В цих мережах інформація надходить тільки в прямому напрямку: від вхідних нейронів через приховані шари до вихідних нейронів. У мережі немає циклів, обернених зв'язків (пам'яті). У MLNN рішення базуються на поточних вхідних даних. Мережа не запам'ятовує попередні дані, і в них немає наступних даних.

Нейронні мережі можна розділити на динамічні та статичні. Статичні нейронні мережі обчислюють вихідні дані безпосередньо з вхідних даних в ті ж самі моменти часу. Багатошарові нейронні мережі належать до статичних мереж.

В динамічних нейронних мережах вихід залежить від поточного входу в мережу і попередніх входів, виходів і / або прихованих станів мережі.

2.1. Основні типи рекурентних нейронних мереж

На відміну від багатошарових нейронних мереж прямого поширення, рекурентні нейронні мережі (RNN - Recurrent Neural Network) [] містять приховані стани, які розподілені в часі. Це дозволяє їм зберігати великий об'єм даних про минуле. Рекурентні нейронні мережі є прикладом динамічних мереж.

Дві прості RNN зображені на рис. 2.1. З них бачимо, що обернені зв'язки в цих мережах дозволяють запам'ятати попередні вхідні дані у внутрішньому стані мережі і тим самим впливати на вихідні дані мережі.

На кожному часовому кроці мережа обробляє вхідні дані x_t, оновлює свій прихований шар з допомогою функції активації h_t, використовує його для прогнозування свого виходу y_t. Ці кроки аналогічні як і для мереж MLNN. Але

значення, які зберігаються в блоці затримки (рис. 2.1), повертаються в приховані шари як додаткові входи.

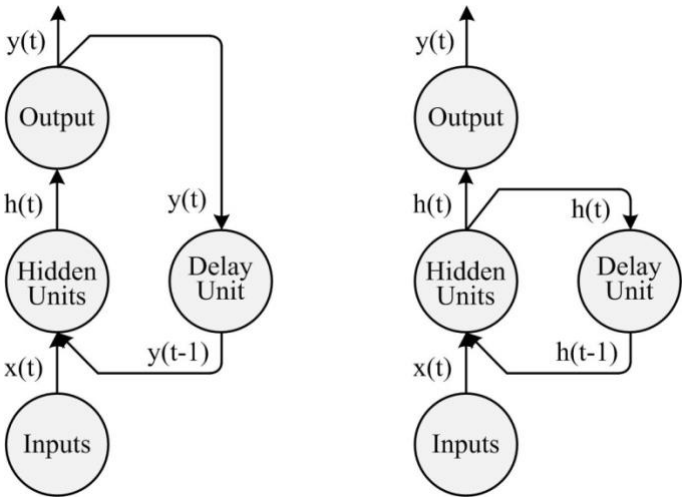

Рис. 2.1. Дві структури RNN

Блоки затримки (рис. 2.1) дозволяють мережі мати короткочасну пам'ять, так як вони зберігають значення активації прихованого шару і / або вихідні значення попереднього часового кроку. Ці значення повертаються блоками затримки на наступному часовому кроці. Саме ця функція короткочасної пам’яті дозволяє RNN навчатися і узагальнювати послідовності вхідних даних.

Рекурентна формула на кожному часовому кроці для обробки вхідних даних x має вигляд:

$$h_t = f_w(h_{t-1}, x_t),$$

де h_t – стан прихованого шару для часового кроку t; f_w – деяка функція активації з параметрами w; h_{t-1} - стан прихованого шару для часового кроку t-1; x_t – вхідний вектор для часового кроку t.

Рекурентна нейронна мережа, зображена зліва на рис. 2.1, отримала назву нейронної мережі Елмана або Vanilla RNN. Її робота описується математичними виразами:

$$h_t = tanh(w_{hh}h_{t-1} + w_{xh}x_t + b_h), \quad (2.1)$$

$$y_t = w_{hy} h_t + b_y. \tag{2.2}$$

Рекурентна нейронна мережа, зображена справа на рис. 2.1, отримала назву нейронної мережі Джордана []. Наступні вирази описують її роботу:

$$h_t = tanh\left(w_{hy} y_{t-1} + w_{xh} x_t + b_h\right), \tag{2.3}$$

$$y_t = f\left(w_{hy} h_t + b_y\right). \tag{2.4}$$

В виразах (2.1), …, (2.4) використовуються наступні матриці вагових коефіцієнтів:

- w_{xh} - використовується для всіх взаємозв'язків між x_t та h_t;
- w_{hh} - використовується для всіх взаємозв'язків між h_{t-1} та h_t;
- w_{hy} - використовується для всіх взаємозв'язків між h_t та y_t.

Також в цих виразах використовується два вектори зміщень b_h, b_y:

- b_h - використовується при обчисленні h_t;
- b_y - використовується при обчисленні y_t.

В виразах (2.1), (2.3) як функція активації використовується функція тангенса гіперболічного.

З рис. 2.2 бачимо, що RNN складається з декількох копій однієї і тієї ж мережі, кожна з яких передає повідомлення наступній. Вона має однакові параметри на всіх часових кроках, так як виконує ту саму задачу на кожному кроці, тільки з різними вхідними даними. Це значно зменшує загальну кількість параметрів, необхідних для навчання, порівняно з традиційною мережею MLNN, яка використовує різний набір вагових коефіцієнтів і зміщень для кожного шару.

Обернене поширення в часі (Backpropagation Through Time – BPTT). Основна ідея RNN – це розгорнути їх в часі. Рис. 2.2 ілюструє розгорнуту версію мережі Елмана. Під розгортанням мається на увазі, що ми отримуємо мережу для всіх часових кроків, що розглядаються. Розгортання просто розгортає цикл в часі, щоб отримати нейронну мережу прямого поширення.

Розгорнута RNN – це фактично мережа MLNN, в якій кожний шар відповідає часовому кроку в вихідній RNN.

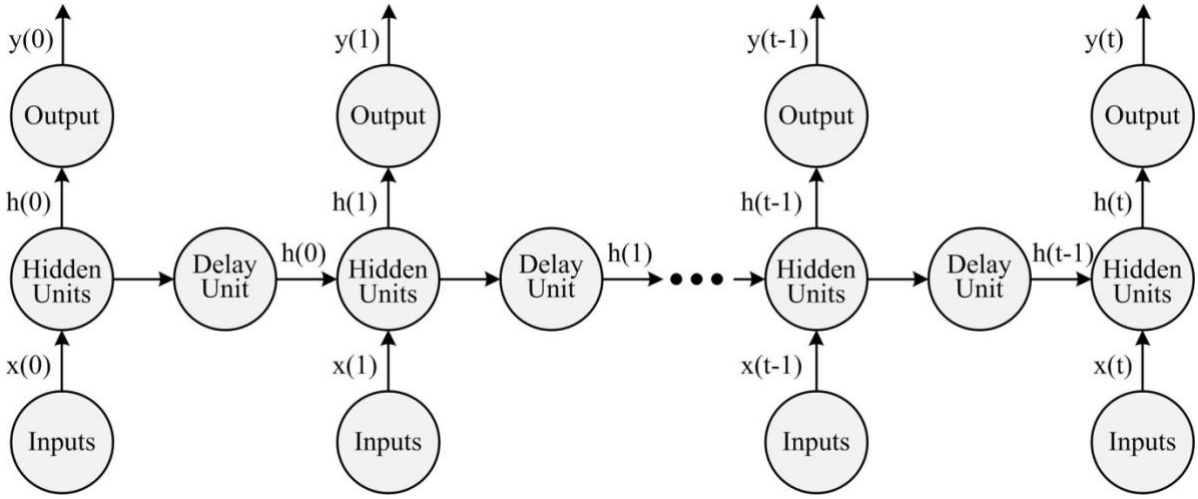

Рис. 2.2. Вигляд розгорнутої в часі RNN

Отримана мережа прямого поширення може бути навчена за допомогою алгоритму оберненого поширення. Розгорнута мережа (рис. 2.2), що використовується під час прямого проходу, розглядається як одна велика мережа прямого поширення, яка приймає весь часовий ряд в якості вхідних даних. Оновлення вагових коефіцієнтів виконується для кожної копії в розгорнутій мережі, потім підсумовуються (або усереднюються) і застосовуються до ваг мережі.

Обчислення похідних цільової функції по вагових коефіцієнтах зводиться до обчислення похідних у кожному шарі мережі прямого поширення. Після навчання мережі, мережа прямого поширення "згортається". Отримуємо вихідну RNN.

Розглянемо детальніше навчання RNN, зображеної на рис. 2.3, на основі алгоритму оберненого поширення в часі. Запишемо цільову функцію для RNN у вигляді:

$$E(y, \hat{y}) = 0.5 \sum_{t=1}^{T} (y_t - \hat{y}_t)^2, \tag{2.5}$$

де $\hat{y}_t$ – бажане значення на виході нейронної мережі в момент часу t.

Ваговий коефіцієнт w_{hy} є однаковим для всієї часової послідовності. Отже, ми можемо диференціювати його на кожному часовому кроці та просумувати всі отримані похідні.

$$\frac{\partial E}{\partial w_{hy}} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial w_{hy}} = \sum_{t=1}^{T} (y_t - \hat{y}_t) h_t, \tag{2.6}$$

де з виразу (2.2) отримуємо, що $\frac{\partial y_t}{\partial w_{hy}} = h_t$.

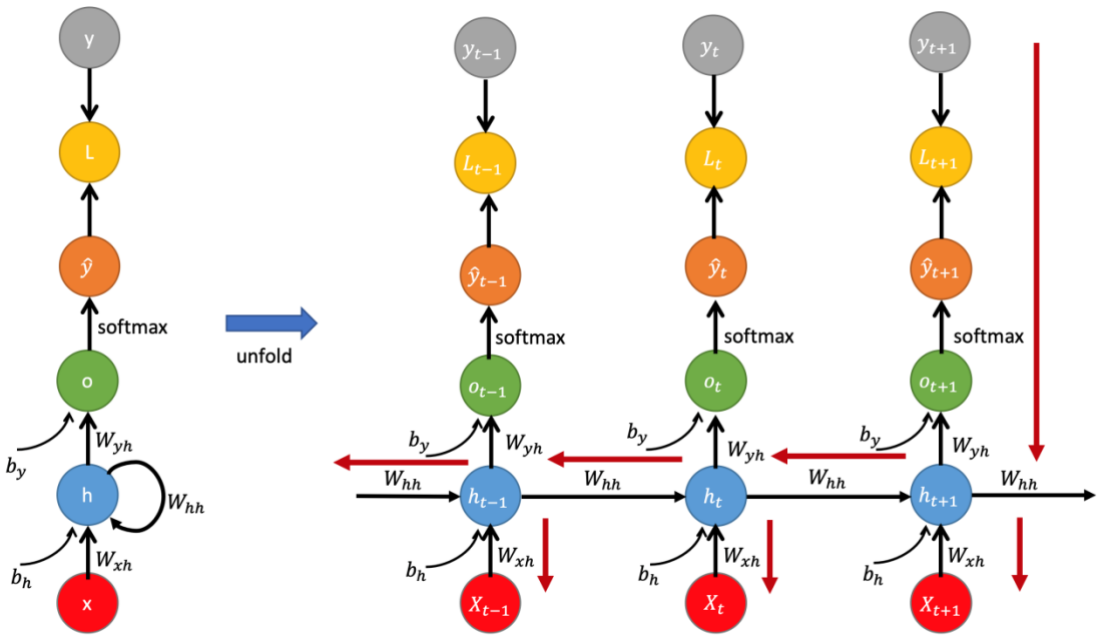

Рис. 2.3. Вигляд RNN розгорнутої на декілька часових кроків

Аналогічно можна отримати похідну по зміщенню нейрона:

$$\frac{\partial E}{\partial b_y} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial b_y} = \sum_{t=1}^{T} (y_t - \hat{y}_t). \tag{2.7}$$

Позначимо через E_{t+1} значення цільової функції для часового кроку t+1. Отримаємо похідну відносно w_{hh} для часового кроку t+1:

$$\frac{\partial E_{t+1}}{\partial w_{hh}} = \frac{\partial E_{t+1}}{\partial y_{t+1}} \frac{\partial y_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial w_{hh}}.$$

Але прихований стан h_{t+1} залежить також частково від h_t згідно виразу (2.1). Отже для часового кроку t можемо додатково отримати часткову похідну відносно w_{hh}.

$$\frac{\partial E_{t+1}}{\partial w_{hh}} = \frac{\partial E_{t+1}}{\partial y_{t+1}} \frac{\partial y_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} \frac{\partial h_t}{\partial w_{hh}}. \quad (2.8)$$

На часовому кроці t+1 можемо обчислити градієнт і далі використати обернене поширення в часі від t+1 до 1, щоб обчислити загальну похідну відносно w_{hh}:

$$\frac{\partial E_{t+1}}{\partial w_{hh}} = \sum_{k=1}^{t+1} \frac{\partial E_{t+1}}{\partial y_{t+1}} \frac{\partial y_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_k} \frac{\partial h_k}{\partial w_{hh}}. \quad (2.9)$$

Похідна $\frac{\partial h_{t+1}}{\partial h_k}$ згідно наступного правила: $\frac{\partial h_3}{\partial h_1} = \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial h_1}$.

Так як обчислюється похідна вектор-функції по вектору, то результатом є матриця Якобі усі елементи якої є похідними по точкам. Перепишемо вираз (2.9) у вигляді:

$$\frac{\partial E_{t+1}}{\partial w_{hh}} = \sum_{k=1}^{t+1} \frac{\partial E_{t+1}}{\partial y_{t+1}} \frac{\partial y_{t+1}}{\partial h_{t+1}} \left(\prod_{j=k}^{t} \frac{\partial h_{j+1}}{\partial h_j} \right) \frac{\partial h_k}{\partial w_{hh}}, \quad (2.10)$$

де

$$\prod_{j=k}^{t} \frac{\partial h_{j+1}}{\partial h_j} = \frac{\partial h_{t+1}}{\partial h_k} = \frac{\partial h_{t+1}}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_{k+1}}{\partial h_k}. \quad (2.11)$$

Додаючи градієнти відносно w_{hh} по всіх часових кроках з оберненим поширенням, отримаємо наступний градієнт відносно w_{hh}:

$$\frac{\partial E}{\partial w_{hh}} = \sum_{t=1}^{T} \sum_{k=1}^{t+1} \frac{\partial E_{t+1}}{\partial y_{t+1}} \frac{\partial y_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_k} \frac{\partial h_k}{\partial w_{hh}}. \quad (2.12)$$

Для отримання градієнта відносно w_{xh} для часового кроку t+1 Потрібно виконати аналогічні перетворення. В результаті цього отримуємо:

$$\frac{\partial E}{\partial w_{xh}} = \sum_{t=1}^{T} \sum_{k=1}^{t+1} \frac{\partial E_{t+1}}{\partial y_{t+1}} \frac{\partial y_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_k} \frac{\partial h_k}{\partial w_{xh}} \quad (2.13)$$

Існує декілька типів використання RNN в ієрархічній обробці послідовних даних. В деяких випадках вимога виходу мережі – це прогнозування значення для кожного кроку. В інших випадках потрібне одне передбачення в кінці послідовності. Існує чотири типи рекурентних нейронних мереж, що найчастіше використовуються.

RNN "один вхід - один вихід" (рис. 2.4). Це класична архітектура нейронної мережі прямого поширення, з одним входом і одним виходом.

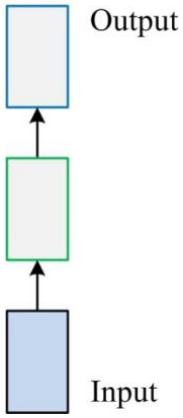

Рис. 2.4. Архітектура RNN "один вхід – один вихід"

RNN "один вхід - багато виходів" (рис. 2.5). Одне з використань такої RNN – це реалізація підпису до зображення. На вхід мережі подається одне зображення фіксованого розміру, а на її виході можуть бути слова або речення змінної довжини.

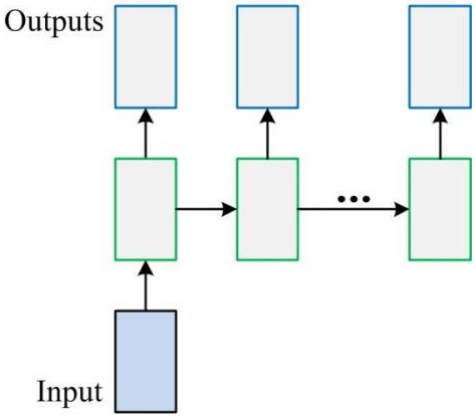

Рис. 2.5. Архітектура RNN "один вхід – багато виходів"

RNN "багато входів - один вихід" (рис. 2.6). Ця RNN приймає послідовність входів і генерує один вихідний сигнал. Прикладом використання

мережі є аналіз настрою. Певне речення, що подається на її вхід можна класифікувати як таке, що висловлює позитивні чи негативні настрої.

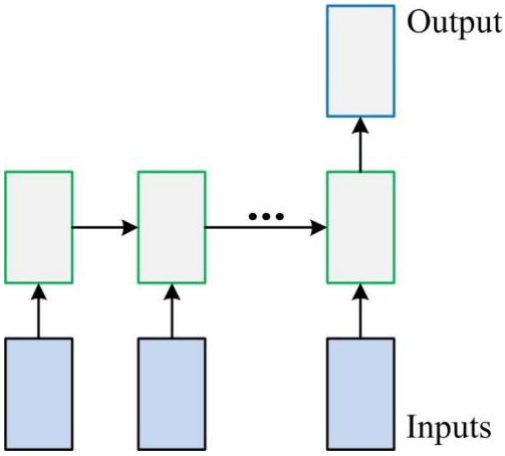

Рис. 2.6. Архітектура RNN "багато входів – один вихід"

RNN "багато входів - багато виходів" (рис. 2.7) використовується для перетворення послідовності вхідних даних у послідовність вихідних даних. Для синхронних RNN цього типу виконується прогноз для кожного кроку послідовності. В асинхронних мережах цього типу RNN спочатку обробляє всю послідовність вхідних даних, а потім починає створювати послідовність вихідних даних.

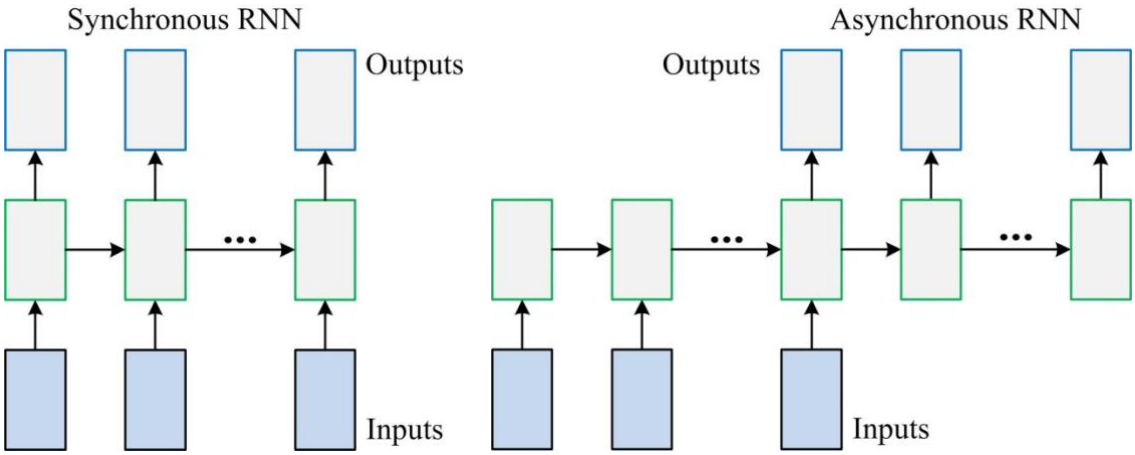

Рис. 2.7. Архітектури RNN "багато входів – багато виходів"

2.2. Нейронні мережі LSTM

Стандартні RNN страждають від проблеми "зникаючих" градієнтів. Градієнти несуть інформацію, яка використовується в цих мережах. Коли градієнт стає занадто малим, оновлення вагових коефіцієнтів мережі будуть незначними. Це ускладнює вивчення довгих послідовностей даних з допомогою рекурентних нейронних мереж.

Інша проблема стандартних RNN – це проблема "розривного" градієнта. Якщо під час навчання нейронної мережі, градієнт має тенденцію до експоненціального зростання замість спадання, то такий градієнт називають "розривним". Ця проблема виникає, коли накопичуються великі градієнти похибок, що призводить до дуже великих оновлень вагових коефіцієнтів моделі нейронної мережі під час її навчання.

Для вирішення цих двох проблем, характерних для стандартних RNN, використовуються рекурентні нейронні мережі довгої короткочасної пам’яті (LSTM – Long Short-Term Memory). Саме мережі LSTM використовуються для вивчення довготривалих залежностей, запам’ятовуючи інформацію протягом тривалого часу.

Усі рекурентні нейронні мережі мають вигляд ланцюжка повторюваних модулів нейронної мережі. У стандартних RNN цей повторюваний модуль матиме дуже просту структуру, таку як один шар функції активації (рис. 2.11).

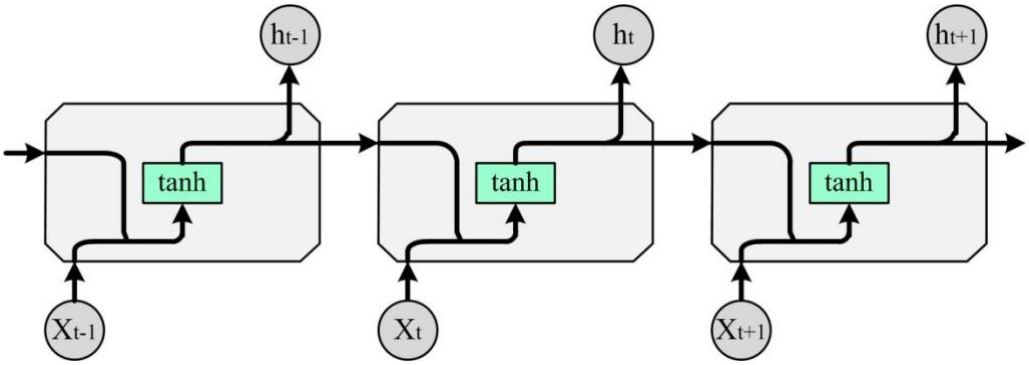

Рис. 2.8. Структура стандартної RNN

Мережі LSTM також мають таку ж ланцюгову структуру, але повторюваний модуль має іншу структуру (рис. 2.9). Замість одного рівня нейронної мережі існує чотири, які взаємодіють особливим чином.

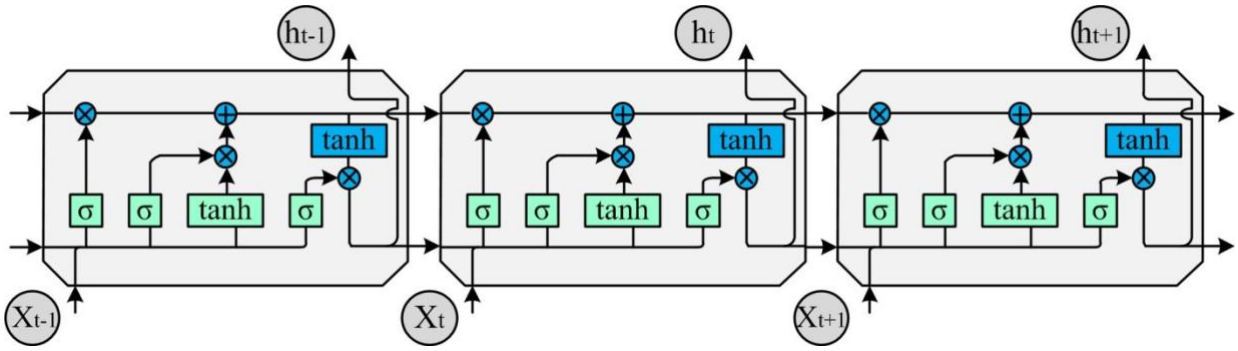

Рис. 2.9. Структура LSTM

Кожна комірка транспортує дані через блоки. Горизонтальна лінія, що проходить через вершину LSTM видаляє або додає інформацію про стан комірки за допомогою вентилів (шлюзів) (рис. 2.10). Ключовим в LSTM є стан комірки. Він проходить по всьому ланцюгу, лише з деякими незначними лінійними взаємодіями.

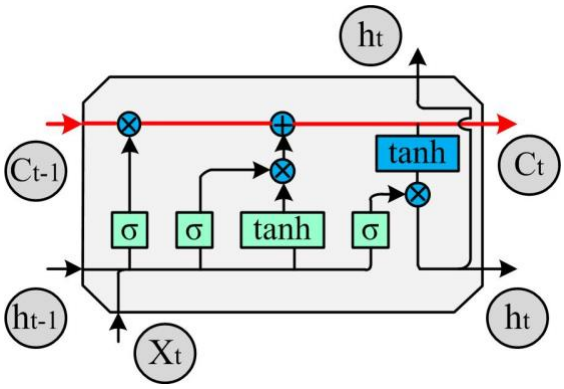

Рис. 2.10. Вигляд комірки з горизонтальною лінією, яка визначає її стан

Мережа LSTM може видаляти або додавати інформацію про стан комірки з допомогою вентилів, які отримали назву вентилів "забування". Вентилі складаються з шару сигмоїдальної нейронної мережі та операції множення:

$$f_t = sigmoid\left(\theta_{xf} x_t + \theta_{hf} h_{t-1} + b_f\right). \qquad (2.14)$$

На виході сигмоїдальної функції можемо отримати значення в межах від 0 до 1. Значення нуль означає "забути" (нічого не пропускати), тоді як значення 1 означає "пам’ятати" (пропускати все). В вентиль поступають сигнали h_{t-1} і x_t та вони видають число від 0 до 1 для кожного значення в стані комірки c_{t-1} (рис. 2.11). Мережа LSTM має три таких вентилі для захисту та контролю стану комірки.

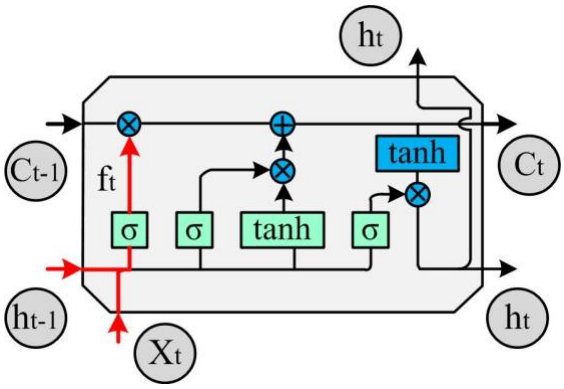

Рис. 2.11. Структура вентиля "забування"

Ми розглянули перший крок у мережі LSTM, який вирішує, які дані викинути зі стану комірки або зберегти.

На наступному кроці вирішується, яка нова інформація зберігатиметься в стані комірки. Це виконують вхідні вентилі, структура яких зображена на рис. 2.12. Вони складаються з сигмоїдального шару, який саме вирішує, які значення будуть оновлюватися та шару з функцією активації tanh. Саме в ньому створюється вектор нових значень c_t, які можна додати до стану. На наступному кроці вони об’єднуються, для оновлення стану.

Робота вхідного вентиля описується наступними виразами:

$$i_t = sigmoid(\theta_{xi}x_t + \theta_{hi}h_{t-1} + b_i). \quad (2.15)$$

$$\tilde{c}_t = tanh\left(\theta_{xg}x_t + \theta_{hg}h_{t-1} + b_g\right). \quad (2.16)$$

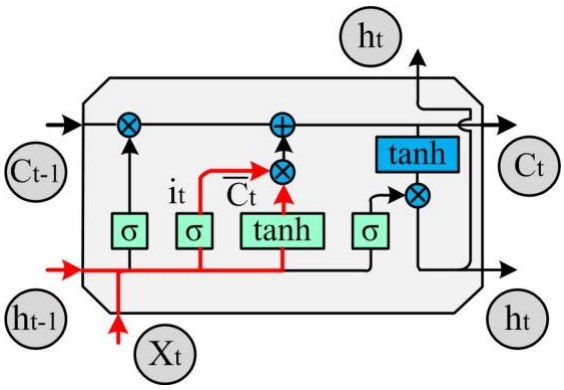

Рис. 2.12. Структура вхідного вентиля

На наступному кроці потрібно оновити стан комірки c_{t-1} в стан c_t. Для цього потрібно спочатку перемножити попередній стан комірки c_{t-1} на f_t, не враховуючи дані, які вирішили раніше забути. Після цього додаємо $i_t \cdot \tilde{c}_t$ і отримуємо нові значення, масштабовані на величину, на яку вирішили оновлювати значення стану (рис. 2.13):

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t. \tag{2.17}$$

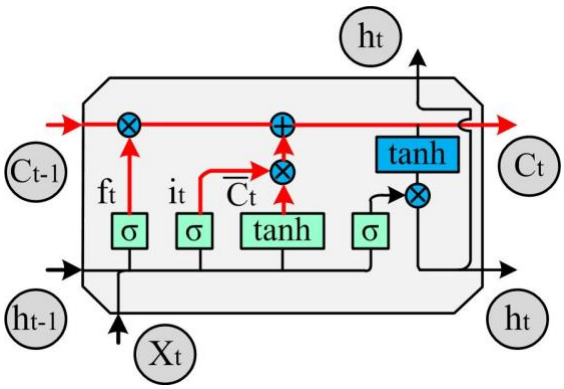

Рис. 2.13. Структура оновлення стану комірки

На наступному кроці вирішується, що буде виводитися через вихідний вентиль. Величина, що виводитиметься, залежить від стану комірки. Сигмоїдальний шар вирішує, яка частина стану комірки буде виводитися. Потім застосуємо функцію тангенса гіперболічного до стану комірки. Це

нормалізує вихід функції в межах від -1 до 1. Після цього перемножимо вихід функції *tanh* на вихід сигмоїдального шару (рис. 2.17):

$$\sigma_t = sigmoid(\theta_{xo}x_t + \theta_{ho}h_{t-1} + b_o), \tag{2.18}$$

$$h_t = o_t \cdot tanh(c_t). \tag{2.19}$$

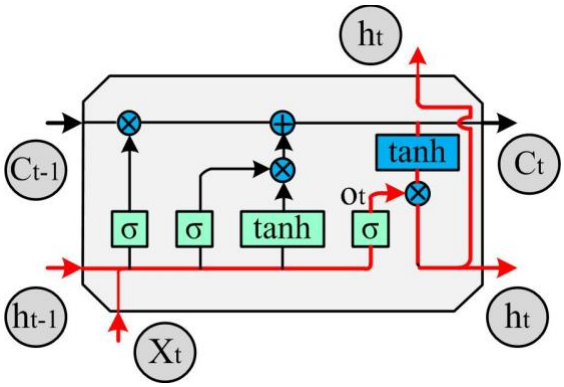

Рис. 2.14. Структура вихідних воріт комірки

3. ОПИС ПРОГРАМНОГО ДОДАТКУ NEURAL TRANSLATOR

3.1. Попередня обробка даних

Для тренування мережі потрібен великий об’єм текстових даних. Зрозуміло, що ці дані потрібно попередньо перетворити в потрібний для навчання вигляд. Перш за все потрібно застосувати певні техніки для очищення та трансформації тексту. При реалізації програмного додатку було застосовано наступні перетворення:

1. Розділити завантажений файл на два текстових, англійською і українською мовою.
2. Вилучити пунктуаційні знаки, крім ‘,’, ‘.’, ‘!’, ‘?’.
3. Відділити пунктуаційні знаки від слів пробілом.
4. Привести текст до нижнього регістру.
5. Вибрати рядки тексту заданої довжини.

Наступним кроком є розбиття речення на окремі лексеми. Щоб передбачити текст нейронною мережею, для початку дані мають бути перетворені в зрозумілий їй вигляд. Так як, нейронна мережа – це послідовність множень і додавань, вхідні дані мають бути цифрами. Тому, потрібно перетворити окремі слова в числа. Цей процес називають токенізацією.

Токенізація може бути як словесна, так і символьна. В курсовій роботі була використана словесна. Сам процес пролягає в тому, що кожному слову присвоюється унікальний ідентифікатор, відповідно речення перетворюється в набір чисел, які в подальшому подаються на вхід нейронної мережі. Для цього було використано функцію Tokenizer з бібліотеки Keras. Знаки пунктуації вилучені функцією за замовчуванням.

При групуванні послідовності слів разом, кожна послідовність має бути однакової довжини. Оскільки речення різні в розмірах, потрібно додати відступи в кінці послідовності, щоб зробити їх однаковими по довжині. Для цього було використано функцію pad_sequences з бібліотеки Keras, яка додає «нулі» в кінці послідовності (рис. 3.1).

```
[18 19  3 20 21]
[18 19  3 20 21  0  0  0  0  0]
```

Рис. 3.1. Послідовність токенів до додавання відступу та після.

3.2. Основні компоненти

3.2.1. Вхідний шар

Після токенізації вектори речень доповнюють до максимального речення для об'єднання в батч – 2+ вимірні матриці. Зазвичай, за перший вимір відповідає кількість екземплярів. За другий вимір відповідає зафіксована довжина речень. Кожна лексема визначається числом, що не є правильним підходом, оскільки допускає штучно створену подібність між словами, яким у процесі токенізації поставили у відповідність близькі числові значення.

Щоб цього уникнути використовують кілька варіантів. Перший – one-hot encoding []. Для кожного токену будується вектор розмірності словника, заповнений нулями на всіх позиціях, крім індексу самого токену. Проте він не підходить, коли за токен вважається ціле слово або його частини через велику розмірність таких векторів, яка катастрофічно впливає на швидкість обчислень.

Embedding layer – це шар tf.keras.layers, який перетворює числові послідовності в вектори (рис. 3.2). Це вектори з числовими даними певної розмірності, набагато меншої за розмір словника, значення яких можуть вираховуватись у процесі тренування нейронної мережі або заздалегідь. Для Embedding гарною рисою вважається властивість зберігати семантичну відстань між векторами при зміні певної характеристики для обох векторів. Наприклад, відстань між словами «королева» та «жінка» в ідеалі має бути дуже близька до відстані між «королем» та «чоловіком». $|vqueen - vwoman| \approx |vking - vman|$. Тому як перший шар моделей було використано Embedding.

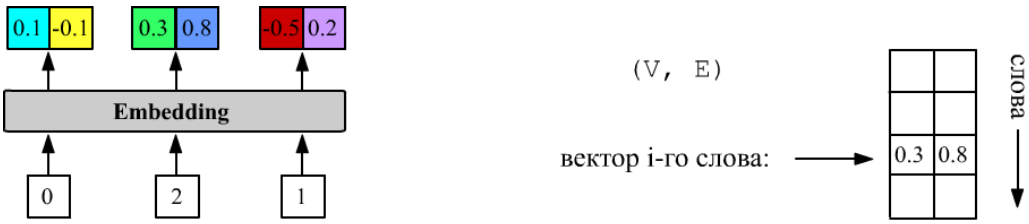

Рис. 3.2. Embedding шар.

3.2.2. Архітектура Transformer

Трансформер використовує кодувально-декодувальну архітектуру (рис. 3.3).

Кодер зчитує всю вхідну послідовність, обробляє її та фіксує деяку контекстну інформацію про вхідну послідовність у так званий контекстний вектор або вектор думки. Кожна комірка на рівні LSTM повертає прихований стан (h_i) і стан комірки (c_i). Останній прихований стан і стан комірки використовуються для ініціалізації декодера, який є другим компонентом цієї архітектури.

Декодер. Як і кодер, декодер зчитує всю цільову послідовність із зміщенням на один часовий крок разом із останнім прихованим станом і станом комірки кодера та передбачає наступне слово в цільовій послідовності.

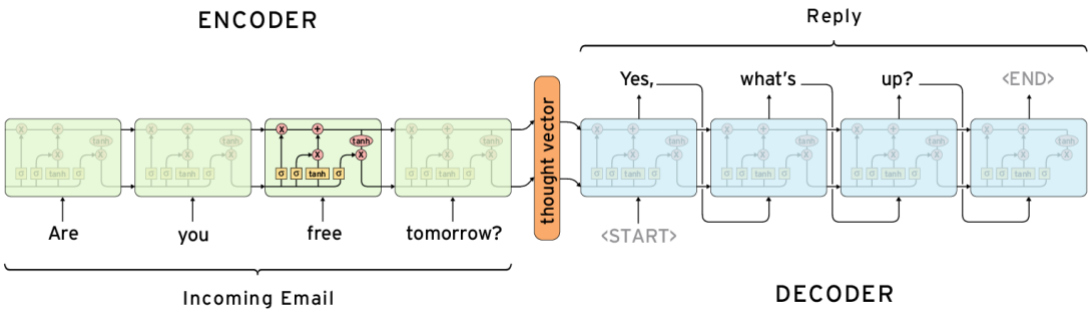

Рис. 3.3. Архітектура кодера - декодера.

Повнозв'язний (Dense) шар з розподіленням у часі (Time distributed) перетворює вихідні дані декодера LSTM безпосередньо в прогнозовані вихідні дані на кожному майбутньому часовому кроці (рис. 3.4) [12].

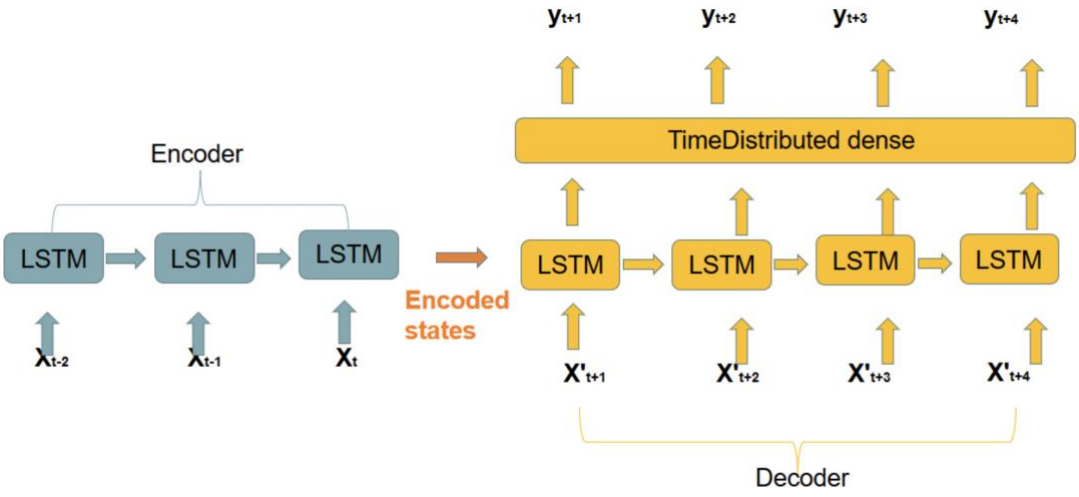

Рис. 3.4. Архітектура LSTM-LSTM моделі з повнозв'язним шаром.

Також для запобігання перенавчання додають шар Dropout, який виключає частину нейронів у нейромережі. Суть методу полягає в тому, що в процесі навчання із загальної мережі випадковим чином виділяється підмережа, для якої здійснюється навчання. Вибір нейронів для підмережі відбувається випадковим чином з ймовірністю, яка називається коефіцієнтом Dropout. Більш навчені нейрони отримують в мережі більшу вагу. Під час тестування Dropout не застосовується.

3.2.3. Двонаправлений рекурентний шар

Замість звичайної RNN/LSTM для енкодера та декодера часто використовують Bidirectional RNN (рис. 3.5) [13]. Її структура дозволяє мережі для кожного токена мати інформацію як про попередні до неї токени, так і про наступні, що більше походить на дійсність, де значення слова інколи не можна виявити лише з попередніх слів.

Така мережа складається з двох RNN, одна з яких рахує всі приховані стани, як раніше, у прямому напрямку, а інша - у зворотному. Після цього, щоб

отримати результуючий вектор на кожному кроці обидва приховані стани об’єднуються в один, який потім проходить через функцію активації.

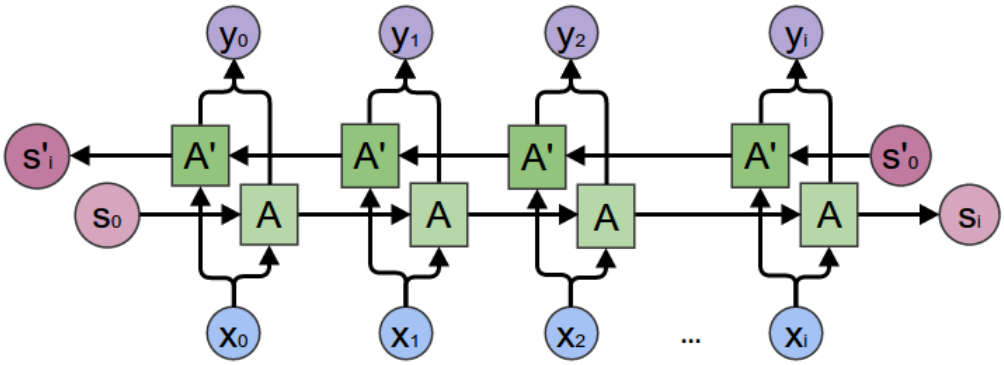

Рис. 3.5. Bidirectional RNN

3.3. Опис додатку Neural Translator та результати досліджень

Розробка, планування архітектури з параметрами та навчання нейронної мережі вимагає певного спеціального програмного забезпечення, а також вибору бібліотек котрі полегшують задачу реалізації мережі та її навчання. Тому для реалізації було використано програмні та апаратні засоби:

1. Мова програмування Python версії 3.10;
2. Операційна система MacOS Monterey 12.6;
3. Середовище розробки InteliJ IDEA;
4. Процесор Apple M1 Pro, який має 8 продуктивних ядер «Firestorm» та 2 ядра ефективності «Icestorm» та 16-ти ядерний графічний процесор.

Також для реалізації нейронної мережі та підготовки даних використовуються бібліотеки, котрі мають певні підготовлені функції, з допомогою яких спрощується процес розробки, та мають певні задані та перевірені стандартні параметри:

1. Keras – є у прямому сенсі надбудовою над бібліотекою TensorFlow, яка містить готові функції для побудови нейронної мережі блоками та явно задання параметрів.

2. Re – бібліотека, яка використовується для написання регулярних виразів у програмі мовою Python. Буде використовуватися для підготовки даних до навчання.
3. Pandas – це бібліотека для маніпулювання даними та їхнього аналізу.
4. Sclearn – ця бібліотека використовується для вирішення певних задач машинного навчання. Використаємо з неї дуже зручну функцію, яка ділить dataset на частини випадковим чином - тренувальну та перевірочну.
5. Natural Language Toolkit (NLTK) – використовується для перевірки якості перекладу з допомогою метрик.
6. Matplotlib та PrettyTable – використовуються для візуалізації результатів.

3.3.1. Підготовка даних

У якості даних використовувалась вибірка з сайту https://tatoeba.org. Однак ця вибірка була в CSV форматі, тому було реалізовано програму «dataset_converter.py», яка цю вибірку розділяє на два текстові файли однакового розміру, та попередньо очищає (див. розділ 3.2).

Для простоти навчання було вибрано речення з довжиною 8 слів. Очищена вибірка в результаті складається з 180 тисяч рядків, речення довжиною не більше 8 слів, 34 тисячі унікальних українських слів, та 13 тисяч англійських. Однак час навчання на такий вибірці займав близько п’яти годин.

3.3.2. Компоненти для навчання та візуалізації результатів

Програмою для навчання нейромережі є «learn.py». Також було написано підпрограми з функціями, які виконують певну роботу, а в «learn.py» будуть тільки викликатись, для зручності читання коду. Назви цих функцій вказують на їх функціональне призначення.

Програма «learn.py» працює наступним чином:

З допомогою підпрограми «helper.py» завантажуються вибірки

```
# Load English data
en_sentences = helper.load_data('data/vocab_eng.txt')
helper.print_words_count(en_sentences, "English")
# Load Ukrainian data
uk_sentences = helper.load_data('data/vocab_ukr.txt')
helper.print_words_count(uk_sentences, "Ukrainian")
print('Dataset length:', len(en_sentences))
```

Далі вибірка токенізується за допомогою підпрограми «tokenizer.py», яка в свою чергу використовує Tokenizer і pad_sequences з бібліотеки Keras та повертає токенізовані речення та сам токенізатор, який буде використовуватись пізніше.

```
preproc_en_seq, preproc_uk_seq, en_tokenizer, uk_tokenizer =
tokenizer.preprocess(en_sentences, uk_sentences)
```

Після цього створюємо модель навчання. Для цього використано «train_test_split» з бібліотека «scilearn» та підпрограму «model_builder.py» для побудови моделі нейромережі. У «train_test_split» передаємо токенізовані речення, визначаємо відсоток даних, які будуть використані тільки для валідації, і вказуємо, щоб речення були вибрані випадковим чином з всієї вибірки. Функція «train_test_split» повертає ті ж токенізовані речення але поділені на навчальну та валідаційну вибірку.

```
en_vocab_size = len(en_tokenizer.word_index) + 1
uk_vocab_size = len(uk_tokenizer.word_index) + 1

X_train, X_test, y_train, y_test = train_test_split(preproc_en_seq,
preproc_uk_seq, test_size=0.2, random_state=1)

model = model_builder.build(preproc_en_seq.shape, preproc_uk_seq.shape[1],
en_vocab_size, uk_vocab_size, 3)
```

Підпрограма «model_builder.py» має функцію «build», в якій будуються моделі. Вхідними параметрами функції є розміри вибірки (довжина речень) та розмір словника. Останнім параметром є номер моделі, який за замовчуванням

рівний числу 3. Відповідно можна вибрати модель для навчання. Для побудови шарів нейромережі було використано клас Sequential з бібліотеки Keras, який дозволяє додавати шар за шаром.

```
def build(input_shape, output_sequence_length, en_vocab_size, uk_vocab_size,
model_num=3):
    learning_rate = 0.003
    model = Sequential()

    if model_num == 1:
        # Build the layers Embedding
        model.add(Embedding(en_vocab_size, 512, input_length=input_shape[1],
input_shape=input_shape[1:]))
        # Encoder
        model.add(LSTM(512))
        model.add(RepeatVector(output_sequence_length))
        # Decoder
        model.add(LSTM(512, return_sequences=True))
        model.add(Dense(uk_vocab_size, activation='softmax'))
        model.compile(loss=sparse_categorical_crossentropy,
                      optimizer=Adam(learning_rate),
                      metrics=['accuracy'])

    if model_num == 2:
        # Build the layers Embedding
        model.add(Embedding(en_vocab_size, 512, input_length=input_shape[1],
input_shape=input_shape[1:]))
        # Encoder
        model.add(Bidirectional(LSTM(512)))
        model.add(RepeatVector(output_sequence_length))
        # Decoder
        model.add(Bidirectional(LSTM(512, return_sequences=True)))
        model.add(Dropout(0.3))
        model.add(TimeDistributed(Dense(uk_vocab_size, activation='softmax')))
        model.compile(loss=sparse_categorical_crossentropy,
                      optimizer=Adam(learning_rate),
                      metrics=['accuracy'])

    if model_num == 3:
        # Build the layers Embedding
```

```
        model.add(Embedding(en_vocab_size, 512, input_length=input_shape[1],
input_shape=input_shape[1:]))
        # Encoder
        model.add(Bidirectional(LSTM(512)))
        model.add(RepeatVector(output_sequence_length))
        # Decoder
        model.add(Bidirectional(LSTM(512, return_sequences=True)))
        model.add(TimeDistributed(Dense(512, activation='relu')))
        model.add(Dropout(0.5))
        model.add(TimeDistributed(Dense(uk_vocab_size, activation='softmax')))
        model.compile(loss=sparse_categorical_crossentropy,
                      optimizer=Adam(learning_rate),
                      metrics=['accuracy'])

    model.summary()
    return model
```

Embedding клас приймає розмір вхідного шару як розмір словника, вихідних вибрано 512, та розмірності вхідних даних.

LSTM - шар LSTM з 512 нейронами. Однак в декодері використано додатковий параметр «return_sequences=True» для виводу послідовності результатів, а не лише останнього вихідного стану. В одній з моделей також є Bidirectional LSTM.

RepeatVector шар - це шар у моделі нейромережі, який повторює вхідний тензор декілька разів для створення нового тензора з більшою розмірністю. У глибокому навчанні. RepeatVector шар часто використовується у контексті рекурентних нейромереж, таких як LSTM або GRU. Його головна мета - повторити вхідний тензор на кожному часовому кроці, щоб мати однаковий вхід для кожного елементу послідовності. Вихідний розмір цього шару задано розміром вихідної послідовності.

Dense дозволяє вирішити задачу застосування повнозв'язного шару з функцією активації ReLU до кожного елемента вхідної послідовності незалежно, що може покращити здатність моделі до аналізу та витягування важливих ознак з послідовностей даних.

Dense для перекладу текстів застосовує повнозв'язний шар з функцією активації softmax до кожного елемента вхідної послідовності для отримання ймовірностей перекладу на різні мови, поліпшуючи генерацію перекладів відповідно до контексту та попередніх перекладів.

TimeDistributed у Keras є обгорткою, яка дозволяє застосовувати один і той самий шар до кожного часового кроку в вхідній послідовності даних.

Dropout - шар регуляризації, який випадковим чином вимикає половину нейронів вхідного шару під час тренування для запобігання перенавчанню. Параметр 0.5 вказує на ймовірність вимкнення кожного нейрона.

Далі йде процес навчання. Для цього у моделі є функція «fit», в яку передається тренувальна вибірка, розмір батчу, кількість епох навчання та валідаційна вибірка. Для того, щоб в майбутньому використати модель, вона зберігається в файлі, а також зберігається історія навчання, використовуючи підпрограму «helper.py».

```
training_history = model.fit(X_train, y_train, batch_size=512, epochs=50,
validation_data=(X_test, y_test))
helper.save_model('en-uk-model1-w15k-l12-c3-e50', model, training_history)
```

І в кінці програми виводяться результати перекладу. В якості перевірки було взяти перших 5 речень з вибірки та 8 власноруч написаних речень, яких немає у вибірці. Для цього також було написано підпрограму «test_helper.py», яка виводить результати. Для роботи їй потрібно передати саму модель, вибірку яку хочемо перевірити, і токенізатори для перетворення отриманих від нейромережі послідовностей в текст.

```
test_helper.print_predictions(model, preproc_en_seq[:5], preproc_uk_seq[:5],
en_tokenizer, uk_tokenizer)

preproc_test_en_seq, preproc_test_uk_seq =
test_helper.get_test_phrases(en_tokenizer, uk_tokenizer, preproc_en_seq)
test_helper.print_predictions(model, preproc_test_en_seq, preproc_test_uk_seq,
en_tokenizer, uk_tokenizer)
```

Крім основної програми було реалізовано програми для перевірки результатів. Програма «tester.py» використовує ті ж підпрограми, що і «learn.py» але завантажує натреновану модель, вибірку, яка була використана при навчанні, створює токенізовану вибірку, виводить приклади перекладів, будує графіки та виводить результат метрик BLEU та METEOR.

```
model_name = 'en-uk-model1-w15k-l12-c3-e50'

preproc_en_seq, preproc_uk_seq, en_tokenizer, uk_tokenizer =
test_helper.load_preprocessed_dataset(model_name)
# Load trained model
model = load_model('models/{}/{}.h5'.format(model_name, model_name))

# Test model
test_helper.print_predictions(model, preproc_en_seq[:5], preproc_uk_seq[:5],
en_tokenizer, uk_tokenizer)

preproc_test_en_seq, preproc_test_uk_seq =
test_helper.get_test_phrases(en_tokenizer, uk_tokenizer, preproc_en_seq)
test_helper.print_predictions(model, preproc_test_en_seq, preproc_test_uk_seq,
en_tokenizer, uk_tokenizer)

# Print logs
logs_df = helper.load_logs('models/{}/logs-{}.csv'.format(model_name,
model_name))

logs_df.plot()
plt.xlabel('Епохи')
plt.ylabel('Значення')
plt.show()

reference_sentences, predicted_sentences = test_helper.get_predictions(model,
preproc_test_en_seq, preproc_test_uk_seq,

uk_tokenizer)
bleu_scores = []
meteor_scores = []

for predicted_sentence, reference_sentence in zip(predicted_sentences,
reference_sentences):
```

```
    predicted_tokens = nltk.word_tokenize(predicted_sentence.lower())
    reference_tokens = nltk.word_tokenize(reference_sentence.lower())


bleu_scores.append(nltk.translate.bleu_score.sentence_bleu([reference_tokens],
predicted_tokens, weights=(0.5, 0.5),

smoothing_function=SmoothingFunction().method1))

meteor_scores.append(nltk.translate.meteor_score.meteor_score([reference_tokens]
, predicted_tokens))

print("Результат BLEU:", sum(bleu_scores) / len(bleu_scores))
print("Результат METEOR:", sum(meteor_scores) / len(meteor_scores))
```

Програма «neural_translator.py» використовується для перекладу в режимі реального часу. Працює подібним чином як «tester.py» але тексти для перекладу просить ввести з клавіатури. Текст введений з клавіатури попередньо обробляється: приводиться до нижнього регістру, вилучаються розділові знаки. Можна вводити як окремі слова, так і речення, так і декілька речень через знак «;».

```
model_name = 'en-uk-model1-w15k-l12-c3-e50'

preproc_en_seq, preproc_uk_seq, en_tokenizer, uk_tokenizer =
test_helper.load_preprocessed_dataset(model_name)
# Load trained model
model = load_model('models/{}/{}.h5'.format(model_name, model_name))

# model.summary()

def test(test_seq_en):
    # Test model
    prep_test_en_seq, prep_test_uk_seq = tokenizer.preprocess_with_tokenizer(
        test_seq_en, ['-' for _ in range(0, len(test_seq_en))], preproc_en_seq,
en_tokenizer, uk_tokenizer)
    return test_helper.print_predictions(model, prep_test_en_seq,
prep_test_uk_seq, en_tokenizer, uk_tokenizer,
                                         with_orig=False)
```

```
print("Translator. Enter phrases (can be split by ';') or type 'exit'")

while True:
    inp = input("Enter phrase:")
    inp = re.sub(r'([,.?!])', '', inp).replace('  ', ' ')
    inp = inp.lower().rstrip().lstrip()
    if inp == 'exit':
        break
    test(inp.split(";"))
```

В результаті виводиться введений текст з клавіатури, який попередньо оброблений, але містить слова які присутні в словнику, та прогнозований переклад (рис. 3.6).

```
Translator. Enter phrases (can be split by ';') or type 'exit'
Enter phrase:I think there's been a mistake.;do you know me?;unknownword i've been there twice
1/1 [==============================] - 1s 1s/step
+--------------------------------+-----------------------+
|        Текст анлійською        |   Машинний Переклад   |
+--------------------------------+-----------------------+
| i think there's been a mistake | думаю виникла помилка |
|         do you know me         |     ти мене знаєш     |
|      i've been there twice     |    я там був двічі    |
+--------------------------------+-----------------------+
```

Рис. 3.6. Приклад роботи програми Neural Translator.

3.3.3. Результати навчання моделей

Було побудовано три моделі, на основі кодер-декодер моделі, в якості рекурентного шару в першій моделі використано звичайний LSTM (рис. 3.7) шар, в другій (рис. 3.8) та третій (рис. 3.9) Bidirectional LSTM.

В всіх моделях в якості останнього шару використовується Dense (activation=’softmax’) – перетворення векторів Embedding на вектори розмірності всього словника. Через використання softmax під номером кожного слова буде стояти ймовірність, з якою вектор відповідає цьому слову.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 8, 512)            6866432

 lstm_2 (LSTM)               (None, 512)               2099200

 repeat_vector_1 (RepeatVect  (None, 8, 512)           0
 or)

 lstm_3 (LSTM)               (None, 8, 512)            2099200

 dense (Dense)               (None, 8, 34517)          17707221

=================================================================
```

Рис. 3.7. Структура моделі 1 з звичайним LSTM шаром

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 8, 512)            6866432

 bidirectional (Bidirectiona  (None, 1024)             4198400
 l)

 repeat_vector (RepeatVector  (None, 8, 1024)          0
 )

 bidirectional_1 (Bidirectio  (None, 8, 1024)          6295552
 nal)

 dropout (Dropout)           (None, 8, 1024)           0

 time_distributed (TimeDistr  (None, 8, 34517)         35379925
 ibuted)

=================================================================
```

Рис. 3.8. Структура моделі 2 з Bidirectional LSTM шаром

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 8, 512)            6866432

 bidirectional (Bidirectiona  (None, 1024)             4198400
 l)

 repeat_vector (RepeatVector  (None, 8, 1024)          0
 )

 bidirectional_1 (Bidirectio  (None, 8, 1024)          6295552
 nal)

 time_distributed (TimeDistr  (None, 8, 512)           524800
 ibuted)

 dropout (Dropout)           (None, 8, 512)            0

 time_distributed_1 (TimeDis  (None, 8, 34517)         17707221
 tributed)

=================================================================
```

Рис. 3.9. Структура моделі 3 з Bidirectional LSTM шаром

В моделі 2 і 3 крім шару Dense також використовується шар Dropout, який допомагає уникнути перенавчання, тобто модель не стає занадто "залежною" від конкретних шаблонів у тренувальних даних і здатна краще узагальнювати нові дані. Використання Dropout також може покращити стійкість моделі до шуму та покращити її загальну ефективність. В моделі 3 перед цим шаром також додано шар Dense з лінійною функцією активації relu, який допомагає моделі отримати більш виразні ознаки з кожного шару в послідовності та покращити її здатність до аналізу та розуміння послідовних даних.

Під час навчання використовувалось 55% центрального процесора та 87% графічного (рис. 3.10), а також 8 Гб оперативної пам’яті. Навчання проводилось на протязі 50 епох, процес навчання тривав близько 5 годин. Як видно з графіків функції втрат (рис.3.11) та точності (рис.3.12) модель 3 навчалась краще.

Process Name	% CPU	% GPU
python3.10	54,6	87,3

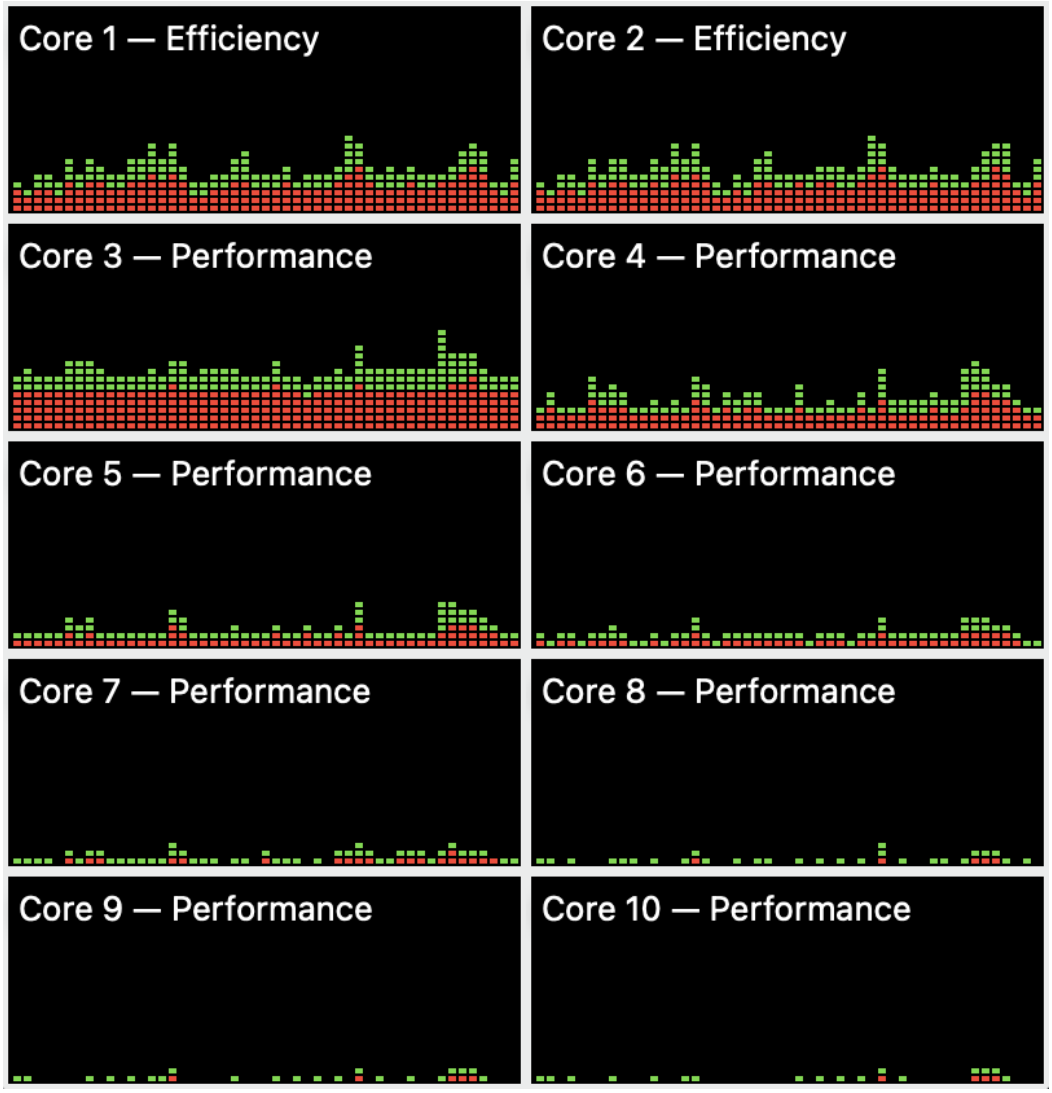

Рис. 3.10. Використання ресурсів комп’ютера під час навчання нейромережі

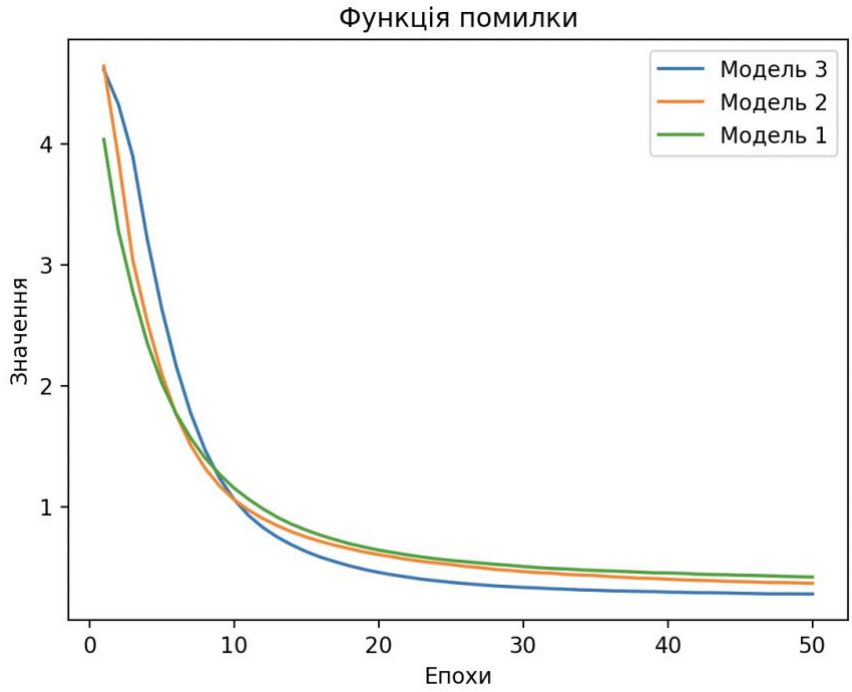

Рис. 3.11. Графік залежності значення функції похибки від номеру епохи

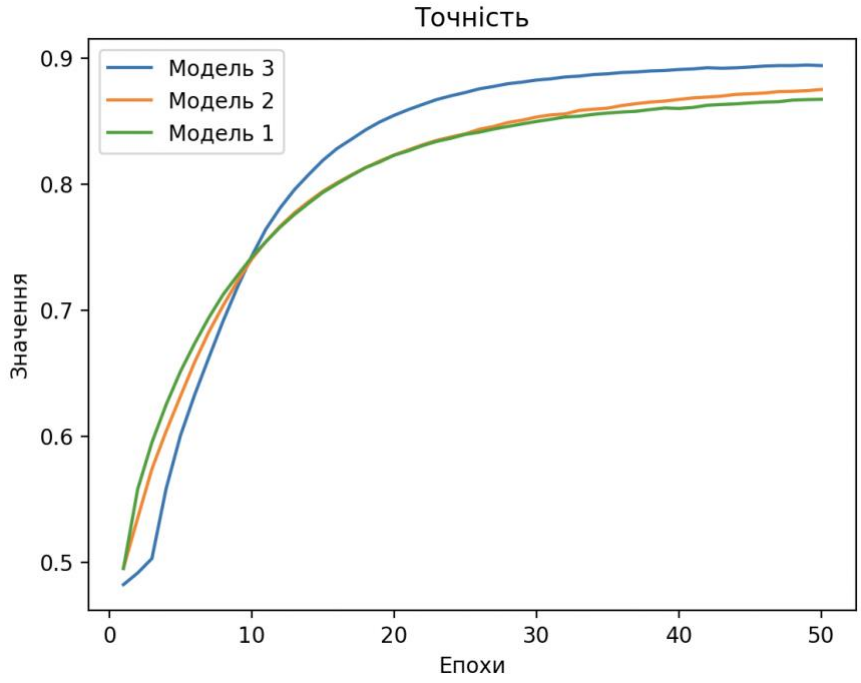

Рис. 3.12. Графік залежності значення точності від номеру епохи

З результатів перекладу можна побачити, що дійсно модель 3 (табл. 3.3) показала кращі результати перекладу, ніж модель 1 (табл. 3.1) та модель 2 (табл. 3.2). Також ці переклади були досліджені метриками BLEU та METEOR.

Для цього в програмі «tester.py» було використано бібліотеку Natural Language Toolkit (NLTK) (табл. 3.4).

```
bleu_scores = []
meteor_scores = []
for predicted_sentence, reference_sentence in zip(predicted_sentences,
reference_sentences):
    predicted_tokens = nltk.word_tokenize(predicted_sentence.lower())
    reference_tokens = nltk.word_tokenize(reference_sentence.lower())
bleu_scores.append(nltk.translate.bleu_score.sentence_bleu([reference_tokens],
predicted_tokens, weights=(0.5, 0.5),
smoothing_function=SmoothingFunction().method1))
meteor_scores.append(nltk.translate.meteor_score.meteor_score([reference_tokens]
, predicted_tokens))
print("Результат BLEU:", sum(bleu_scores) / len(bleu_scores))
print("Результат METEOR:", sum(meteor_scores) / len(meteor_scores))
```

Таблиця 3.1. Результати перекладу моделлю 1

Текст англійською	**Машинний переклад**	**Очікуваний переклад**
do you know me	ви мене знаєш	ти мене знаєш
show me your car	покажіть мені свій машину	покажи мені свою машину
i have never ever been there	я ніколи там ніколи не був	я ніколи не був там
i want to buy a car	я хочу купити машину	я хочу купити машину
i wish i had a car	шкода що я не маю машини	шкода що я не маю машини
please	будь ласка	будь ласка
don’t touch this tea	не торкайся цієї	не чіпай цей чай
i hope	сподіваюся прикро	я сподіваюся

Таблиця 3.2. Результати перекладу моделі 2

Текст англійською	**Машинний переклад**	**Очікуваний переклад**
do you know me	ви мене знаєш	ти мене знаєш
show me your car	покажи мені свою машину	покажи мені свою машину
i have never ever been there	я ніколи ніколи ніколи не був	я ніколи не був там
i want to buy a car	я хочу купити машину	я хочу купити машину
i wish i had a car	шкода що я не маю машини	шкода що я не маю машини
please	будь ласка	будь ласка
don’t touch this tea	не торкайся цього чай	не чіпай цей чай
i hope	я що	я сподіваюся

Таблиця 3.3. Результати перекладу моделі 3

Текст англійською	**Машинний переклад**	**Очікуваний переклад**
do you know me	ти мене знаєш	ти мене знаєш
show me your car	покажи мені свою машину	покажи мені свою машину
i have never ever been there	я ніколи ніколи не не була	я ніколи не був там
i want to buy a car	я хочу купити машину	я хочу купити машину
i wish i had a car	шкода що я не маю машини	шкода що я не маю машини
please	будь ласка ласка	будь ласка
don’t touch this tea	не чіпай цього чай	не чіпай цей чай
i hope	я сподіваюся	я сподіваюся

Таблиця 3.4. Результати дослідження моделей метриками BLEU та METEOR

Модель	**BLEU**	**METEOR**
Модель 1	0.59	0.63
Модель 2	0.7	0.72
Модель 3	0.82	0.78

ВИСНОВКИ

У курсовій роботі було досліджено різні моделі нейронних мереж для лінгвістичного перекладу. Були проаналізовані літературні джерела з обраної тематики, а саме – методи нейронного машинного перекладу текстів: методи засновані на RNN, основна модель кодер-декодер. Під час розробки застосовані різні алгоритми та бібліотеки мови Python, яка найкращим чином пристосована для реалізації методів машинного навчання.

При порівнянні розглянутих моделей нейронних мереж кращі результати перекладу показує мережа на основі Transformer з Bidirectional LSTM, ніж зі звичайною LSTM та мережа з додатковим Dense шаром з лінійною функцією активації, ніж без нього. Це пов’язано з тим, Bidirectional LSTM структура дозволяє мережі для кожного токена мати інформацію як про попередні до неї токени, так і про наступні, де значення слова не можна виявити лише з попередніх слів, а шар Dense покращив здатність моделі до витягування важливих ознак з послідовностей даних.

Великий вплив на ефективність часу для тренування моделі має швидкість роботи програми, тому слід використовувати один з фреймворків, що підтримують обчислення на GPU.

Список літератури

1. Хорошун О. О. Машинний переклад: історичний огляд. // Нова філологія - 2021, (82), 333-337.
2. Hutchins, J. (2005) The history of machine translation in a nutshell
3. Л. О. Гречуха, В. В. Кузебна // Молодий вчений. - 2017. - № 2. - С. 372-375.
4. Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre - Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation // eprint arXiv:1406.1078
5. Alexander Molchanov and Fedor Bykov. 2016. PROMT Translation Systems for WMT 2016 Translation Tasks. // First Conference on Machine Translation: Volume 2, Shared Task Papers, pages 339–343, Berlin, Germany. Association for Computational Linguistics.
6. Alexander Molchanov, Vladislav Kovalenko, and Fedor Bykov. 2021. PROMT Systems for WMT21 Terminology Translation Task. // Sixth Conference on Machine Translation, pages 835–841, Online. Association for Computational Linguistics.
7. Yonghui Wu, Mike Schuster, Zhifeng Chen - Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation // 2016, arXiv:1609.08144.
8. Despoina Mouratidis, Maria Stasimioti, Vilelmini Sosoni - NoDeeLe: A Novel Deep Learning Schema for Evaluating Neural Machine Translation Systems // TRITON, vol. 37, 2021.
9. Хайкин, Саймон. Нейронные сети: полный курс, 2-е издание. : Пер. с англ. – М. : Издательский дом «Вильямс», 2006. – 1104 с.
10. TensorFlow tutorialspoint. [Електронний ресурс] Режим доступу: https://www.tutorialspoint.com/tensorflow/tensorflow_tutorial.pdf
11. Keras Documentation [Електронний ресурс] Режим доступу: https://keras.io/api/

12. Zhen Fang, Nicolas Crimier, Lisa Scanu, Alphanie Midelet, Amr Alyafi, et al.. Multi-zone indoor temperature prediction with LSTM-based sequence to sequence model. Energy and Buildings, 2021, 245, pp.111053.

13. Muhammad Sarim Mehdi - Trajectory Prediction for ADAS // Thesis for Master's in Automation Engineering, DOI:10.13140/RG.2.2.11891.35369, 2020.

Додаток 1. Лістінг додатку «Neural Translator»

Файл «neural_transaltor.py»:

```
import re
from keras.models import load_model
import test_helper
import tokenizer

model_name = 'en-uk-model1-w15k-l12-c3-e50'

preproc_en_seq, preproc_uk_seq, en_tokenizer, uk_tokenizer =
test_helper.load_preprocessed_dataset(model_name)
# Load trained model
model = load_model('models/{}/{}.h5'.format(model_name, model_name))


def test(test_seq_en):
    # Test model
    prep_test_en_seq, prep_test_uk_seq = tokenizer.preprocess_with_tokenizer(
        test_seq_en, ['-' for _ in range(0, len(test_seq_en))], preproc_en_seq,
en_tokenizer, uk_tokenizer)
    return test_helper.print_predictions(model, prep_test_en_seq,
prep_test_uk_seq, en_tokenizer, uk_tokenizer,
                                         with_orig=False)


print("Перекладач.\nВведіть фрази (можуть бути розділені симоволом ';') або
напишіть 'exit' для виходу")

while True:
    inp = input("Enter phrase:")
    inp = re.sub(r'([,.?!])', '', inp).replace('  ', ' ')
    inp = inp.lower().rstrip().lstrip()
    if inp == 'exit':
        break
    test(inp.split(";"))
```

Файл «learn.py»:

```
from sklearn.model_selection import train_test_split

import helper
import tokenizer
import test_helper
import model_builder

# Load English data
en_sentences = helper.load_data('data/vocab_eng.txt')
helper.print_words_count(en_sentences, "English")
# Load Ukrainian data
uk_sentences = helper.load_data('data/vocab_ukr.txt')
helper.print_words_count(uk_sentences, "Ukrainian")
print('Dataset length:', len(en_sentences))


# Preprocess dataset
preproc_en_seq, preproc_uk_seq, en_tokenizer, uk_tokenizer =
tokenizer.preprocess(en_sentences, uk_sentences)
# Build and train model
```

```
en_vocab_size = len(en_tokenizer.word_index) + 1
uk_vocab_size = len(uk_tokenizer.word_index) + 1

X_train, X_test, y_train, y_test = train_test_split(preproc_en_seq,
preproc_uk_seq, test_size=0.2, random_state=1)

model = model_builder.build(preproc_en_seq.shape, preproc_uk_seq.shape[1],
en_vocab_size, uk_vocab_size, 3)
training_history = model.fit(X_train, y_train, batch_size=512, epochs=50,
validation_data=(X_test, y_test))
helper.save_model('en-uk-model2-w15k-l12-c3-e50', model, training_history)

# Test model
test_helper.print_predictions(model, preproc_en_seq[:5], preproc_uk_seq[:5],
en_tokenizer, uk_tokenizer)

preproc_test_en_seq, preproc_test_uk_seq =
test_helper.get_test_phrases(en_tokenizer, uk_tokenizer, preproc_en_seq)
test_helper.print_predictions(model, preproc_test_en_seq, preproc_test_uk_seq,
en_tokenizer, uk_tokenizer)
```

Файл «dataset_converter.py»:

```
import helper
import re
from collections import Counter
import pandas as pd


def preprocess_string(old_string):
    new_string = old_string.replace('\u00A0', ' ').lower()
    new_string = re.sub(r'(["«»—;\-`':])', r' ', new_string)
    new_string = re.sub(r'([,.?!])', r' \1 ', new_string).replace('  ', '
').rstrip()
    return new_string


def get_words_stats(array):
    print('Dataset length:', len(array))
    ukr_sentences = [s[0] for s in array]
    ukrainian_words_counter = Counter([word for sentence in ukr_sentences for
word in sentence.split()])
    helper.print_words_count(ukr_sentences, "Ukrainian")
    eng_sentences = [s[1] for s in array]
    english_words_counter = Counter([word for sentence in eng_sentences for word
in sentence.split()])
    helper.print_words_count(eng_sentences, "English")
    return ukrainian_words_counter, english_words_counter


def get_most_common_words(counters, unique):
    most_common_ukr = [w for w in
list(zip(*counters[0].most_common(unique)))[0]]
    most_common_eng = [w for w in
list(zip(*counters[1].most_common(unique)))[0]]
    return most_common_ukr, most_common_eng


def trim_sentences(array, commons, max_length, contains_commons):
    print('\ntrimming........')
```

```
    new_array = []
    for i in array:
        ukr_words_in_sentence = i[0].split()
        eng_words_in_sentence = i[1].split()

        if len(ukr_words_in_sentence) >= max_length or
len(eng_words_in_sentence) >= max_length:
            continue

        count_of_common = 0


        for w in i[0].split():
            if w in commons[0]:
                count_of_common += 1
        if count_of_common >= contains_commons:
            new_array.append([" ".join(ukr_words_in_sentence), "
".join(eng_words_in_sentence)])
    print('end trimming........\n')
    return new_array

options = {'commons': 15000, 'max_length': 10, 'min_commons': 3}
df = pd.read_csv('data/ukr-eng-pairs.csv', delimiter=';', header=None)
sentences = [[preprocess_string(string[0]), preprocess_string(string[1])] for
string in df[[1, 3]].to_numpy()]
most_common_words = get_most_common_words(get_words_stats(sentences),
options['commons'])
sentences = trim_sentences(sentences, most_common_words, options['max_length'],
options['min_commons'])
get_words_stats(sentences)

ukr_sentences = [w[0] for w in sentences]
eng_sentences = [w[1] for w in sentences]
helper.write_file("data/vocab_ukr.txt", ukr_sentences)
helper.write_file("data/vocab_eng.txt", eng_sentences)
```

Файл «helper.py»:

```
import os
import collections
import numpy as np
import pandas as pd
from prettytable import PrettyTable
import tokenizer


def load_data(path):
    input_file = os.path.join(path)
    with open(input_file, "r") as f:
        data = f.read()

    return data.split('\n')

def load_logs(file_name, delimiter=';'):
    df = pd.read_csv(file_name, delimiter=delimiter)
    df = df[df.columns[:2]]
    df.columns = ['Функція помилки', 'Точність']
    return df

def write_file(name, data_array):
```

```
    with open(name, "w") as txt_file:
        for line in data_array:
            txt_file.write(line + "\n")


def print_words_count(sentences, lang_name):
    counter = collections.Counter([word for sentence in sentences for word in
sentence.split()])
    print('{} {} words.'.format(len([word for sentence in sentences for word in
sentence.split()]), lang_name))
    print('{} unique {} words.'.format(len(counter), lang_name))
    print('20 Most common words in the {} dataset:'.format(lang_name))
    print('"' + '" "'.join(list(zip(*counter.most_common(20)))[0]) + '"')
    print()


def print_results(predictions, en_seq, uk_seq, en_tk, uk_tk, with_orig=True):
    results = PrettyTable(max_table_width=195)
    if with_orig:
        results.field_names = ["Текст анлійською", "Машинний Переклад",
"Очікуваний переклад"]
        for i in range(0, len(predictions)):
            results.add_row([tokenizer.tokens_to_text(en_seq[i], en_tk),
                             tokenizer.logits_to_text(predictions[i], uk_tk),
                             tokenizer.tokens_to_text([u[0] for u in uk_seq[i]],
uk_tk)])
    else:
        results.field_names = ["Текст анлійською", "Машинний Переклад"]
        for i in range(0, len(predictions)):
            results.add_row([tokenizer.tokens_to_text(en_seq[i], en_tk),
                              tokenizer.logits_to_text(predictions[i], uk_tk)])
    print(results)
    return results

def save_train_history(file_name, training_history):
    np.savetxt(fname=file_name, delimiter=';',
               X=np.column_stack((training_history.history['loss'],
training_history.history['accuracy'],
                             training_history.history['val_loss'],
training_history.history['val_accuracy'])),
               header="loss;accuracy;val_loss;val_accuracy")

def save_model(model_name, model, training_history):
    model.save('models/{}.h5'.format(model_name))
    save_train_history('logs/logs-{}.csv'.format(model_name), training_history)
```

Файл «model_builder.py»:

```
from keras.layers import Dense, TimeDistributed, RepeatVector, Bidirectional,
Dropout, LSTM
from keras.layers import Embedding
from keras.losses import sparse_categorical_crossentropy
from keras.models import Sequential
from keras.optimizers import Adam

def build(input_shape, output_sequence_length, en_vocab_size, uk_vocab_size,
model_num=3):
    learning_rate = 0.003
    model = Sequential()
```

```
    if model_num == 1:
        # Build the layers Embedding
        model.add(Embedding(en_vocab_size, 512, input_length=input_shape[1],
input_shape=input_shape[1:]))
        # Encoder
        model.add(LSTM(512))
        model.add(RepeatVector(output_sequence_length))
        # Decoder
        model.add(LSTM(512, return_sequences=True))
        model.add(Dense(uk_vocab_size, activation='softmax'))
        model.compile(loss=sparse_categorical_crossentropy,
                      optimizer=Adam(learning_rate),
                      metrics=['accuracy'])

    if model_num == 2:
        # Build the layers Embedding
        model.add(Embedding(en_vocab_size, 512, input_length=input_shape[1],
input_shape=input_shape[1:]))
        # Encoder
        model.add(Bidirectional(LSTM(512)))
        model.add(RepeatVector(output_sequence_length))
        # Decoder
        model.add(Bidirectional(LSTM(512, return_sequences=True)))
        model.add(Dropout(0.3))
        model.add(TimeDistributed(Dense(uk_vocab_size, activation='softmax')))
        model.compile(loss=sparse_categorical_crossentropy,
                      optimizer=Adam(learning_rate),
                      metrics=['accuracy'])

    if model_num == 3:
        # Build the layers Embedding
        model.add(Embedding(en_vocab_size, 512, input_length=input_shape[1],
input_shape=input_shape[1:]))
        # Encoder
        model.add(Bidirectional(LSTM(512)))
        model.add(RepeatVector(output_sequence_length))
        # Decoder
        model.add(Bidirectional(LSTM(512, return_sequences=True)))
        model.add(TimeDistributed(Dense(512, activation='relu')))
        model.add(Dropout(0.5))
        model.add(TimeDistributed(Dense(uk_vocab_size, activation='softmax')))
        model.compile(loss=sparse_categorical_crossentropy,
                      optimizer=Adam(learning_rate),
                      metrics=['accuracy'])

    model.summary()
    return model
```

Файл «test_helper.py»:

```
import numpy as np
import helper
import tokenizer

def print_predictions(trained_model, prep_en_seq, prep_uk_seq, tokenizer_en,
tokenizer_uk, with_orig=True):
    sentences_to_test = np.array(prep_en_seq)
```

```
    model_predictions = trained_model.predict(sentences_to_test,
len(sentences_to_test))
    return helper.print_results(model_predictions, prep_en_seq, prep_uk_seq,
tokenizer_en, tokenizer_uk, with_orig)

def get_predictions(trained_model, prep_en_seq, prep_uk_seq, tokenizer_uk):
    sentences_to_test = np.array(prep_en_seq)
    model_predictions = trained_model.predict(sentences_to_test,
len(sentences_to_test))
    original = []
    translated = []

    for i in range(0, len(model_predictions)):
        original.append(tokenizer.tokens_to_text([u[0] for u in prep_uk_seq[i]],
tokenizer_uk))
        translated.append(tokenizer.logits_to_text(model_predictions[i],
tokenizer_uk))

    return original, translated

def get_test_phrases(tokenizer_en, tokenizer_uk, preproc_en_seq):
    test_seq_en = ["do you know me", "show me your car", "i have never ever been
there",
                   "i want to buy a car", "i wish i had a car", "please", "don't
touch this tea", "i hope"]
    test_seq_uk = ["чи ти мене знаєш", "покажи мені свою машину", "я ніколи не
був там",
                   "я хочу купити машину", "шкода , що я не маю машини", "будь
ласка", "не чіпай цей чай", "я сподіваюся"]
    prep_test_en_seq, prep_test_uk_seq = tokenizer.preprocess_with_tokenizer(
        test_seq_en, test_seq_uk, preproc_en_seq, tokenizer_en, tokenizer_uk)
    return prep_test_en_seq, prep_test_uk_seq

def load_preprocessed_dataset(model_name):
    # Load English data
    en_sentences =
 helper.load_data('models/{}/vocab_eng.txt'.format(model_name))
    # Load Ukrainian data
    uk_sentences =
helper.load_data('models/{}/vocab_ukr.txt'.format(model_name))
    print('Dataset length:', len(en_sentences))

    # Preprocess dataset
    preproc_en_seq, preproc_uk_seq, en_tokenizer, uk_tokenizer =
tokenizer.preprocess(en_sentences, uk_sentences)
    return preproc_en_seq, preproc_uk_seq, en_tokenizer, uk_tokenizer
```

Файл «tester.py»:

```
from keras.models import load_model
from matplotlib import pyplot as plt
import nltk
from nltk.translate.bleu_score import SmoothingFunction
import helper
import test_helper

model_name = 'en-uk-model1-w15k-l12-c3-e50'
```

```
preproc_en_seq, preproc_uk_seq, en_tokenizer, uk_tokenizer =
test_helper.load_preprocessed_dataset(model_name)
# Load trained model
model = load_model('models/{}/{}.h5'.format(model_name, model_name))

# Test model
test_helper.print_predictions(model, preproc_en_seq[:5], preproc_uk_seq[:5],
en_tokenizer, uk_tokenizer)

preproc_test_en_seq, preproc_test_uk_seq =
test_helper.get_test_phrases(en_tokenizer, uk_tokenizer, preproc_en_seq)
test_helper.print_predictions(model, preproc_test_en_seq, preproc_test_uk_seq,
en_tokenizer, uk_tokenizer)

# Print logs
logs_df = helper.load_logs('models/{}/logs-{}.csv'.format(model_name,
model_name))

logs_df.plot()
plt.xlabel('Епохи')
plt.ylabel('Значення')
plt.show()

reference_sentences, predicted_sentences = test_helper.get_predictions(model,
preproc_test_en_seq, preproc_test_uk_seq,
uk_tokenizer)
bleu_scores = []
meteor_scores = []

for predicted_sentence, reference_sentence in zip(predicted_sentences,
reference_sentences):
    predicted_tokens = nltk.word_tokenize(predicted_sentence.lower())
    reference_tokens = nltk.word_tokenize(reference_sentence.lower())

bleu_scores.append(nltk.translate.bleu_score.sentence_bleu([reference_tokens],
predicted_tokens, weights=(0.5, 0.5),
smoothing_function=SmoothingFunction().method1))

meteor_scores.append(nltk.translate.meteor_score.meteor_score([reference_tokens]
, predicted_tokens))

print("Результат BLEU:", sum(bleu_scores) / len(bleu_scores))
print("Результат METEOR:", sum(meteor_scores) / len(meteor_scores))
```

Файл «tokenizer.py»:

```
import numpy as np
from keras.preprocessing.text import Tokenizer
from keras.utils import pad_sequences

def tokenize(x):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(x)
    return tokenizer.texts_to_sequences(x), tokenizer
def pad(x, length=None):
```

```
    return pad_sequences(x, maxlen=length, padding='post')

def preprocess(en_sentences, uk_sentences):
    preprocess_en, en_tk = tokenize(en_sentences)
    preprocess_uk, uk_tk = tokenize(uk_sentences)

    preprocess_en = pad(preprocess_en)
    preprocess_uk = pad(preprocess_uk)

    # Keras's sparse_categorical_crossentropy function requires the labels to be
in 3 dimensions
    preprocess_uk = preprocess_uk.reshape(*preprocess_uk.shape, 1)

    print('Data Preprocessed')
    print("Max English sentence length:", preprocess_en.shape[1])
    print("Max Ukrainian sentence length:", preprocess_uk.shape[1])
    print("English vocabulary size:", len(en_tk.word_index))
    print("Ukrainian vocabulary size:", len(uk_tk.word_index))

    return preprocess_en, preprocess_uk, en_tk, uk_tk

def logits_to_text(logits, tokenizer):
    index_to_words = {id: word for word, id in tokenizer.word_index.items()}
    index_to_words[0] = ''
    return ' '.join([index_to_words[prediction] for prediction in
np.argmax(logits, 1)]).rstrip()


def tokens_to_text(tokens, tokenizer):
    index_to_words = {id: word for word, id in tokenizer.word_index.items()}
    index_to_words[0] = ''
    return ' '.join([index_to_words[token] for token in tokens]).rstrip()

def text_to_tokens(sentences, eng_sentences, tokenizer):
    sentences = [[tokenizer.word_index.get(word, 0) for word in
sentence.split()] for sentence in sentences]
    sentences = pad_sequences(sentences, maxlen=eng_sentences.shape[-1],
padding='post')
    return sentences

def preprocess_with_tokenizer(test_seq_en, test_seq_uk, preproc_en_seq,
en_tokenizer, uk_tokenizer):
    preproc_test_en_seq = text_to_tokens(test_seq_en, preproc_en_seq,
en_tokenizer)
    preproc_test_uk_seq = text_to_tokens(test_seq_uk, preproc_en_seq,
uk_tokenizer)
    preproc_test_uk_seq =
preproc_test_uk_seq.reshape(*preproc_test_uk_seq.shape, 1)
    return preproc_test_en_seq, preproc_test_uk_seq
```

Файл «graph_plotter.py»:

```
from matplotlib import pyplot as plt
import helper

models = ["en-uk-model3-w15k-l12-c3-e50", "en-uk-model2-w15k-l12-c3-e50", "en-
uk-model1-w15k-l12-c3-e50"]
model_labels = ["Модель 3", "Модель 2", "Модель 1"]
logs_dfs = [helper.load_logs('models/{}/logs-{}.csv'.format(i, i)) for i in
models]
columns = logs_dfs[0].columns
```

```
for i in range(len(models)):
    logs_dfs[i].plot()
    plt.title(model_labels[i])
    plt.xlabel('Епохи')
    plt.ylabel('Значення')
    plt.show()

for i in range(len(columns)):
    for k in range(len(models)):
        y = logs_dfs[k][logs_dfs[k].columns[[i]]].to_numpy().reshape(-1)
        x = range(1, len(y) + 1)
        plt.plot(x, y, label=model_labels[k])
    plt.title(columns[i])
    plt.legend()
    plt.xlabel('Епохи')
    plt.ylabel('Значення')
    plt.show()
```