Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

Використання нейронних мереж при побудові систем машинного перекладу

Текстова частина до курсової роботи

за спеціальністю «Інженерія Програмного Забезпечення» 121

Керівник курсової роботи

доктор техн. наук,

декан ФІ Глибовець А.М.

____

(підпис)

“___” _____________ 2021 року

Виконала студентка ІПЗ-3

Чередник К.В.

“___” _____________ 2021 року

Київ 2021

Календарний план виконання курсової роботи

№ п/п	Назва етапу курсової роботи	Термін виконання	Примітка
1	Отримання завдання на курсову роботу	14.10.2020	
2.	Пошук тематичної наукової літератури	15.10.2020	
3.	Ознайомлення з інтерфейсом бібліотеки “Tensorflow”	18.10.2020	
4.	Ознайомлення з літературою по нейронним мережам	25.10.2020	
5.	Створення і тренування RNN моделі	05.11.2020	
6.	Опис результатів	08.11.2020	
7.	Створення і тренування LSTM моделі	10.11.2020	
8.	Опис результатів	15.11.2020	
9.	Створення і тренування моделі Transformers	01.12.2020	
10.	Опис результатів	07.12.2020	
11.	Об’єднання всіх частин курсової роботи	10.03.2021	
12.	Написання теоретичної частини	20.03.2021	
13.	Надання роботи керівнику для перевірки	07.04.2021	
14.	Корегування роботи відповідно до зауважень керівника	15.04.2021	
15.	Остаточне оформлення курсової роботи та презентації	16.05.2021	
16.	Захист курсової роботи		

Студент ____________Чередник К.В.__________

Керівник ____________Глибовець А.М.________

“______” ________________________

Зміст

АНОТАЦІЯ 4

ВСТУП 5

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ 7

1.1 Проблеми перекладу 7

1.2 Використання нейронних мереж.. 8

1.3 Оцінювання згенерованого перекладу мережі 10

РОЗДІЛ 2. ПРЕДСТАВЛЕННЯ ДАНИХ 11

2.1 Обробка тексту 11

2.2 Токенізація.. 11

2.2.1 Word-based tokenization 11

2.2.2 Character-based tokenization.. 12

2.2.3 Subword tokenization.. 12

2.3 Embeddings 13

2.4 Створення вхідних та вихідних даних для моделі.. 15

2.5 Розділення на вибірку для тренування, валідації та тестування моделі 16

РОЗДІЛ 3. ОСНОВНА КОНЦЕПЦІЯ ПОБУДОВИ МОДЕЛЕЙ ДЛЯ МАШИННОГО ПЕРЕКЛАДУ 17

3.1 Архітектура RNN 17

3.1.1 Загальний опис 17

3.1.2 Проблеми RNN 18

3.2 Архітектура LSTM 19

3.3 Архітектура GRU 20

3.4 Архітектура Bidirectional RNN 21

3.5 Архітектура Transformers 22

3.5.1 Загальний опис 22

3.5.2 Позиційне кодування.. 23

3.5.3 Маскування 23

3.5.4 Scaled dot product attention 24

3.5.5 Multi-head attention 25

РОЗДІЛ 4. ПРАКТИЧНА ЧАСТИНА ДОСЛІДЖЕНЬ 26

4.1 ВИЗНАЧЕННЯ ФРЕЙМВОРКУ ДЛЯ РОБОТИ З НЕЙРОННИМИ МЕРЕЖАМИ ... 26

4.2 ПОШУК ТРЕНУВАЛЬНОЇ ВИБІРКИ 27

4.3 ДОСЛІДЖЕННЯ РОБОТИ МОДЕЛЕЙ ТИПУ RNN 28

4.3.1 Основні моменти побудови моделі RNN у програмному коді ... 28

4.3.2 Результати RNN 31

4.3.3 Розгляд варіантів представлення вхідних даних при великій довжині речень 36

4.4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ МОДЕЛЕЙ ТИПУ TRANSFORMERS .. 36

4.4.1 Основні моменти побудови моделі Transformers у програмному коді 36

4.4.2 Результати Transformers 40

ВИСНОВОК 43

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 44

ДОДАТОК A 46

ДОДАТОК B 47

ДОДАТОК C 48

ДОДАТОК D 49

ДОДАТОК E 49

ДОДАТОК F 50

ДОДАТОК G 51

ДОДАТОК H 54

Анотація

У даній роботі описуються різні підходи до рішення проблеми машинного перекладу через нейронні мережі. Наводяться кілька інструментів для зручного перегляду результатів тренування моделей та відображення їх структур. Розглядаються різні архітектури мереж. Пояснюються основні моменти написання коду для розробки таких архітектур. Порівнюються результати використання різних моделей та досліджується вплив гіперпараметрів на ці результати.

В той час як у якості практичного застосування моделей, що розглядаються, пропонується тільки задача машинного перекладу, основні концепти роботи мереж можуть стати у пригоді для вирішення завдань з інших галузей.

Аналогічно, основні етапи попередньої обробки вхідного корпусу речень можуть бути використані й для інших задач з автоматизації лінгвістичної сфери.

Вступ

Переклад є однією з найважливіших задач людства, оскільки є ланкою формування міжкультурних відносин. Мета перекладу – якомога точніше передати зміст тексту з однієї мови на іншу при цьому не знехтувавши стилістичною своєрідністю вихідної мови, в ідеалі зі збереженням мистецьких особливостей тексту.

В наш час до кожної сфери намагаються під'єднати автоматизацію та комп'ютаризацію, звичайно, це не оминуло й сферу перекладу. Машинний переклад може використовуватись як сам по собі, так і як допоміжний інструмент для перекладу людиною.

Три основні підходи до машинного перекладу - rule-based, статичні та нейронні системи.

Найбільш примітивний машинний переклад полягає у заміні слів однієї природної мови на слова іншої. Зрозуміло, що такий спосіб у будь-якому разу не можна назвати якісним. Опираючись на відмінності між граматикою та семантикою різних мов, а також на приналежність мові омонімів, такий підхід з великою вірогідністю може призвести по спотворення смислового навантаження вхідного речення, у гіршому випадку, коли структура мов сильно різниться, на виході можна очікувати повну нісенітницю.

Перші спроби побудувати справжню системи автоматизації перекладу були здійснені у 1950-их роках. У ранніх системах використовувались великі двомовні словники та закодовані вручну правила для визначення порядку слів у висхідному продукті. У результаті цей метод було визнано занадто обмеженим, а завдяки тогочасному розвитку лінгвістики для покращення якості перекладу було запропоновано дослідження генеративної лінгвістики та трансформаційної граматики. [1]

Також до недавніх часів був поширеним статистичний переклад. Він використовував розподіл ймовірностей для визначення, що слово (або фраза чи інша семантична одиниця) у вихідному реченні є перекладом іншого слова у

вхідному. На початку свого існування, з 2006 року, Google Translate базувався саме на статистичному методі машинного перекладу. [2]

Як статистичні, так і нейронні системи можуть бути побудовані як на перекладі між двома мовами, так звані білінгвістичні, так і на перекладі з кількох мов - мультилінгвістичні, що базуються на приведенні вхідних речень до логічної штучної мови, що має набір семантичних примітивів, спільних для усіх мов. У цій роботі буде розглянуто виключно білінгвістичні системи.

Суттєвим привілеєм нейронних мереж над системами перекладу, заснованими на правилах, є більша гнучкість у виборі семантичних об'єктів для навчання. Для rule-based систем існує необхідність вручну позначати частини мови, інші граматичні правила, які властиві певній мові - очевидно, що граматичні норми однієї мови можуть суттєво розходитись з іншою (порівняти, наприклад, англійську та арабську), отже для кожної пари мов для перекладу потрібно продумувати свою систему – проаналізувати всі правила, винятки, вказати на більш уживані та доречніші варіанти, не забути про фразеологізми, щоб переклад виглядав подібно до живої мови.

Все це робить неможливим узагальнення роботи системи на переклад інших мов. Проте, це також демонструє переваги нейронних мереж, яка може виявити усі ці закономірності впродовж тривалого навчання.

Розділ 1. Дослідження предметної області

1.1 Проблеми перекладу

Через неоднозначність, що трапляється у текстах на морфологічному, синтаксичному та семантичному рівнях, навіть гарно навчена система може згенерувати переклад, що буде мати зовсім інший зміст, аніж оригінал.

Однією з причин існування семантичної неоднозначності є наявність омонімів. Наприклад, bank – банк та берег.

Приклад морфологічної неоднозначності - англійське слово sibling – гендерно-нейтральне, українською мовою може бути перекладене як брат або сестра в залежності від контексту.

Однакові структури використовуються для різних граматик:

Since you suggested it, I now must deal with it.

Оскільки Ви запропонували це …

Since you suggested it, we have been working on it.

З того моменту як ти запропонував це ….

За наявності особливого контексту, слова, що були відсутні але малися на увазі у вхідному реченні, мають бути присутні у коректному перекладі:

Having said that, I see the point. – У цьому реченні наявне неявне протиставлення, отже, переклад має містити одне з таких слів як «хоча», «проте», «незважаючи на» і т.д.

Хоча основною метою машинного навчання є генерація високоякісних перекладів, на практиці кращий результат можна завжди досягти під час пост-обробки вихідного тексту людиною. Покращення системи можна досягти також через обмеження змістового навантаження вхідних речень. Наприклад, переклад виключно хімічних статей, як приклад обмеження за темою, або переклад з відкоригованих текстів, що не містять омонімів, складної структури речень та полісемії.

1.2 Використання нейронних мереж

Хоча питанню автоматизації перекладу почали приділяти увагу ще в 50х, статистичні моделі панували у 1990х – 2010х, а нейронні мережі для цього почали активно використовувати лише з 2010х.

Ідея роботи нейронної мережі базується, як не дивно, на нейронах в мозку, що передають, оброблюють інформацію та роблять висновки за допомогою зв'язків з іншими нейронами через дендрити.

Штучні нейронні мережі беруть ідею комбінації сигналів, привласнюючи їм певні ваги, функцію активації та вихідне значення. У мережі нейрони компонуються по шарам, кожен з яких послідовно передає один одному дані, в той час як природні мережі мають набагато складніші шаблони взаємодії. Інтуїція за функцією активації нейронів – якщо нейрони отримують сильні скомбіновані сигнали, вони активуються та передають сигнали до інших нейронів.

Схема роботи одного нейрону штучної мережі:

Нехай n - кількість навчальних прикладів, m - кількість даних/ознак у одному прикладі.

На вхід подається матриця In_{nm} (n x m), один нейрон являє собою вектор ваг певних ознак v_m (розмірності m).

Результат – вектор Out_{n1} , де i-те значення відповідає i-тому прикладу, отримуємо після множення I на v_m - співставивши j-тій ознаці кожного прикладу j-ту вагу (j-тий елемент v_m) і просумувавши зважені ознаки.

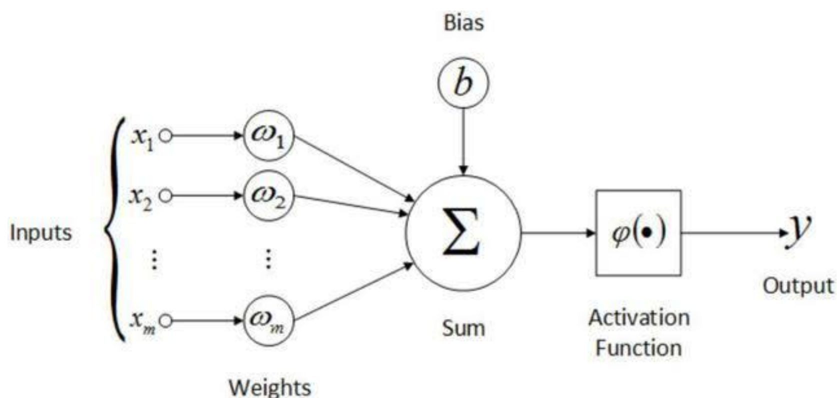


Рис.1.1

Перша й найпростіша з видів нейронних мереж є Feedforward. Вона названа так, оскільки в ній сигнали поширюються в одному напрямку, починаючи від вхідного шару нейронів, через приховані шари до вихідного шару і на вихідних нейронах отримується результат опрацювання сигналу. В мережах такого виду немає зворотніх зв'язків. [3]

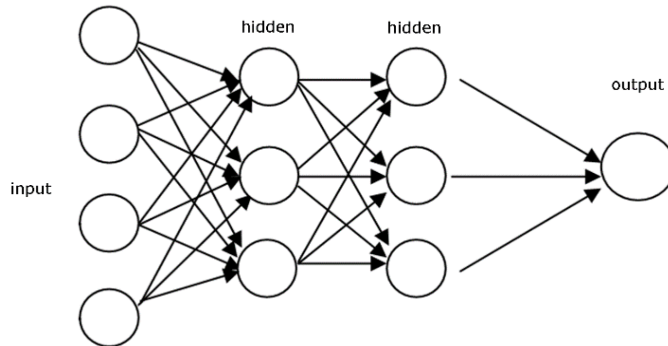


Рис.1.2

Один стовпчик нейронів називають одним шаром.

Маючи на вході дані In_{nm} на виході з шару W_{km}, що містить k нейронів, кожен з яких обов'язково є вектором v_m, отримаємо матрицю $Out_{nk} = In_{nm} * W_{km}$, де кожному прикладу замість m буде співставлено k ознак.

Оскільки якщо робити таким чином, то результат проходження даних через кілька матриць W_1, $W_2 \ldots W_l$ можна замінити на одну матрицю, що є їх добутком, то для вивчення більш складних зв'язків, ніж може надати один шар, після кожного шару результат проходить через певну функцію активації, що надає обчисленню нелінійності.

1.3 Оцінювання згенерованого перекладу мережі

Частіше за все як основну метрику оцінювання якості роботи мережі використовують точність – кількість правильно визначених значень з усіх отриманих. Для перекладу зазвичай така метрика не є найвдалішою. Метрика BLEU score (Bi-Lingual Evaluation Understudy)[19] була розроблена спеціально для оцінювання прогнозів, зроблених системами автоматичного машинного перекладу.

Для кожного унікального слова з передбаченого речення вона рахує мінімум з кількості повторів слова в передбаченому реченні та очікуваному перекладі.

Очевидна різниця та недолік BLEU score – ця метрика не бере до уваги розташування слова в реченні, що в реальному житі може повність змінити значення речення. Інший недолік метрики – вона надає перевагу меншим реченням. Наприклад, передбачене речення з одного слова, що наявне в очікуваному перекладі, дасть ідеальну оцінку 1. Оскільки метрика лише підраховує кількість співпадінь слів, то вона погано оцінить гарний переклад, який близький до очікуваного результату по змісту, але використовує інші слова для його передачі.

Попри всі недоліки цієї метрики BLEU score залишається основною метрикою багатьох дослідників з питань машинного перекладу. Наразі нібито досконаліші показники якості перекладу, такі як METEOR[20] та LEPOR[21], не набрали такої популярності, оскільки BLEU, попри всі свої недоліки, є більш зрозумілою метрикою, а отже й більш надійною, особливо якщо використовувати її разом з оцінкою експертів. [4]

Розділ 2. Представлення даних

У цьому розділі описані основні етапи попередньої обробки та видозмінення джерела даних перед тренуванням кожної з нейронних мереж, що будуть розглянуті далі.

2.1 Обробка тексту

Для тренування мережі потрібен великий об'єм текстових даних. Зрозуміло, що не всі дані з вибірки можуть відповідати певному вигляду, який ми очікуємо для подальшої взаємодії з ними. Перш за все потрібно застосувати певні техніки для очищення та трансформації текстового корпусу. У практичних дослідженнях було застосовано наступні перетворення:

1. Привести речення до UTF-8 формату, замінити всі літери з діакритичними знаками та дифтонги на їх відповідники з латиниці.
2. Вилучити пунктуаційні знаки, крім ‘,’, ‘.’, ‘!’, ‘?’, для того, щоб авторська пунктуація не впливала на переклад.
3. Прибрати великі літери
4. Додати на початок та кінець речень токени початку та кінця.

2.2 Токенізація

2.2.1 Word-based tokenization

Наступним етапом є розбиття кожного речення на окремі лексеми.
Для отримання більш швидких результатів зі списку закодованих речень прибирають ті, у яких довжина перевищує певне порогове значення.
Після цього кожній лексемі співставляється певне унікальне числове значення – індекс у створеному словнику.

2.2.2 Character-based tokenization

Ще один спосіб представлення вхідного речення[22] – розглядати його як послідовність символів. Таким чином, словник токенів, що буде складатися лише з літер, чисел та знаків, виявиться значно меншим, що призведе до значної оптимізації роботи мережі. Також при такому підході мережа буде здатна інтерпретувати слова, які не бачила під час тренування, проте також зможе генерувати нові форми слів, що можуть виявитися некоректними. Такий переклад почне стрімко погіршуватись зі зростанням довжини речень через більшу кількість токенів, що потрібно передбачити.

2.2.3 Subword tokenization

Моделі нейронних мереж використовують словник фіксованого розміру для обробки мови. Оскільки потенційний простір словникового запасу величезний, особливо для завдання нейронного машинного перекладу, а ресурси обмежені, модель не зможе якісно перекласти речення, в яких буде використано багато невідомих слів. Тому інколи для кодування речень використовують subword encoding[17] – розділення слів у реченні на послідовності символів (subword units), що часто зустрічаються у всьому тексті.

Найбільш поширений підхід для створення словника таких subword units називається byte pair encoding. Алгоритм наступний:

1) На вхід подається великий корпус текстів.
2) Кожне слово в корпусі розділяється на літери, вони додаються в словник
3) Найчастіші комбінації літер об’єднуються та додаються в словник
4) Попередній етап повторюється певну кількість ітерацій
5) Врешті-решт слова з найбільшою частотою повторів так само потрапляють до словника.

Якщо модель зустріне незнайоме слово, вона також зможе його обробити, оскільки завжди можна розділити слово таким чином, щоб його частини

належали такому словнику, хоча б посимвольно. Недоліком такого підходу є дуже великий обсяг сформованого словника.

В Tensorflow[23] є свій клас, для такої токенізації tensorflow_datasets.features.Text.SubwordTextEncoder

Приклад роботи:

```
encoder = SubwordTextEncoder.build_from_corpus(
 ['Hello world'], target_vocab_size=258)
encoder.encode('H')             # [75]
encoder.encode('Hello')         # [75, 104, 111, 111, 114]
encoder.encode('Hello ')        # [2]
encoder.encode('world')         # [1]
encoder.encode('world ')        # [1, 35]
```

Що характерно, у об'єкта цього класу в словнику завжди знаходяться усі символи, тому навіть якщо подати на вхід слово, символів якого не було у корпусі, з якого енкодер будувався, на виході все одно отримаємо прийнятний результат.

```
encoder.encode('say ?')  ( [118, 100, 124, 35, 66]
```

2.3 Embeddings

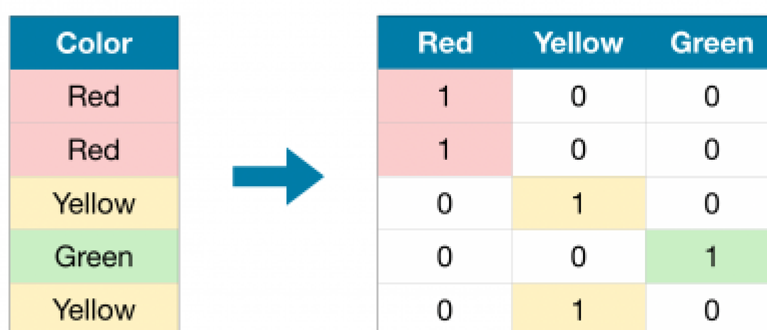
Після токенізації вектори речень доповнюють до максимального речення для об'єднання в батч – 2+ вимірні матриці. Зазвичай, за перший вимір відповідає кількість екземплярів.

Наразі вхідні дані представлені 2-вимірними батчами, 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[24]. Для кожного токену будується вектор розмірності словника, заповнений нулями на всіх позиціях, крім індексу самого токену. Таким чином відстань між будь-якими двома токенами є однаковою й дорівнює 1. Такий підхід є ідеальним для використання разом з символьною токенізацією. Проте він не підходить, коли за токен вважається ціле слово або його частини через велику розмірність таких векторів, яка катастрофічно впливає на швидкість обчислень.



Color
Red
Red
Yellow
Green
Yellow

Red	Yellow	Green
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0

Рис.2.1

Ембеддінги є значно потужнішим і більш релевантним способом закодовування слів. Це вектори з числовими даними певної розмірності (набагато меншої за розмір словника), значення яких можуть вираховуватись у процесі тренування нейронної мережі або заздалегідь. Концептуально кожне поле такого вектору відповідає за певну ознаку. Вектори синонімічних слів будуть мати великий добуток. Для ембеддінгів гарною рисою вважається властивість зберігати семантичну відстані 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». $|v_{queen} - v_{woman}| \approx |v_{king} - v_{man}|$

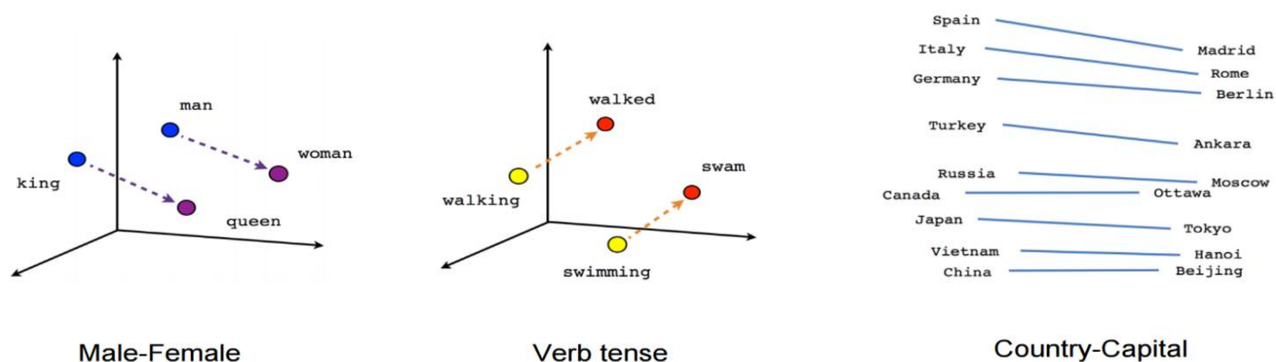


Рис.2.2

В певних випадках ембеддінги обмежуються для усунення расової/статевої/іншої дискримінації, що може трапитися у вхідних текстах.

Також популярною є практика використання претренованих ембеддінгів, потрібно лише переконатися, що слова з власного тексту є в словнику цих ембеддінгів, в іншому випадку для таких слів ембеддінги можуть виявитися невдалими.

2.4 Створення вхідних та вихідних даних для моделі

Є різні принципи обрання даних, на яких за крок тренується модель

- стохастичний – використання одного екземпляру
- батч - використання всього набору даних на кожному кроці
- мінібатч – використання певної кількості екземплярів з батчу

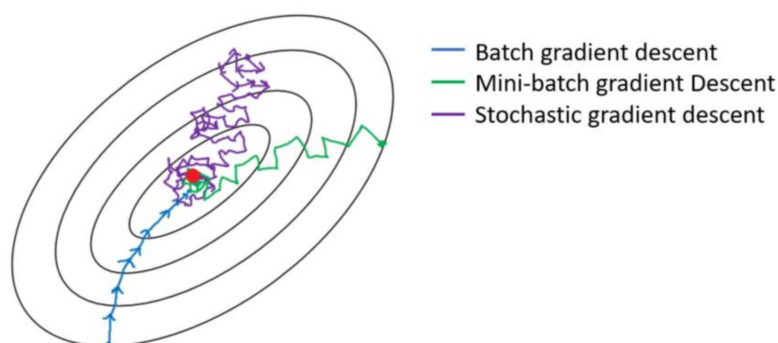


Рис.2.3

Як видно з малюнку, стохастичний градійний спуск на кожному кроці може мати велике відхилення. Батч та мінібатч узагальнюють результати по своїм наборами даних, тому відхилення набагато менше та модель швидше

досягає оптимального результату. Проте, батч дуже неефективний через множення матриць на всю вибірку, тому на практиці найчастіше застосовують мінібатч.

Розмір таких мінібатчей є одним з гіперпараметрів моделі.

Після виділення мінібатчів вхідні та очікувані дані мають розмірність

$Input.shape = Output.shape = (\text{к-ть елементів у мінібатчі, довжина речень})$

2.5 Розділення на вибірку для тренування, валідації та тестування моделі

На вибірці для тренування модель вираховує оптимальні коефіцієнти. Вибірка для валідації потрібна, щоб оцінити моделі з різними гіперпараметрами, перевірити, що моделі не перенавчаються на даних для тренування(тобто, що модель здатна працювати і на даних, які вона бачить вперше).

Вибірка для тестування потрібна, щоб зробити кінцеву оцінку моделі, яку ми обрали.

Для розділення даних можна використовувати функцію з бібліотеки scikit-learn.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.33)
```

Розділ 3. Основна концепція побудови моделей для машинного перекладу

3.1 Архітектура RNN

3.1.1 Загальний опис

Одна з найбільш відомих моделей для роботи з текстовими (і не тільки) послідовностями є Рекурентні Нейронні Мережі – RNN[25]. На відміну від більш базових Feedforward Network, основною характеристикою RNN є наявність кількох кроків(часових позначок, таймстепів), на кожному кроці мережа має 2 входи і два виходи , один з входів та один з виходів з'єднується з виходом з попереднього кроку та входом на наступний крок відповідно, а інший вихід відповідає за передбачений на цьому кроці вектор. Входи та виходи, що з'єднуються між собою мають назву hidden state та слугують певною пам'яттю між кроками. Таким чином мережа моделює передбачення не лише на поточних даних, а й на узагальненню з даних з минулих кроків.

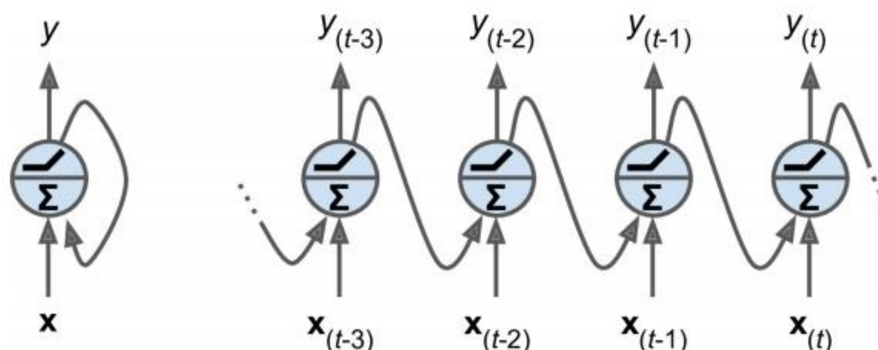


Рис.3.1

Є 3 типи RNN:

1. послідовність у послідовність — sequence to sequence, Seq2Seq
2. вектор у послідовність — vector to sequence
3. послідовність у вектор — sequence to vector

Другий тип для генерації послідовності на кожному кроці передає той самий вектор на вхід, третій тип – ігнорує всі передбачення на кожному кроці, окрім останнього.

Наостанок, є ще одне застосування RNN у вигляді моделі, що об’єднує мережу типу «послідовність у вектор», яку називають Енкодером, з мережею «вектор у послідовність», Декодером. Для перекладу зазвичай використовують таку мережу, яка через Енкодер вхідну послідовність перетворює у векторне представлення змісту, в той час як Декодер перетворює цей вектор на послідовність слів вихідної мови.

Використання моделі Енкодер-Декодер набагато краще, ніж «послідовність у послідовність», оскільки вона не ігнорує те, що останні слова в реченні можуть впливати на переклад перших слів, отже, потрібно проаналізувати все речення цілком перед його перекладом.

3.1.2 Проблеми RNN

- ***Вибухання коефіцієнтів***

Перша проблема може виникнути, якщо у якості функції активації використовувати ReLU ($f(x) = max(0, x)$). Нехай вагові коефіцієнти матриць після батчу були оновлені таким чином, що значення вихідних даних на першому кроці трохи зросли. Оскільки ті самі коефіцієнти використовуються на кожному кроці, з кожним кроком значення на виході може зрости до занадто великих чисел. Щоб уникнути такого, слід використати менший learning rate або функцію активації, що обмежує значення, наприклад tanh.

Таким же чином можуть вибухнути й значення самих коефіцієнтів, це потрібно відслідковувати і робити обмеження до мінімального та максимального встановленого значення.

- ***Затухання коефіцієнтів***

Через особливості алгоритму backpropagation[5], що всюди використовується для НМ для РНН характерна проблема затухання градієнтів. Через це після кількох кроків інформація може загубитися.

Це відображено на наступному рисунку через різну пропорційність кольорів, кожен з яких відображає вплив відповідного слова на передбачення, на кожному кроці:

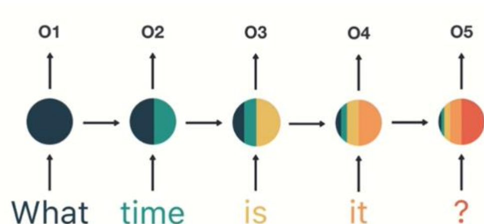


Рис.3.2

Це є так звана проблема короткочасної пам'яті, якої можна певною мірою уникнути використовуючи модифіковані версії РНН.

3.2 Архітектура LSTM

LSTM (Long Short Term Memory networks)[x] були створені спеціально для того, аби вирішити проблему відсутності довготривалої залежності між даними на різних кроках у звичайної RNN. Їх дизайн був змодельований, щоб більш гнучко керувати залежністю між даними з минулих кроків на результат поточного кроку.

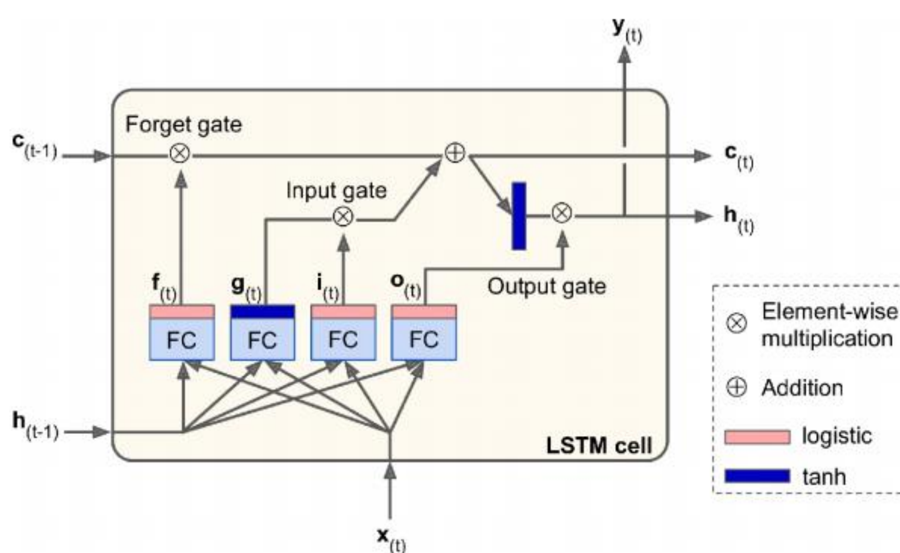


Рис.3.3

Можна побачити, що, на відміну від RNN, LSTM на вхід приймає не тільки стан з минулого кроку, а й вектор передбачень з минулого кроку.

Основна особливість LSTM – наявність трьох вентилів (gates), кожен з яких має своє призначення.

Forget gate приймає на вхід поточний вхідний вектор та минулий вектор результату та вираховує, що з минулої пам’яті слід забути.

Таким же чином діє input gate, проте він навпаки вирішує, яка частина поточних вхідних даних буде зберігатися у пам’яті.

Аналогічно, output gate на основі вхідних даних вирішує, що зі стану клітини потрапить на вихід.

При цьому для вентилів використовується функція активації софтмах ($softmax(x) = \sigma(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$), яка повертає вектор значень від 0 до 1 (для деякого набору значень, що містяться у векторі, за умови, що більше значення відповідає більшій ймовірності, софтмах поверне розподіл цих ймовірностей).

Для даних що проходять через вхідний та вихідний вентиль перед цим використовується tanh, що відображає їх значення на проміжок від -1 до 1.

3.3 Архітектура GRU

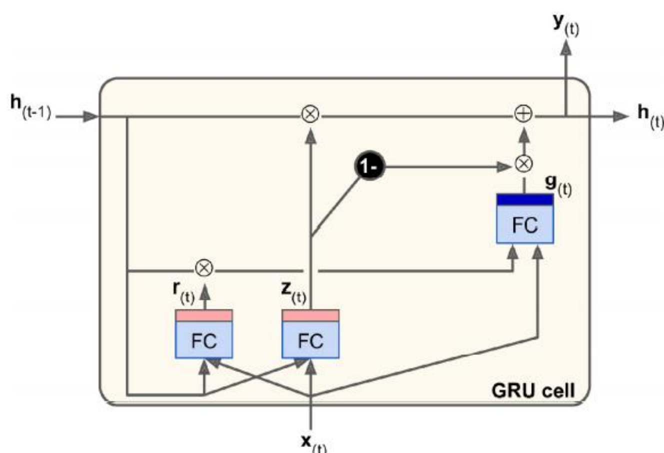


Рис.3.4

GRU (Gated Reccurent Unit)[26] - спрощена версія LSTM, у якої знову лише 2 входи та виходи.

Замість двох використовується один вентиль update gate, що працює як forget gate та заперечений input gate. На вихід подається та сама інформація, що й на наступний крок. Замість цього reset gate вирішує, яка частина стану за минулі кроки потрібно відкинути.

3.4 Архітектура Bidirectional RNN

Замість звичайної RNN/GRU/LSTM для Енкодера часто використовують Bidirectional RNN[27]. 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.

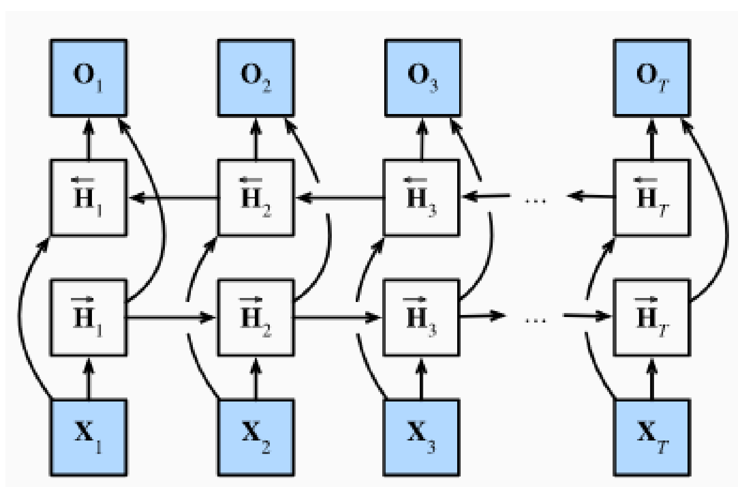


Рис.3.5

Така мережа складається з двох РНН, одна з яких рахує всі hidden state, як раніше, у прямому порядку, а інша - у зворотному. Після цього, щоб отримати результуючий вектор на кожному кроці обидва hidden_state об'єднуються в один, що потім проходить через функцію активації. Оскільки для моделі Енкодера потрібно повертати лише її стан, то на вхід Декодера подаються об'єднані стани останніх кроків прямої РНН (для останньої лексеми) та зворотньої (для першої лексеми).

3.5 Архітектура Transformers

3.5.1 Загальний опис

На відміну від RNN, в Трансформерах[28] широко застосовується механізм уваги та така мережа гарно працює навіть з довгими послідовностями. Трансформер розділяється на Енкодер та Декодер. В обох моделях є кілька блоків, через які проходять дані, в кожному блоці є шари уваги (Attention layers). Мережа не рекурентна, тому проблеми з градієнтами, характерні для RNN, не настільки суттєві. Також внаслідок незалежності блоків один від одного, можна виконувати обчислення паралельно.

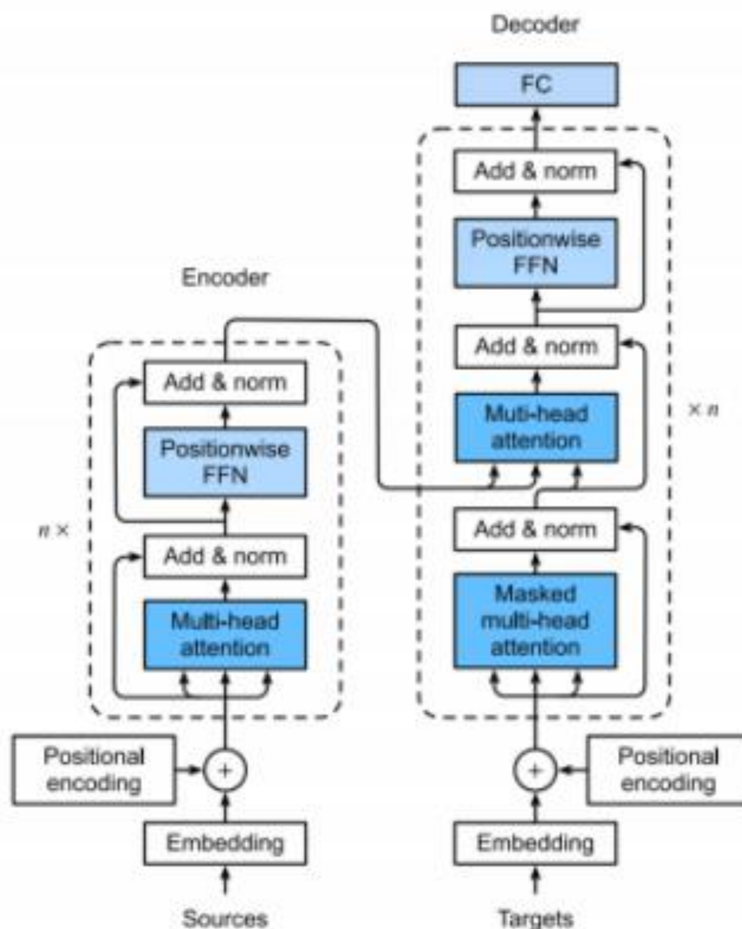


Рис.3.6

Хоча Трансформер є більш простою моделлю для обчислення, її архітектура має більше концептуальних нюансів та тонкощів в реалізації.

З нових речей, що представлені на діаграмі моделі можна виділити позиційне кодування та (замасковані) шари Multi-Head Attention.

3.5.2 Позиційне кодування

Оскільки у самій моделі відсутній покроковий механізм обробки кожного наступного токену, що характерний для RNN, тому замість нього використовується позиційне кодування, аби додати до ембеддінгу кожного слова інформацію про його розташування в реченні.

Формули для позиційного кодування наступні:

$$PE(pos, 2*index) = \sin\left(\frac{pos}{10000^{\frac{2index}{d_{model}}}}\right)$$

$$PE(pos, 2*index+1) = \cos\left(\frac{pos}{10000^{\frac{2index}{d_{model}}}}\right)$$

3.5.3 Маскування

Для того щоб модель не брала до уваги доповнення речень нульовими символами до потрібної довжини (paddings) робиться маскування. Для цієї роботи був вибраний наступний спосіб – повертається вектор розмірності вхідного речення, на всіх ненульових позиціях – 0, на нульових – 1. Таким чином маскування досягається, якщо помножити цей вектор на дуже велике від'ємне число і додати до вектору, що ми хочемо замаскувати – після функції активації softmax на місці великих від'ємних значень будуть 0.

Для того, щоб модель при роботі зі словом на певній позиції не звертала увагу на слова на наступних позиція також робиться маскування. Look-ahead mask повертає квадратну матриця розміру довжини речення, кожен рядок якої відповідає за відповідну маску для і-того слова – ненульові позиції відповідають словам, що йдуть після цього слова. Після цього ця маска використовується так само як і padding mask.

3.5.4 Scaled dot product attention

Коли моделі Трансформер передають речення, ваги уваги між усіма лексемами обчислюються одночасно. Вузол уваги створює ембеддінги для кожної лексеми в контексті, який містить інформацію не лише про саму лексему, але й про зважену суму інших релевантних лексем. [6]

Модель при навчанні для кожного свого блоку з Механізмом уваги визначає коефіцієнти для трьох матриць Query, Keys та Values.

$$Attention(Q,K,V) = \sigma\left(\frac{QK^T}{\sqrt{d_k}}\right) * V$$

Інтуїтивно цю формулу можна зрозуміти, замінивши множення матриць на ітерування по кожній лексемі.

Для кожної лексеми x вираховується вектор запиту $q_x = x*Q$, вектор ключа $k_x = x*K$, вектор значення $v_x = x*V$.

Вага релевантності a_{ij} між i-тою та j-тою лексемами є скалярним добутком q_i та k_j. Тобто вага релевантності визначає, наскільки ключ j-тої лексеми підходить до запиту, що створений i-тою лексемою.

Повертаючись до матриць, потім ці ваги ділять на $\sqrt{d_k}$ – корінь розміру вектору ключа, що стабілізує градієнти під час тренування, та визивають функцію softmax для визначення долі кожного з можливих значень при обчислені результату.

Приклад роботи:

Запит: [10, 10, 0] – Потрібно більшу уваги приділити таким значенням V, ключі K яких більше збігаються з запитом.

Матриця ключів:

```
[[10, 0, 0],
 [ 0,10, 0],
 [ 0, 0,10],
 [ 0, 0,10]]
```

Матриця значень:

```
[[     1,0],
 [    10,0],
 [   100,5],
 [1000,6]]
```

Отже, отримаємо більші ймовірності для перших двох рядків-векторів, а саме: [[0.5, 0.5, 0, 0]],

Що показує, що обидва значення за цими ключами рівноймовірно можливі, а всі інші – неможливі.

Отримаємо результат: $[\left[\frac{1+10}{2}, 0\right]]$

3.5.5 Multi-head attention

Один набір матриць (Q, K, V) називають головою уваги (attention head), й кожен шар у моделі Трансформер має декілька голів уваги для того, щоб кожна голова могла навчитися звертати увагу на релевантні лексеми з різних поглядів на релевантність. Наприклад, є голови уваги, що для кожної лексеми звертають увагу переважно на наступне слово, або голови уваги, що переважно звертають увагу дієслів на їхні безпосередні об'єкти.. Результат є об’єднанням всіх векторів значень з різних інтерпретацій релевантності.

Розділ 4. Практична частина досліджень

4.1 Визначення фреймворку для роботи з нейронними мережами

Як мову для реалізації системи машинного перекладу було обрано Python, оскільки вона є найбільш популярною для роботи з нейронними мережами. У практичній частині цієї роботи на меті ставився розгляд якомога більшої кількості різних моделей, а не повністю самостійна їх імплементація, тому наступні моделі будуть побудовані з використанням API одного з фреймворків.

Ще однією причиною не писати код, використовуючи виключно бібліотеку для зручної роботи з векторами та матрицями numpy(а тим паче без неї) є те, що зараз для нейронних мереж активно використовують графічний прискорювач GPU, що часто пришвидшує роботу в 50 та більше разів, який не використовується в numpy.

Оскільки сфера штучного інтелекту стрімко розвивається, не дивно, що існує безліч фреймворків з готовими реалізаціями різних шарів, моделей та інших функцій для роботи з нейронними мережами.

Було переглянуто 3 основних – PyTorch, MxNet та TensorFlow.

В першу чергу вибір пав на PyTorch та MxNet. В статтях [7], де порівнювались три фреймворки, як основний недолік TensorFlow наводили використання статичних графів, тоді як PyTorch та MxNet підтримують динамічні. Чим це гірше? В статичних графах параметри та операції спочатку оголошуються, потім виконуються, їх неможливо змінити під час виконання програми. Обмежується процес налагодження програми: доступний лише кінцевий результат, але не результати проміжних етапів.

Оскільки MxNet порівняно молодий фреймворк для нього ще не сформувалось достатньо велика спільнота, щоб мати досить прикладів роботи з ним. PyTorch по характеристикам та по API досить схож на MxNet. Тому перші

спроби написати нейронну мережу були з використання цього фреймворку. Проте, приклад мережі для генерації тексту з офіційного сайту PyTorch займав значно більше часу, ніж хотілося, тому замість одного відомого фреймворку потім було розглянуто інший, не менш відомий TF.

Виявилось, що в TensorFlow 2.0 підтримуються, як статичні, так і динамічні графи. Для нього написано велику кількість посібників та документації, він підтримується великою спільнотою розробників і технічними компаніями. Головне - приклади з офіційного сайту виконуються швидко.

До того ж фреймворк пропонує потужний засіб моніторингу процесу навчання моделей і його візуалізації – Tensorboard [8]. Разом з TF використовується Keras – мінімалістична бібліотека, що працює на базі TF, яка має не тільки гарний API для роботи з нейронними мережами, а й окремі модулі спеціально для роботи з текстом, набором даних (датасетом) тощо.

4.2 Пошук тренувальної вибірки

У якості даних використовувались датасет з сайту https://www.tensorflow.org/datasets, а саме такий, для якого характерна більша за середню (~20 слів) довжина речень та набір коротких фраз з застосунку http://www.manythings.org/anki.

Спочатку було проведено кілька дослідів з короткими реченнями.

Вхідне речення подається англійською мовою, на вихід – французькою. Довжина таких речень до 10 слів.

Для більш складних моделей розглядаються важчі випадки з довгими реченнями – на вхід – англійська мова, на вихід – руминська.

Середня довжина речень – 30 слів. Трапляються речення з довжиною в 100 слів та більше.

4.3 Дослідження роботи моделей типу RNN

4.3.1 Основні моменти побудови моделі RNN у програмному коді

Наступний код відповідає реалізації RNN, використовуючи функціональний інтерфейс моделі TF:

```
encoder_rnn = RNN(latent_dim, return_state=True)
decoder_rnn = RNN(latent_dim, return_state=True,
return_sequences=True)

encoder_inputs = Input(shape=(None,))
x = Embedding(input_vocab _size, embedding_size)(encoder_inputs)
encoder_outputs, encoder_state = encoder_rnn(x)

decoder_inputs = Inputs(shape=(None,))
y = Embedding(target_vocab_size, embedding_size)(decoder_inputs)
decoder_outputs, decoder_state =
decoder_rnn(y,initial_state=encoder_state)

res = Dense(target_vocab_size,
activation='softmax')(decoder_outputs)

model = Model([encoder_inputs, decoder_inputs], res)
```

Декілька уточнень стосовно наведеного коду:

- embedding_size – розмірність векторів ембеддінгу
- latent_dim – розмірність вектору, що повертається на кожному кроці RNN
- RNN(return_state=True) – RNN поверне стан після останнього кроку, false за замовчуванням
- RNN(return_sequences=True) - RNN поверне матрицю з вихідних векторів на кожному кроці, а не тільки останній вектор, false за замовчуванням
- RNN(embedding_size) – розмірність векторів, що повертає RNN буде дорівнювати розмірності ембеддінгу

- Dense(target_vocab_size, activation=’softmax’) – перетворення векторів ембеддінгу на вектори розмірності всього словника Через використання softmax під номером кожного слова буде стояти ймовірість вектора відповідати цьому слову
- На кожному і-тому кроці RNN декодера має визначити значення і-тої лексеми перекладу, отже на вхід йому подається (і-1) лексема перекладу, тому decoder_input подається як START_TOKEN + переклад, а лейбл для порівняння – як переклад + END_TOKEN (використовується техніка teacher-forcing learning)
- Заміна звичайної RNN на інші її види робиться наступним чином

<table>
<tr><td>GRU</td><td>

```
RNN(…) -> GRU(…)
```

</td></tr>
<tr><td>LSTM</td><td>

```
RNN(…) -> LSTM(…)

out, state = enc_rnn(inp) -> out, stateh, statec =
enc_lstm(inp)

_ = dec_rnn(inp, state)-> _ = dec_lstm(inp, [stateh,
statec])
```

</td></tr>
<tr><td>Bidirectional RNN</td><td>

```
enc = RNN(…) -> enc = Bidirectional(RNN(…)),

dec = RNN(latent_dim) -> dec = RNN(2*latent_dim)

enc_out, state = enc_rnn(inp) ->
enc_out, forward_state, backward_state = enc_rnn(inp)

_ = dec_rnn(inp, state) -> _ = dec_rnn(inp
Concatenate()([forward_state, backward_state]),
```

</td></tr>
<tr><td>Char-level RNN</td><td>прибрати шари ембеддінгу, у якості vocab_size використовувати кількість символів алфавіту вхідної/вихідної мови, перед цим речення розбити на символи та закодувати їх через one-hot encoding</td></tr>
</table>

Наступний код відповідає реалізації перекладу на основі такої моделі:

```
input_seq = input_tokenizer.encode(input_sentence)
encoder_state = encoder_rnn.predict(input_seq)

stop_condition = False,
decoder_input = [[[ target_tokenizer.encode(START_TOKEN) ]]]
predicted_sentence = ''
decoder_state = encoder_state
while True:
     decoder_out, decoder_state =
decoder_rnn.predict(decoder_input + [decoder_state])
     predicted_ind = argmax(decoder_out[0])
     predicted_token = target_tokenizer.decode(predicted_id)
     predicted_sentence += predicted_token + ' '

     if target_tokenizer.decode(decoder_out) == END_TOKEN or
len(predicted_sentence) >= max_sentence_length:
          break
```

Один з варіантів підготовки вхідних даних з використанням tf.data.Dataset – конвеєру, що ітерує по даним таким чином, щоб не зберігати їх повністю в пам'яті:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import pad_sequences

...
def split_two(enc_inp, tar):
    dec_inp = tar[:-1]
    target = tar[1:]
    return (enc_inp, dec_inp), target



input_sequences = input_tokenizer.encode(input_sequences)
```

```
target_sequences = map(lambda x: START_TOKEN + ' ' +x+ ' '
END_TOKEN, target_sequences)


input_sequences = pad_sequences(input_sequences, padding='post')
target_sequences = pad_sequences(target_sequences, padding='post')

dataset_input =
tf.data.Dataset.from_tensor_slices(tf.cast(input_sequences,
tf.int64))
dataset_target =
tf.data.Dataset.from_tensor_slices(tf.cast(target_sequences,
tf.int64))
dataset = tf.data.Dataset.zip((dataset_input, dataset_target))


dataset = dataset.map(split_two)
dataset = dataset.shuffle(buffer_size).padded_batch(batch_size)
dataset = dataset.prefetch(tf.data.experimental.AUTOTUNE)
```

Уточнення:

- dataset.prefetch використовується для передчасної вибірки батчів з вхідних даних до того, як вони будуть використані,

  AUTOTUNE автоматично налаштовує кількість батчів такої вибірки

4.3.2 Результати RNN

В першу чергу була досліджена робота моделей RNN та LSTM на символьному рівні на коротких реченнях.

LSTM

4s 30ms/step - loss: 0.2351 - accuracy: 0.9902 - val_loss: 0.4399 - val_accuracy: 0.9200

RNN

3s 57ms/step - loss: 0.3238 - accuracy: 0.9344 - val_loss: 0.5731 - val_accuracy: 0.8879

Як видно, навіть з зовсім невеликими реченнями LSTM справляється трохи краще. Різниця стане більш помітною при використанні обох мереж на рівні слів. Було досліджено, як різні оптимізатори впливають на роботу LSTM

У додатку А наведено результати роботи моделі з такими оптимізаторами, як:

- Adam optimizer
- SGD
- Adagrad
- Rmsprop

Як видно за графіками, найкраще було б далі працювати або з Adam , або з Rmsprop. Другий показав трохи кращі результати, тому саме він використовується в наступних тестах. Оптимізатор Adam є скомбінованою версією Adagrad та RMSProp, отже на кращі його результати сприяла саме складова від RMSProp, характеристикою якого є здатність уповільнювати learning rate таким чином, щоб через великий коефіцієнт оновлення (learning rate) ваги моделі не пропустили ті значення, при яких досягається глобальний мінімум.

Також на цих графіках можна побачити, що на даних для тренування, модель працює набагато краще, ніж під час валідації, що свідчить про перенавчання (overfitting) моделі, тобто, що вона не узагальнює дані, а пристосовується саме до даних з тренування, моделює закономірності саме в цих даних. Далі цей ефект було певною мірою подолано, використовуючи dropout та регуляризацію.

Наступним етапом було проведено пошук кращих значень для одного з гіперпараметрів моделі - latent_dim, що відповідає за розмірність вихідного вектору ембеддінгу та вектору стану у RNN:

При latent_dim = 32:

```
loss: 0.3750 - accuracy: 0.8881 - val_loss: 0.4880 - val_accuracy: 0.8582
```

При latent_dim = 128:

```
loss: 0.1370 - accuracy: 0.9581 - val_loss: 0.5652 - val_accuracy: 0.8702
```

При latent_dim = 640:

```
loss: 0.0255 - accuracy: 0.9884 - val_loss: 0.7206 - val_accuracy: 0.8768
```

Пошук також було автоматизовано з використанням одного з тьюнерів, а саме kerastuner, що використовує метод random search[9] для пошуку набору гіперпараметрів, що дасть близькі до найкращих результати.

Як очевидно, зі зростанням розмірності векторів модель стає здатна запам'ятовувати та закодовувати більше інформації, а отже результати її роботи виявляються кращими.

На рис 4.1 наведено результати роботи моделі LSTM з розмірністю latent_dim 640 юнітів.

```
Input sentence: ['y o u   a r e   g o o d .']
Decoded sentence with LSTM : vous êtes bons.>
Decoded sentence with RNN  : tu es bon.>
-
Input sentence: ['w e   b r o k e   u p .']
Decoded sentence with LSTM : nous avons éclaté de rire
Decoded sentence with RNN  : nous nous sommes séparés.
-
Input sentence: ['b e   p r e p a r e d .']
Decoded sentence with LSTM : soyez préparé !>
Decoded sentence with RNN  : soyez prêtes !>
-
Input sentence: ["i t ' s   a l l   r i g h t ."]
Decoded sentence with LSTM : ce n'est pas grave.>
Decoded sentence with RNN  : ce n'est pas grave.>
```

Рис. 4.1

Далі було здійснене порівняння моделей LSTM та GRU. Для цього було використано рівень слів. Також можна порівняти LSTM на рівні слів та символів і дійти до висновку, що на коротких реченнях також краще використовувати рівень символів.

RNN

```
Epoch 49/50
125/125 [==============================] - 20s 158ms/step - loss: 0.1901 - bleu_score: 0.8526
Epoch 50/50
125/125 [==============================] - 20s 159ms/step - loss: 0.1858 - bleu_score: 0.8532
```

LSTM

```
Epoch 49/50
125/125 [==============================] - 22s 173ms/step - loss: 0.1247 - bleu_score: 0.8642
Epoch 50/50
125/125 [==============================] - 21s 172ms/step - loss: 0.1203 - bleu_score: 0.8651
```

GRU

```
Epoch 49/50
125/125 [==============================] - 21s 171ms/step - loss: 0.1554 - bleu_score: 0.8653
Epoch 50/50
125/125 [==============================] - 21s 170ms/step - loss: 0.1563 - bleu_score: 0.8631
```

Bidirectional RNN

```
Epoch 49/50
125/125 [==============================] - 22s 179ms/step - loss: 0.1032 - bleu_score: 0.8510
Epoch 50/50
125/125 [==============================] - 22s 180ms/step - loss: 0.1022 - bleu_score: 0.8503
```

LSTM показала найкращі результати з трьох типів мереж. Проте ще кращі результати виявила Bidirectional LSTM, що більш точно вираховувала контекст речення.

Також була перевірена теорія, що якщо слова у вхідному реченні подавати у зворотному порядку, то модель з такими даними може показати кращі результати [11].

LSTM з оберненим порядком слів на вхід

```
Epoch 49/50
125/125 [==============================] - 23s 187ms/step - loss: 0.1240 - bleu_score: 0.8856
Epoch 50/50
125/125 [==============================] - 23s 186ms/step - loss: 0.1233 - bleu_score: 0.8833
```

Результати роботи LSTM:

```
Input sentence: you're my type .
Decoded sentence, model trained on 50 000 sentences:  tu es mon type .
Decoded sentence, model trained on 10 000 sentences:  tu es de mon âge
Expected sentence: tu es mon type . >
-
Input sentence: you're nervous .
Decoded sentence, model trained on 50 000 sentences:  vous êtes nerveux
Decoded sentence, model trained on 10 000 sentences:  tu es difficile
Expected sentence: vous êtes nerveux . >
-
```

Для оцінювання результату була викорисана метрика bleu-score. Tensorflow дозволяє визначити свої метрики, що будуть використані під час тренування моделі. На вхід функція для метрики повинна приймати два параметри – вхідний

батч та передбачений. Оскільки обчислення в Tensorflow виконуються з побудованого графу, то не всі підходи реалізації алгоритмів є можливими.
Зокрема такий наївний підхід для bleu-score, що використовував ітерацію по токенам для пошуку збігів довелось змінити на більш математичну реалізацію з використанням допоміжних функцій для роботи з векторами, що наявні у фреймворці. Обидва методи можна переглянути у додатку H.

На основі результатів тренування моделі у програмі від Tensorflow “Embedding Projector” [18] було зображено ембеддінги слів вхідної та вихідної мови. Результат можна переглянути у додатку F.
Оскільки розмірність таких векторів більше трьох, то для комфортного перегляду вони відображалися на 3 виміри використовуючи техніку PCA (Principal Component Analysis) – один з головних способів зменшити розмірність даних, втративши найменшу кількість інформації .

Наступним кроком було навчання моделі LSTM на довгих послідовностях.
Для руминського датасету з 10 000 пар речень середня довжина речень була 30, серед них було приблизно 300 речень з довжиною більше 70 слів.

Такі дані дуже повільно оброблялися, оскільки кожне речення потрібно було доповнити до максимального, тому довжина кожної вхідної послідовності токенів ставала 95, а на виході очікувалися послідовності довжини 114. На обробку однієї епохи потрібно було 2 хвилини, порівняно з 4 секундами для малих речень.

При такому підході все одно довелось обмежувати довжину речень, адже швидкість тренування моделі суттєво падала зі збільшенням довжини. Що характерно, точність перекладу зі збільшенням кількості речень також не збільшилась через необхідність моделі пристосовуватися до більшої кількості нових даних з новими словами, що з великою ймовірністю могли не мати схожості з даними з меншої вибірки.

4.3.3 Розгляд варіантів представлення вхідних даних при великій довжині речень

Крім цього для оптимізації роботи моделі на великих реченнях був розроблений ще один підхід, який був можливий внаслідок того, що Tensorflow дозволяє у якості параметру розміру вхідних даний на певних місцях вказувати відсутність фіксованого значення – None. Зазвичай це використовується для невизначеності у кількості прикладів в батчі, щоб модель могла потім повернути переклад для одного речення.

Насправді, ж можна два виміри залишити пустими, що дозволяє моделі на кожному кроці брати речення різної довжини, з єдиним обмеженням, що всередині батчу вони повинні бути фіксовані. Таким чином, якщо на вході маємо багато довгих речень, середня довжина яких 30, але зустрічаються й речення по 40, 50, то речення біль менш однакової довжини можна розбити по батчах.

На практиці при такому підході модель з даними впоралась трохи швидше, але через суттєве погіршення результатів на довгих реченнях загальна оцінка перекладу залишалась незадовільною.

4.4 Дослідження результатів роботи моделей типу Transformers

4.4.1 Основні моменти побудови моделі Transformers у програмному коді

Для визначення моделі було визначено свій клас, що наслідується від класу tf.keras.Model. Крім розглянутого раніше функціонального (Functional) стилю для створення моделі також може використовуватись послідовний (Sequence). Їх перевага перед стилем наслідування (Model API) у тому, що такі моделі легше зберегти, їх структуру можна продивитися до початку побудови динамічного графу, як показано у додатку Б. Фреймворк може визначати форми шарів та перевіряти типи, завчасно діагностуючи про помилки ще до моменту, як моделі були надані дані. Такі моделі визначаються статичними графами, тому їх відносно легко відладжувати.

Проте недоліком такого підходу є статичність – для опису алгоритмів роботи певних можливостей моделі потрібно використовувати цикли, шари, розмірність яких визначається динамічно, розгалуження та інші приклади динамічної поведінки [10].

Для того, щоб побудувати динамічну модель у Tensorflow, потрібно наслідувати від класу tf.keras.Model, проініціалізувати потрібні шари в конструкторі та перевизначити метод call для обчислення результату роботи моделі на одному батчі даних.

Також можна перевизначити низку інших методів, зокрема корисними можуть виявитися train_step – метод, що викликається на кожному кроці і має реалізувати всю логіку від знаходження вихідних даних (через метод call або взагалі його не використовуючи), визначення градієнтів через backpropagation, оновлення ваг шарів та обчислення значень усіх метрик. Аналогічно можна визначити інші дії для тестового набору даних через test_step.

Під час написання моделі в ООП стилі було використано декомпозицію – визначено ще два класи Model, окремо для Енкодера там Декодера. Оскільки вони ніяк не взаємодіють один з одним, крім даних на вході/виході з однієї моделі в іншу, така декомпозиція більше відповідає архітектурі самої мережі.

Також для більшого розділення відповідальності для різних дій, що виконують алгоритми, для яких не створено бібліотечних шарів, було визначено кілька власних шарів. Як і моделі, це можна реалізувати, наслідуючи від класу keras.layers.Layer та перевизначивши метод call. Ваги шарів, що використовує об’єкт класу Layer, автоматично відслідковуються при встановлені шарів як атрибутів такого об’єкту.

Було створено шар Multi-Head Attention, що застосовується в обох моделях Енкодеру та Декодеру, та шари блоків Енкодеру та Декодеру, що відповідають за один з кількох блоків, кожен з яких складається з Multi-Head Attention, FeedForward Network та Layer Normalization, згідно зі структурою моделі, яку наведено у додатку С.

Використовуючи малюнок з додатку, можна таким чином описати структуру моделі:

З погляду Трансформеру:

```
def call_train(self, inputs, training=False):
    encoder_input, decoder_input = inputs
enc_padding_mask, look_ahead_mask, dec_padding_mask =
create_masks(encoder_input, decoder_input)
encoder_output = self.encoder(encoder_input, training,
enc_padding_mask)  # (batch_size, inp_seq_len, d_model)

    dec_output, attention_weights = self.decoder( decoder_input,
encoder_output, training, look_ahead_mask, dec_padding_mask)

    # (batch_size, tar_seq_len, target_vocab_size)
    final_output = self.final_layer(dec_output)

    return final_output
```

- final_layer – той самий Dense шар з функцією активації softmax для вираховування ймовірностей отримати на виході кожне слово, що був застосований і для RNN

Що відбувається при виклиці енкодеру:

```
        def call(self, x, training, mask):
     seq_len = tf.shape(x)[1]

        # adding embedding and position encoding.
        x = self.embedding(x)   # (batch_size, input_seq_len,
d_model)
        x += self.pos_encoding[:, :seq_len, :]
        x = self.dropout(x, training=training)

        for i in range(self.num_layers):
            x = self.enc_layers[i](x, training, mask)

        return x  # (batch_size, input_seq_len, d_model)
```

- self.pos_encoding – заздалегідь праховані значення позиційного кодування для всіх позицій
- self.enc_layers – один шар зі списку відповідає одному блоку енкодеру

Для моделі декодеру код виглядає майже аналогічно

Код для реалізації одного блоку енкодеру/ декодеру:

- Спершу відбувається обчислення ваг уваги через MultiHead Attention. Для енкодеру використовується маска для ігнорування значень порожніх символів(padding mask), для декодеру – padding mask + маска для ігнорування значень наступних лексем

```
    attn_output, _ = self.mha(x, x, x, mask)
    attn_output = self.dropout1(attn_output, training=training)
    out1 = self.layernorm1(x + attn_output)
```

- Наступна частина характерна лише для блоку декодеру і відповідає другому шару MultiHead Attention з додатку E

```
    attn2, attn_weights_block2 = self.mha2(enc_output, enc_output, 
out1, padding_mask)
    attn2 = self.dropout2(attn2, training=training)
    out2 =  self.layernorm2(attn2 + out1)
```

- Фінальна частина блоку. Для декодеру у якості параметру до Point-wise feed-foorward network (ffn) замість out1 використовується out2

```
    ffn_output = self.ffn(out1)
    ffn_output = self.dropout3(ffn_output, training=training)
    out2 = self.layernorm3(out1 + ffn_output)
```

Також було розглянуто модель та побудовано модель GPT-2, що має подібну структуру до моделі енкодеру Transformers. (Див. Додаток D)

Інший відомий різновид Transformers – BERT не було розглянуто, оскільки така модель не підходить для вирішення задачі машинного перекладу за [9] .

У процесі тренування моделей для кожної з них Tensorflow побудував динамічний граф, який можна було візуалізувати з використанням Tensorboard. Порівняння концептуальної структури моделі та створених графів наведено у додатку G.

4.4.2 Результати Transformers

З графіків на рис. 4.1 видно, як покращується якість тренування в залежності від параметрів:

1. Точність погіршується зі збільшенням довжини речень
2. При тренуванні на subwords та цілих словах, результати прямують до однакових значень
3. Чим більше речень, тим швидше модель знайде кращі параметри
4. За наявності маскування додаткових нулевих значень в кінці речень, модель з різною довжиною речень в кожному батчі не дає переваг

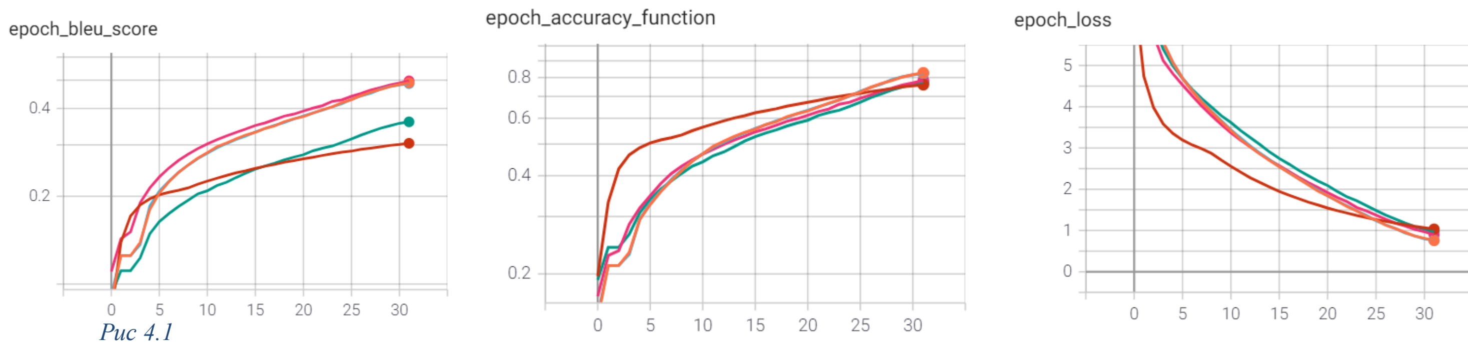


Рис 4.1

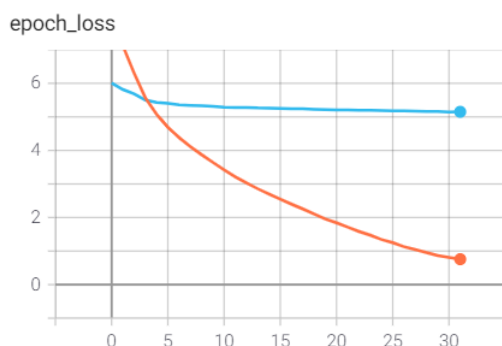
- Помаранчева лінія – Transformers, 6400 речень,
  речення закодовані через subwords, максимальна довжина речень – 16 слів
- Червона лінія – 32 000 речень, subwords, довжина - 16 слів
- Синя лінія – 6400 речень, subwords, довжина - 32 слова

- Рожева лінія – 6400 речень, цілі слова, довжина - 16 слів
- Зелена лінія – 6400 речень, цілі слова, в кожному батчі довжина речень базується на довжині максимального.

Name	Smoothed	Value	Step	Time	Relative
20210110-191231\train	0.7575	0.7575	31	Sun Jan 10, 19:16:19	3m 5s
20210110-204700\train	1.034	1.034	31	Sun Jan 10, 21:05:02	17m 0s
20210110-212617\train	0.7643	0.7643	31	Sun Jan 10, 21:29:34	2m 57s
20210110-222947\train	0.902	0.902	31	Sun Jan 10, 22:32:40	2m 38s
20210110-224623\train	0.9736	0.9736	31	Sun Jan 10, 22:50:08	3m 26s

При тренуванні усіх моделей було використано Adam optimizer з власно визначеною швидкістю навчання, що змінюється (в основному, лінійно збільшується) з кожною епохою.

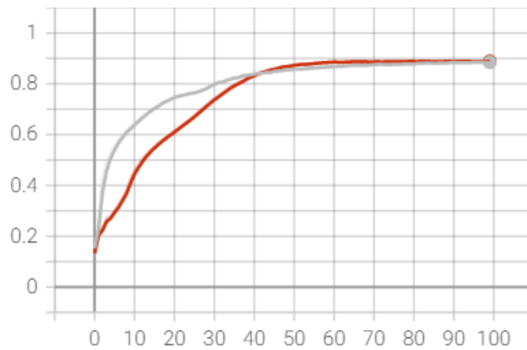
При спробі використати RMSprop optimizer, через його згадану у розділі 4.3.2 особливість, модель показує погані результати (блакитна лінія).

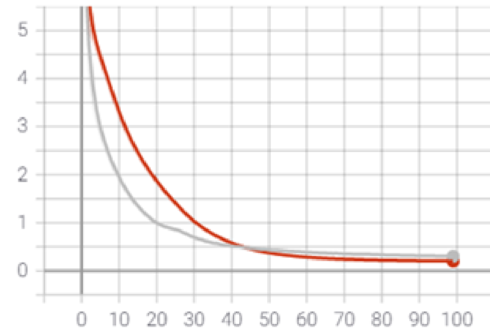


Також для порівняння з результатами минулих моделей, Transformer було використано на тих самих коротких реченнях французько-англійського перекладу, що використовувались й для тренування RNN.

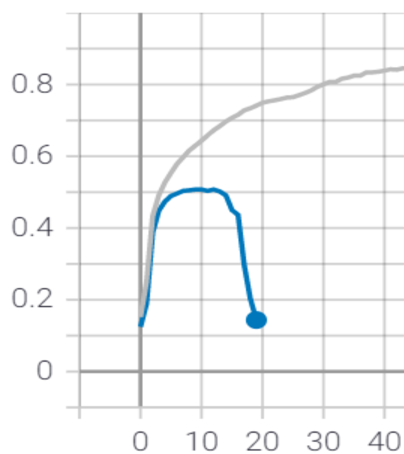
Червона лінія - RNN, сіра - Transformer.

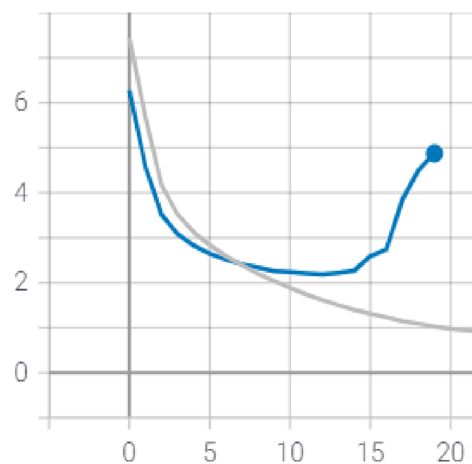
Transformer швидше сходиться до кращих значень. Але за 100 епох кінцеві результати стають майже однаковими.





Синя лінія - результат роботи GPT-2 при тих самих даних на вході та виході.





Як видно з графіків, до першого моменту модель на основі GPT-2 тренується так само гарно як і Transformers, але через кілька епох результати стрімко погіршуються. Можливо, така модель зможе показати кращі результати, якщо підібрати для неї гарний оптимізатор з нестандартними параметрами.

Висновок

У роботі було досліджено різні моделі нейронних мереж для машинного перекладу. Якщо корпус вхідних текстів охоплює більшу часину найвживаніших слів, то речення краще токенізувати на слова. При порівнянні на коротких реченнях найкращі результати показує Bidirectional LSTM. Також з покращенням результатів допомагає зворотний порядок слів на вході.

Гарні результати також демонструє Transformers, яка більше підходить для перекладу довгих речень через свій механізм уваги та розпаралелювання обчислень. GPT-2 та BERT не підходять для вирішення цієї задачі.

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

Для наглядного зображення результатів можна використовувати проекти Tensorflow, а саме Tensorboard та Embedding projector.

У майбутніх дослідженнях може також розглядатися знаходження кращих гіперпараметрів для роботи GPT-2 та використання згорткових нейронних мереж для перекладі речень та.

Перелік використаних джерел

[1] – https://en.wikipedia.org/wiki/History_of_machine_translation

[2] – https://en.wikipedia.org/wiki/Machine_translation

[3] – https://www.sciencedirect.com/topics/engineering/feedforward

[4] – Vashee K. Understanding MT Quality: BLEU Scores [Електронний ресурс] / Vashee. – 2019. – Режим доступу до ресурсу: https://kvashee.medium.com/understanding-mt-quality-bleu-scores-9a19ed20526d.

[5] – Kostadinov S. Understanding Backpropagation Algorithm [Електронний ресурс] / Simeon Kostadinov – 2019. – Режим доступу до ресурсу: https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd.

[6] – https://en.wikipedia.org/wiki/Transformer_(machine_learning_model)

[7] –

Фреймворк глубокого обучения в 2019: выбираем из 10 лучших [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://proglib.io/p/dl-frameworks ,

Komissarenko N. TensorFlow vs PyTorch [Електронний ресурс] / Nick Komissarenko. – 2020. – Режим доступу до ресурсу: https://medium.com/@bigdataschool/tensorflow-vs-pytorch-317bf4ec197f.,

Goldsborough P. Экскурсия по PyTorch [Електронний ресурс] / Peter Goldsborough. – 2018. – Режим доступу до ресурсу: https://habr.com/ru/company/piter/blog/354912/.

[8] – https://www.tensorflow.org/tensorboard

[9] – Jordan J. Hyperparameter tuning for machine learning models. [Електронний ресурс] / Jeremy Jordan. – 2017. – Режим доступу до ресурсу: https://www.jeremyjordan.me/hyperparameter-tuning/.

[10] – Koehn P. Hands-On Machine Learning with Scikit-Learn, Kears & Tensorflow / Philipp Koehn.

[11] – Machine Translation Weekly 32: BERT in Machine Translation [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://jlibovicky.github.io/2020/03/05/MT-Weekly-BERT-for-MT.html.

[12] – Koehn P. Neural Machine Translation / Philipp Koehn.

[13] – Ghodsi A. Attention Mechanism, Transformers, BERT, and GPT: Tutorial and Survey / A. Ghodsi, A. Ghojogh.

[14] – Geron A. – “Hands-On Machine Learning with Scikit-Learn, Kears & Tensorflow” / Aurelien Geron.

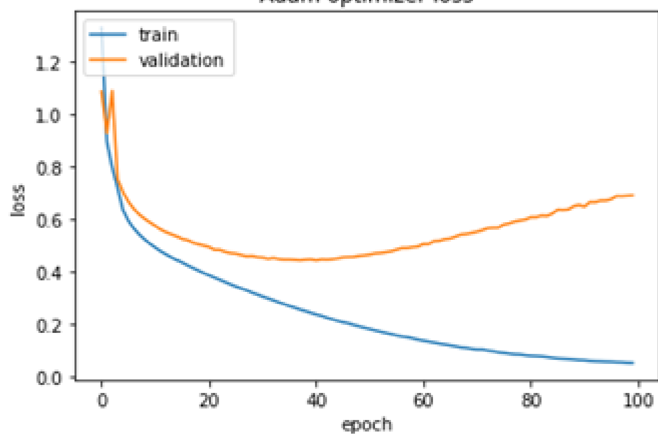
[15] – Singh P. Learn TensorFlow 2.0. Implement Machine Learning and Deep Learning Models with Python / P. Singh, A. Manure.

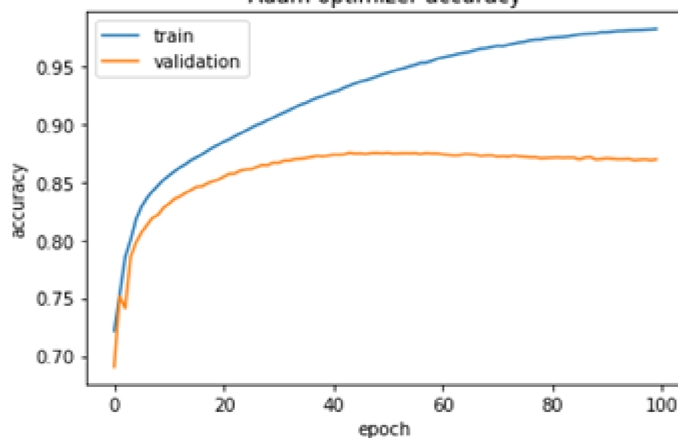
[16] – Deep Learning and Neural Networks: Concepts, Methodologies, Tools / Information Resources Management Association

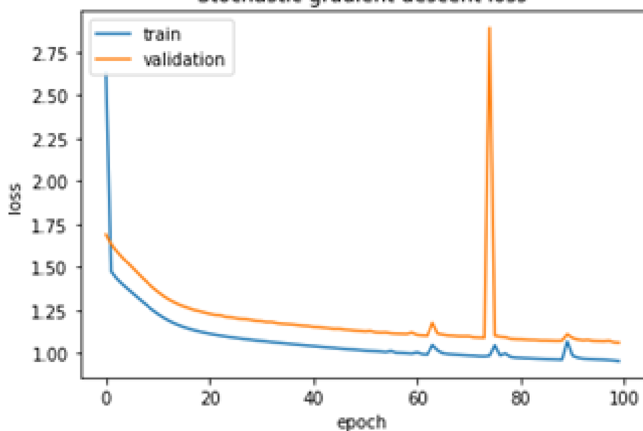
[17] – Токенизация на подслова (Subword Tokenization) [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://dyakonov.org/2019/11/29/токенизация-на-подслова-subword-tokenization.

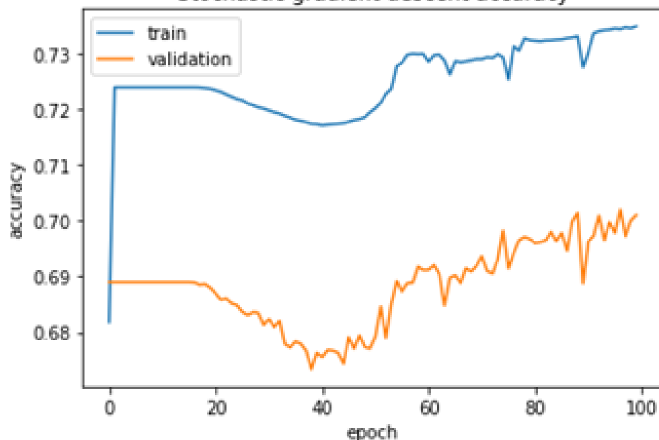
[18] - Embedding Projector [Електронний ресурс] – Режим доступу до ресурсу: https://projector.tensorflow.org/.

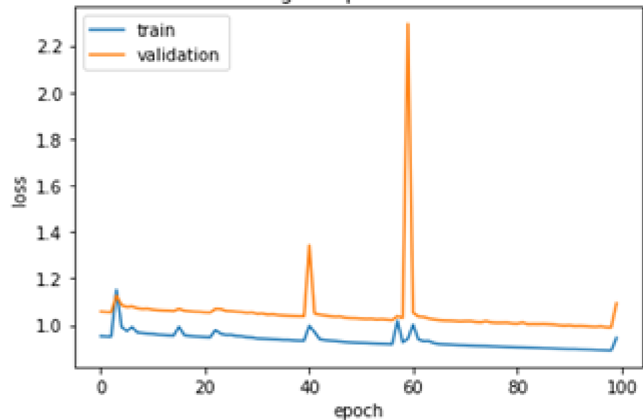
Додаток А

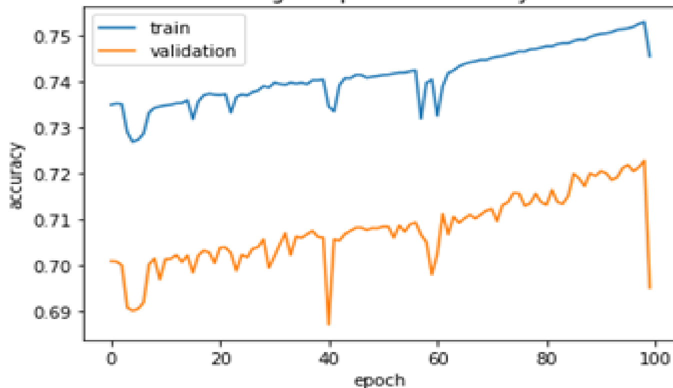


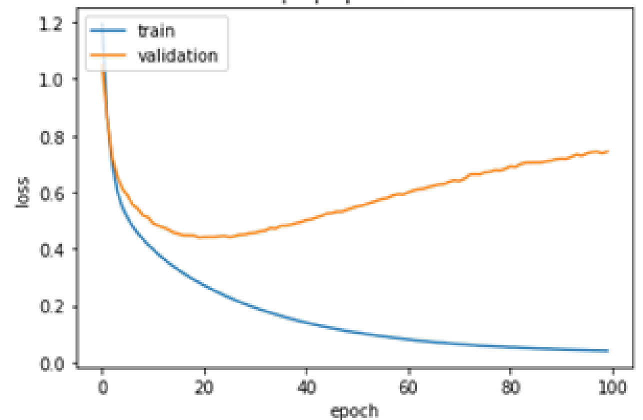


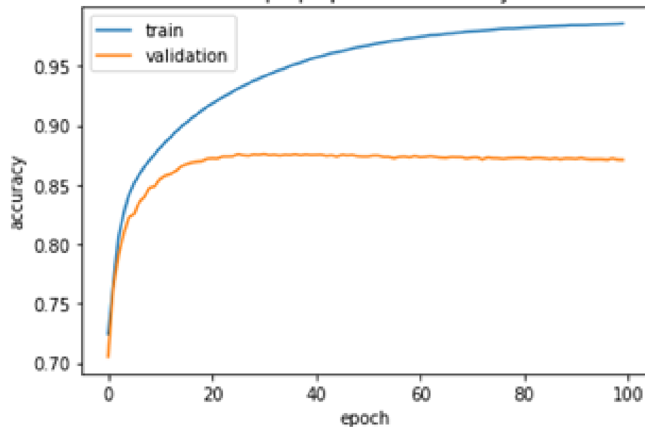






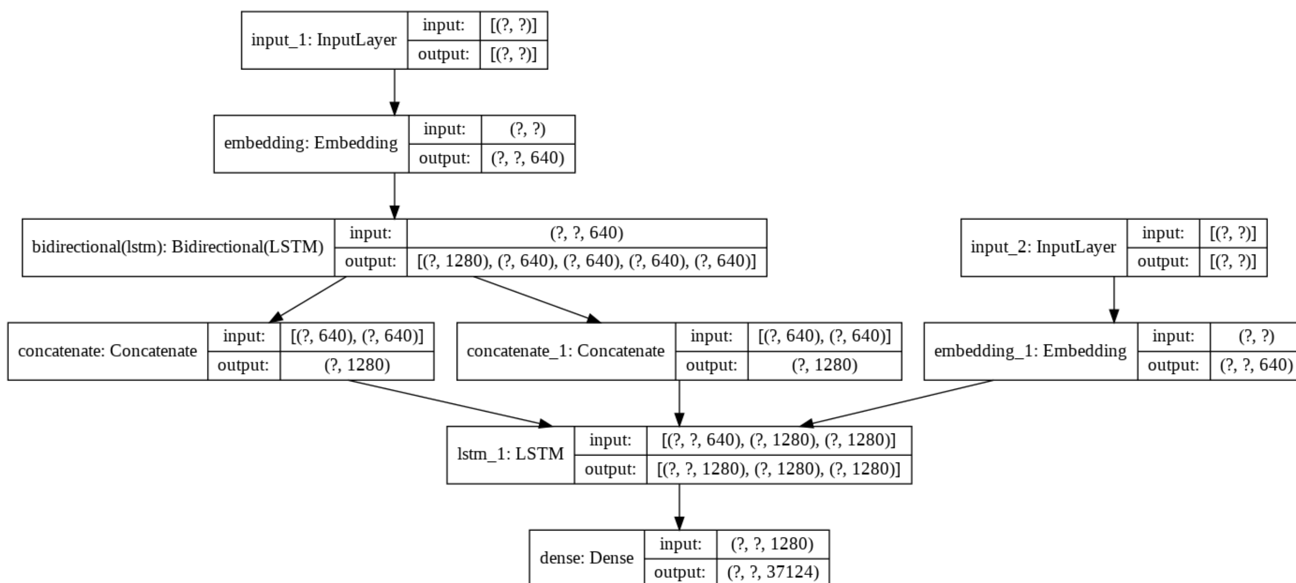






Додаток В

Побудова структури функціональної моделі до початку обчислень з використанням keras.utils.plot_model



Перелік всіх параметрів послідовної моделі до початку обчислень з використанням keras.model.summary()

```
Layer (type)                 Output Shape              Param #
=================================================================
inputs (InputLayer)          (None, 150)               0

embedding_1 (Embedding)      (None, 150, 50)           50000

lstm_1 (LSTM)                (None, 64)                29440

FC1 (Dense)                  (None, 256)               16640

activation_1 (Activation)    (None, 256)               0

dropout_1 (Dropout)          (None, 256)               0

out_layer (Dense)            (None, 1)                 257

activation_2 (Activation)    (None, 1)                 0
=================================================================
Total params: 96,337
Trainable params: 96,337
Non-trainable params: 0
```

Додаток С

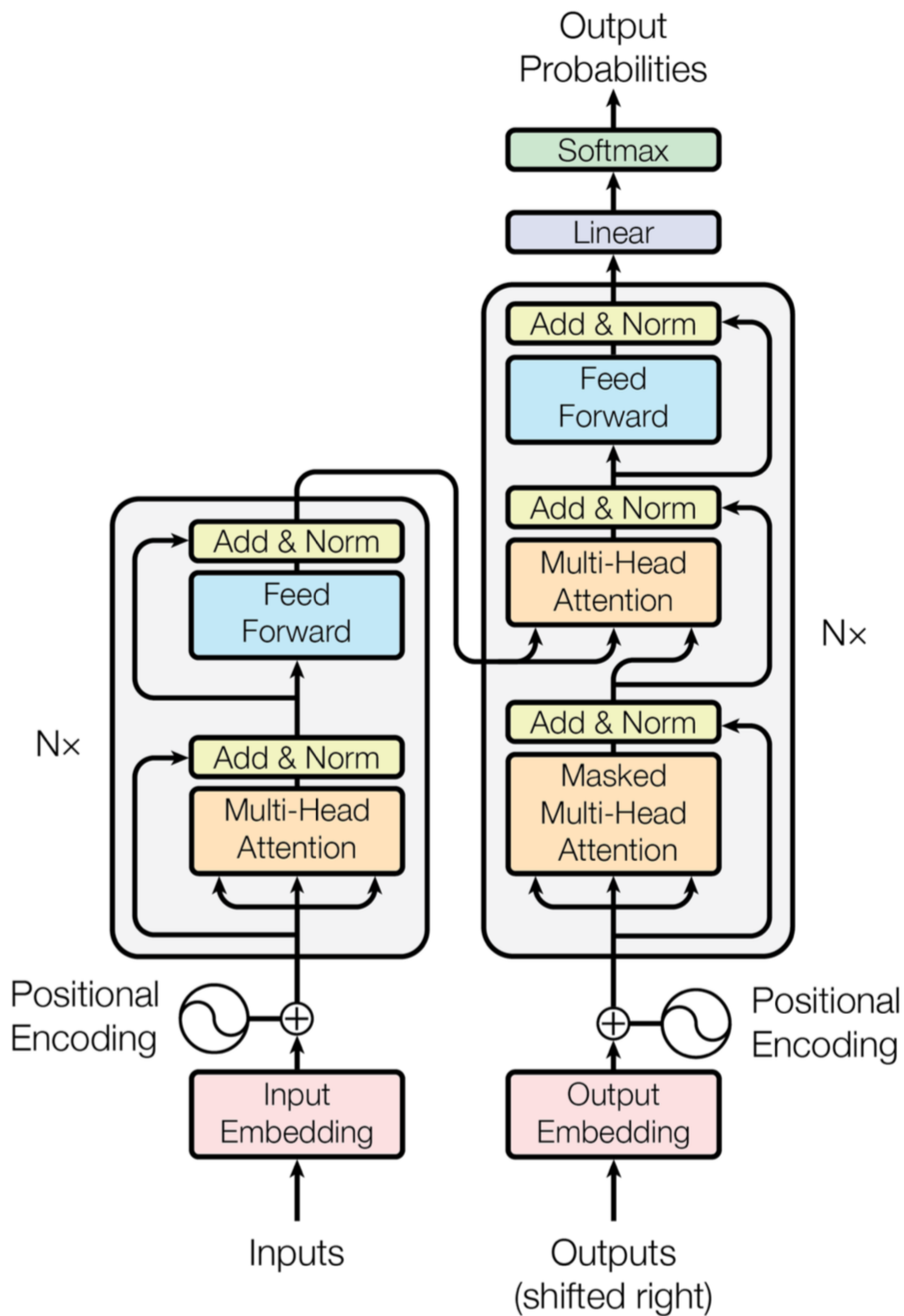
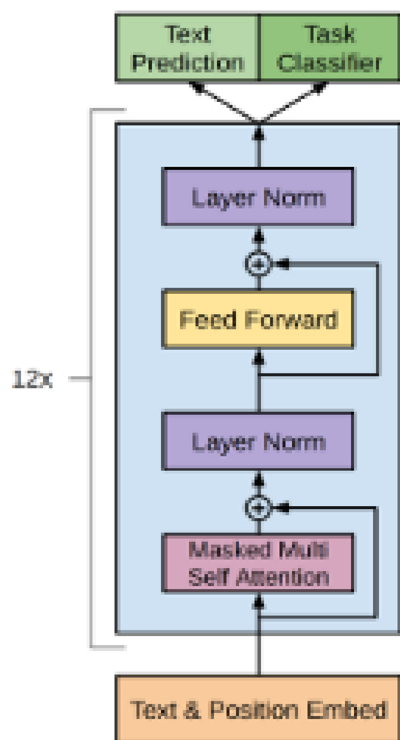
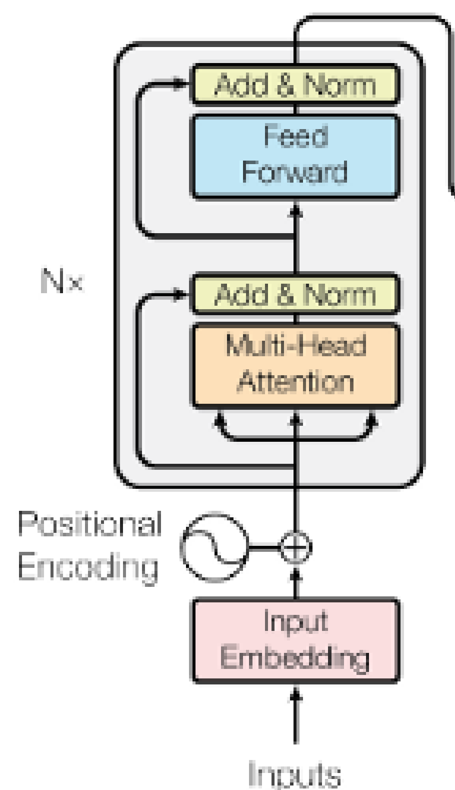


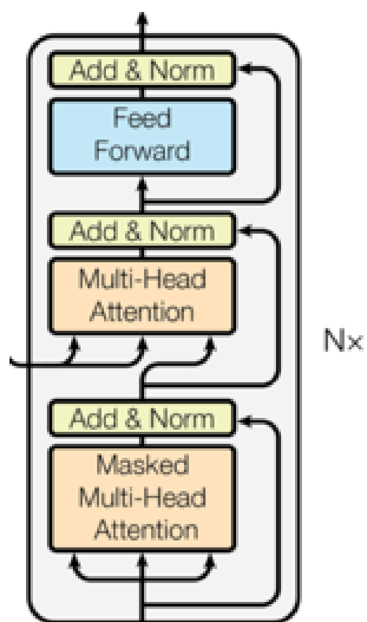
Figure 1: The Transformer - model architecture.

Додаток D

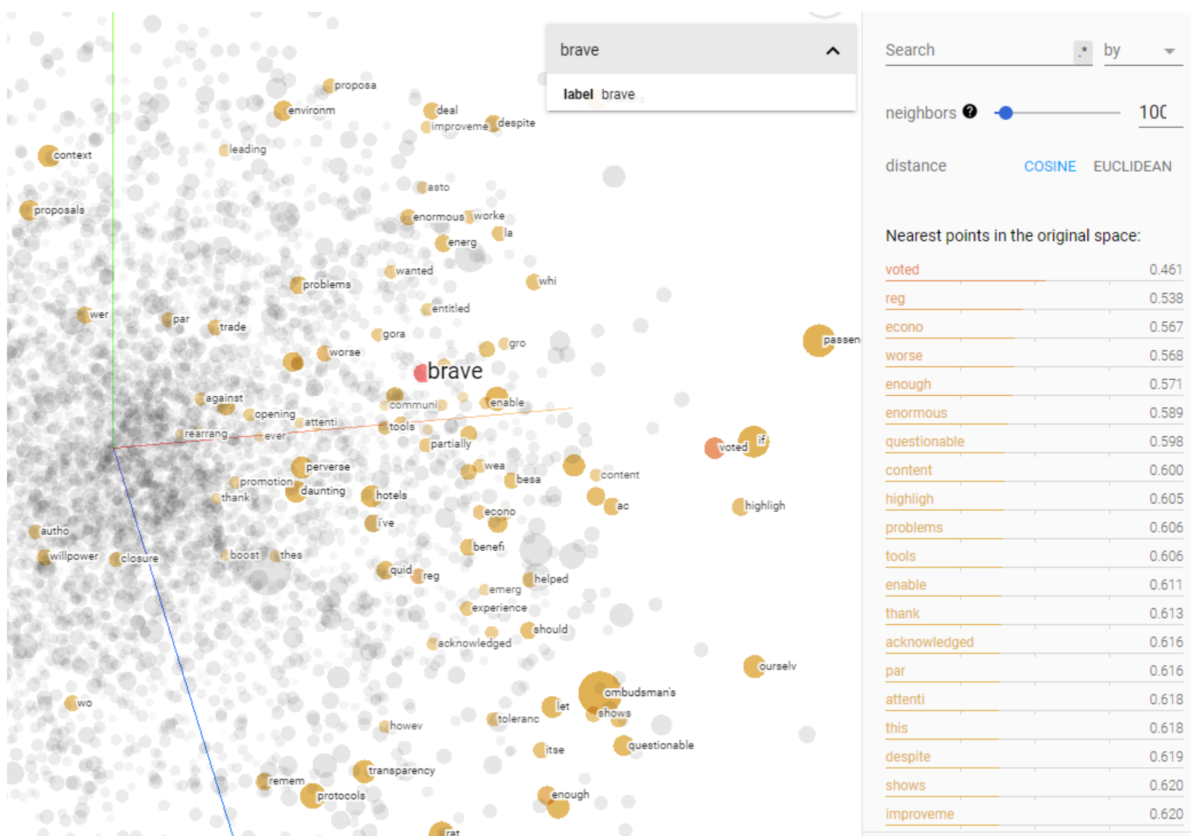


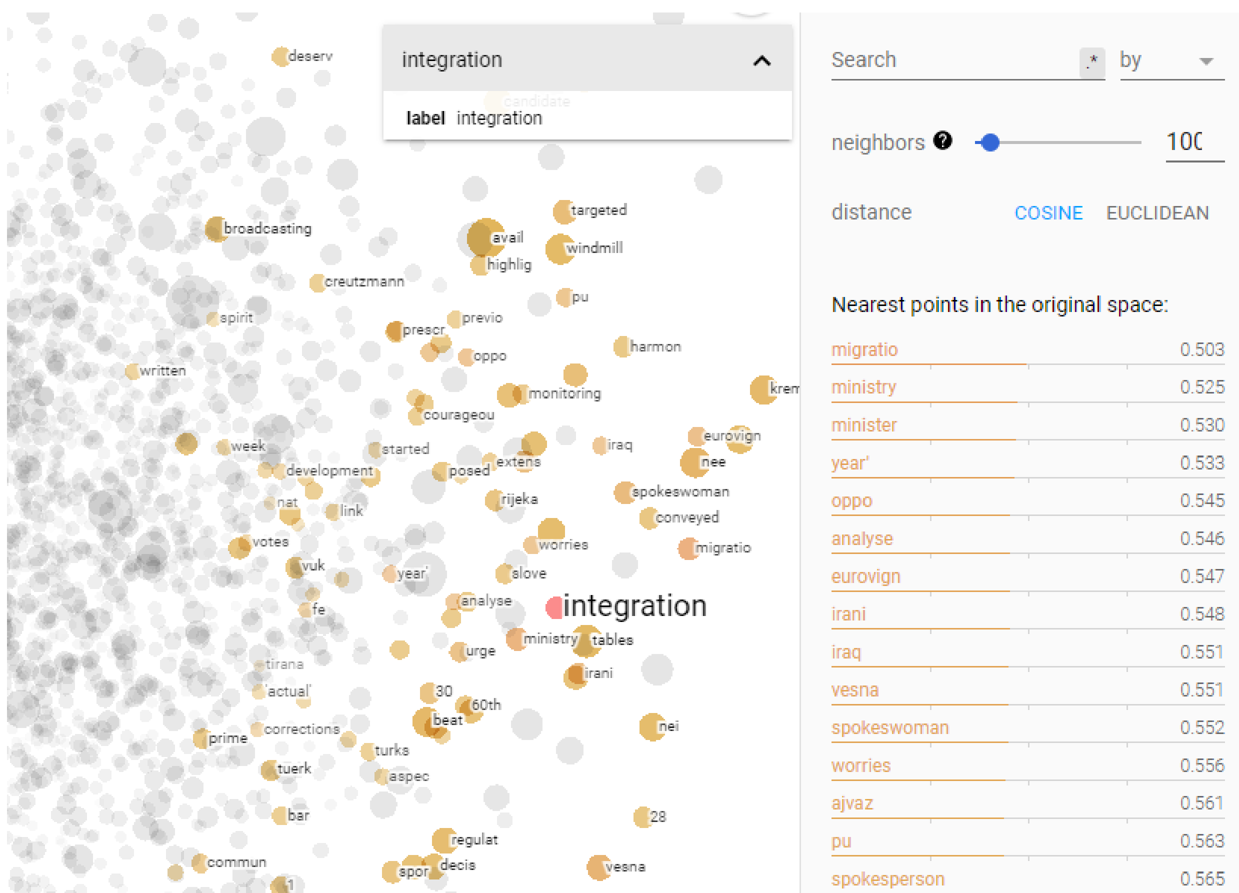


Додаток E



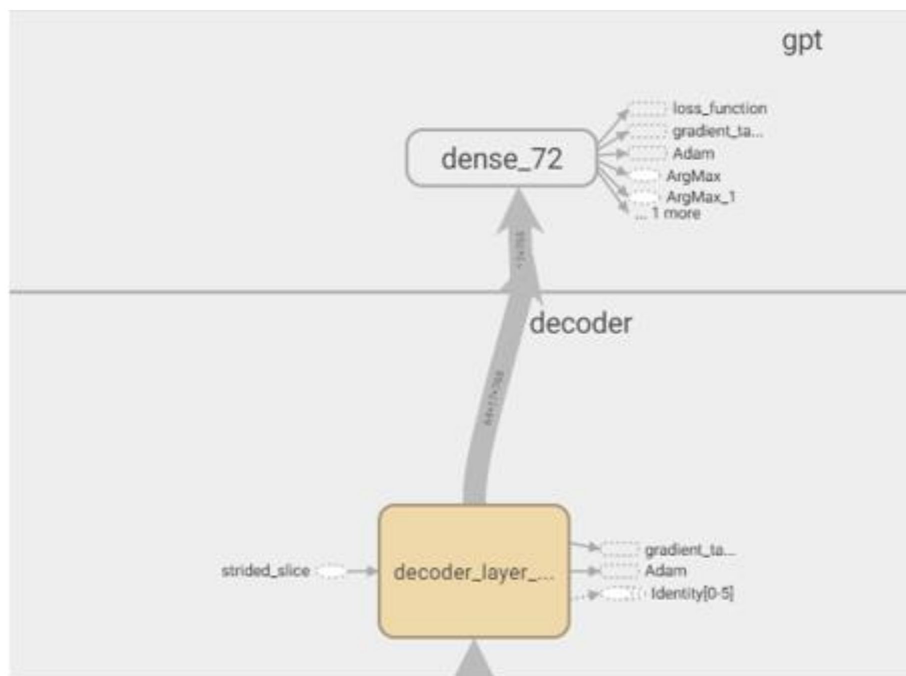
Додаток F



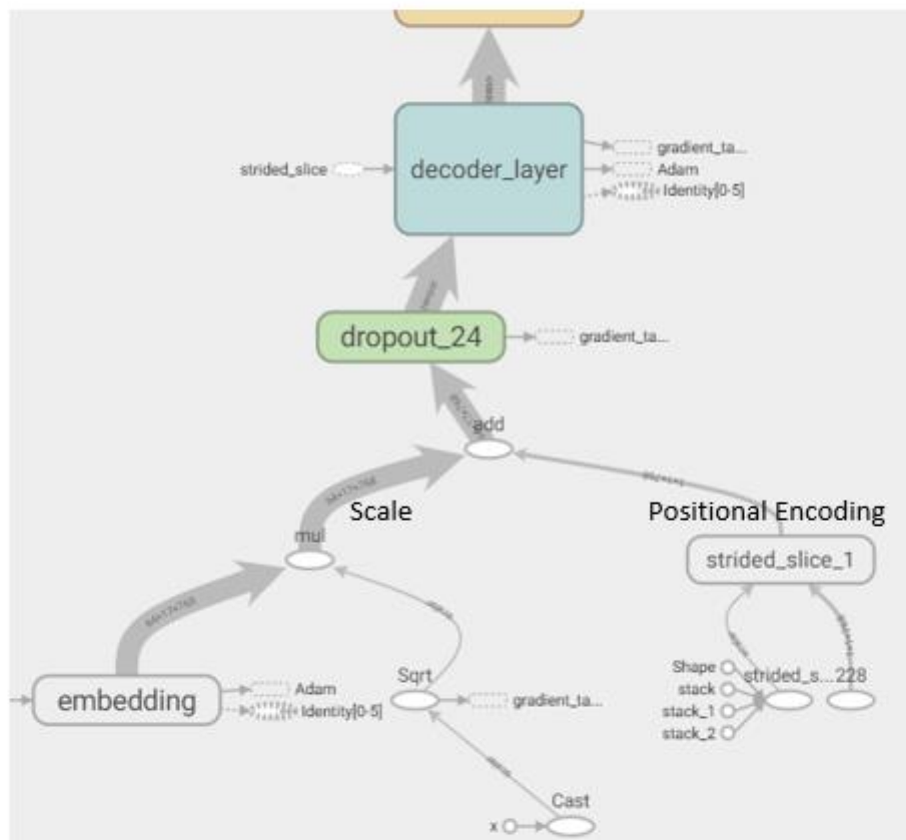


Додаток G

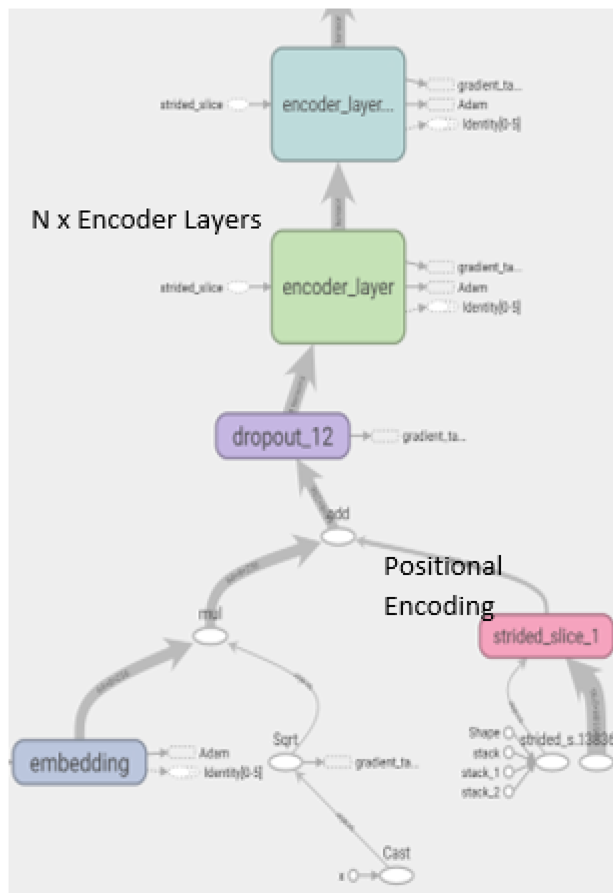
Динамічний граф GPT-2



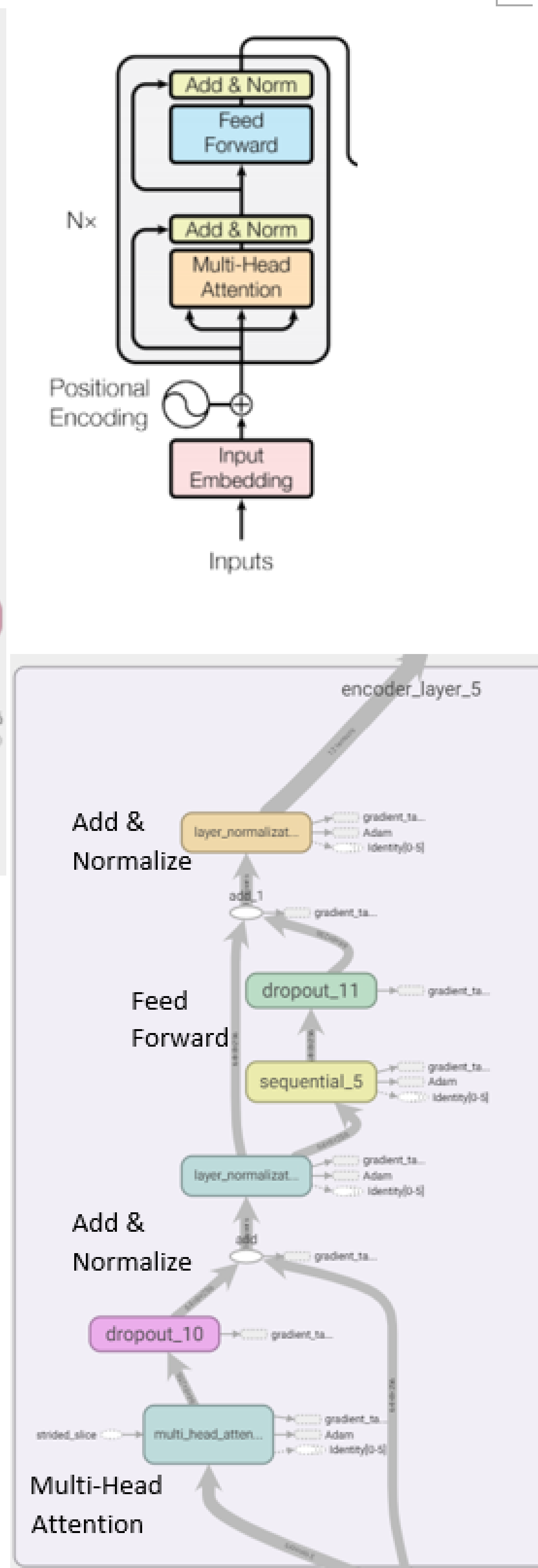
N x Decoder Layer . . .



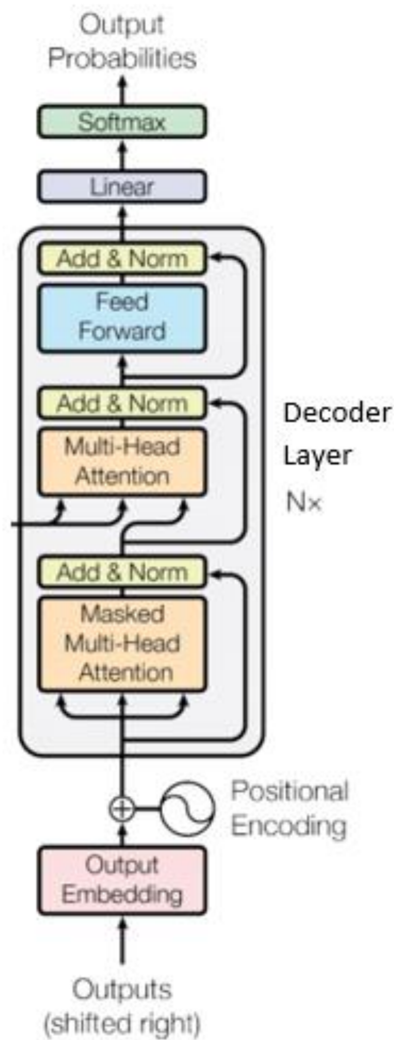
Динамічний граф Енкодера Transformer

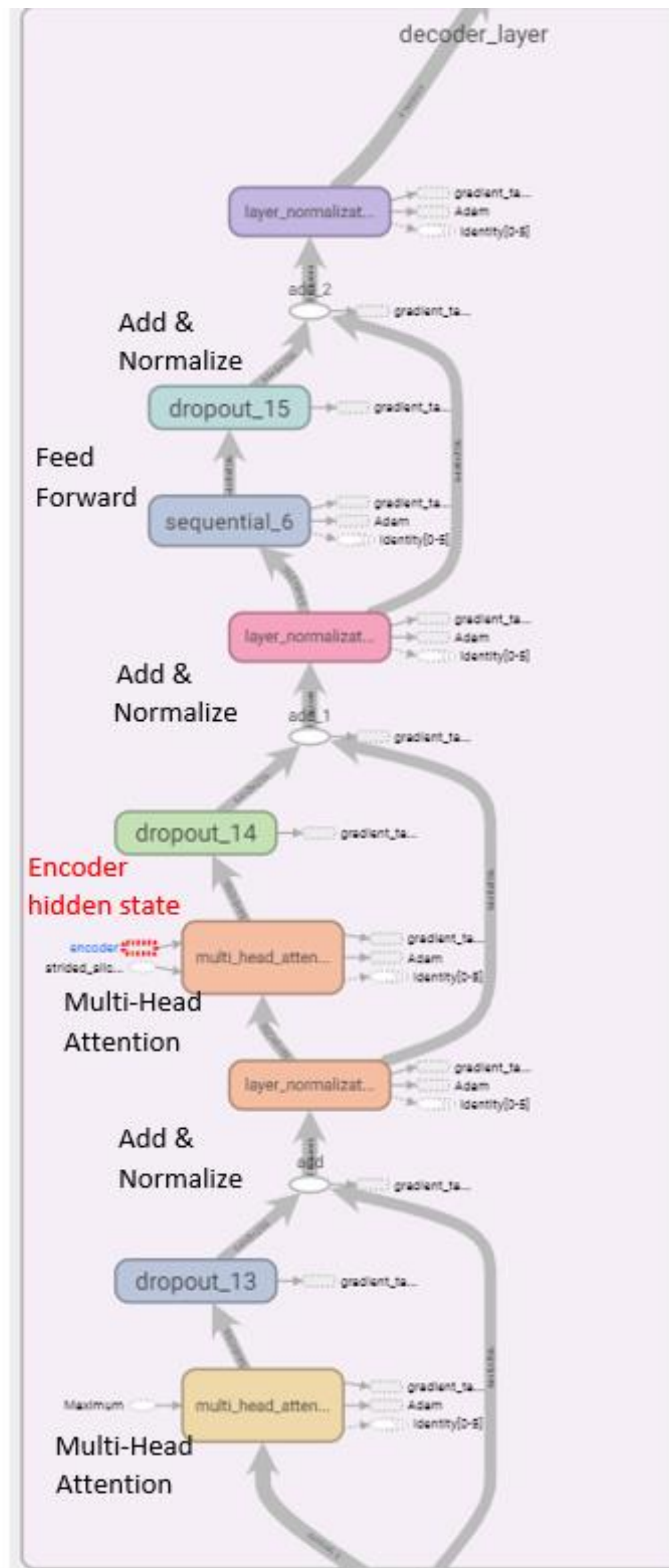


Модель Енкодера в Трансформері



Динамічний граф Декодера Transformers





Додаток Н

Наївний підхід

```
def counter(tokens):
    counter1 = {}
    for token in tokens:
        if token == 0:
            continue
        if token in counter1:
            counter1[token] += 1
        else:
            counter1[token] = 1
    return counter1

def bleu_score1(real, pred):
    counter1 = counter(real)
    counter2 = counter(pred)
    res = 0
    for token in counter2:
        if token in counter1:
            res += min(counter1[token], counter2[token])

    return res / len(pred)
```

Використання допоміжних функцій для створення метрики, яку можна використовувати з графами обчислень.

```
def bleu_score2(real, pred):
    real, pred = tf.boolean_mask(real, tf.cast(real, tf.bool)), tf.boolean_mask(pred, tf.cast(pred, tf.bool))
    real_k, _, real_c = tf.unique_with_counts(real)
    pred_k, _, pred_c = tf.unique_with_counts(pred)
    common_elems = tf.where(tf.equal(tf.expand_dims(real_k,1), tf.expand_dims(pred_k,0) ))
    common_real_idx = common_elems[:,0]
    common_pred_idx = common_elems[:,1]
    scores = tf.math.minimum( tf.gather(real_c, common_real_idx), tf.gather(pred_c, common_pred_idx) )
    return tf.cast( tf.reduce_sum(scores) / tf.size(pred), tf.float32)


def bleu_score(real, pred):
    real = tf.cast(real, tf.float32)
    pred = tf.cast(tf.math.argmax(pred, -1), tf.float32)
    z = tf.stack([real, pred], 1)
    scores = tf.map_fn(lambda x: bleu_score2(x[0], x[1]), z)
    return tf.reduce_sum(scores) / tf.cast(tf.size(scores), tf.float32)
```