Министерство образования и науки РФ

Федеральное государственное автономное образовательное

учреждение высшего образоования

«Казанский (Приволжский) федеральный университет»

Институт Вычислительной математики и информационных технологий

Кафедра системного анализа и информационных технологий

Направление: 02.04.02 - Фундаментальная информатика

и информационные технологии

КУРСОВАЯ РАБОТА

Реализация метода вычисления векторных представлений слов

Студент 1 курса

Группа 09-835

«___»_____ 2019 г. _____ Хуснутдинов Л.Р.

Научный руководитель

к.ф.-м.н., ассистент

«___»_____ 2019 г. _____ Разинков Е.В.

Казань-2019

Содержание

1 Введение **3**

1.1 Научная новизна 3

1.2 Актуальность 4

1.3 Цели и постановка задачи 4

1.4 Терминология 4

2 Методы вычисления векторных представлений слов с помощью нейронной сети Transformer **5**

2.1 Transformer 5

2.2 Transformer Encoder 5

2.2.1 Входная последовательности токенов 6

2.2.2 Таблица векторных представлений 7

2.2.3 Positional Encoding 8

2.2.4 Multi-head Attention 9

2.2.5 Residual connections 11

2.2.6 Layer normalization 12

2.2.7 Полносвязный слой 12

2.2.8 Encoder Layer 14

2.2.9 Encoder Transformer 15

2.3 Входной вектор 16

2.3.1 Векторные представления сегментов 17

2.4 Задачи 17

2.4.1 Masked Language Model 17

2.4.2 Next Sentence Prediction 19

2.5 Число обучаемых параметров BERT 19

2.6 Число операций в BERT 20

3 Реализация и эксперимент **22**

3.1 Язык программирования и библиотеки 22
3.2 Набор данных 22
3.3 Параметры 22
3.4 Результаты 23
3.4.1 Pre-training 23
3.4.2 Задача классификации текстов 23
3.4.3 Эксперимент по изменение параметров 24
3.4.4 Эксперимент с изменением числа операций 27

4 Заключение **30**

Список литературы **31**

Приложение 1 **32**

1 Введение

1.1 Научная новизна

Векторное представление – это n-мерный вектор, с помощью которого представляются различные единицы языка – слова, предложения, параграфы. Таким образом отражаются различные характеристики и особенности естественного языка – семантика, синтаксис и др. Одним из способов представления единиц языка является Bag-of-words (англ. мешок слов). Но подобные методы имеют ряд недостатков – высокая размерность, высокая разреженность, неиспользование информации о порядке слов, неспособность улавливать контекст слова (под контекстом понимается окружение слова).

В 2013 году Томас Миколов представил статью "Efficient Estimation of Word Representations in Vector Space" [3], в которой описывается новый метод векторных представлений слов – Word2Vec. Основа метода – это алгоритмы CBOW (Continuous Bag of Words), Skip-gram и искусственная нейронная сеть прямого распространения (feed-forward neural network). Word2Vec обучается на большом языковом корпусе с помощью нейронной сети в зависимости от алгоритма, предсказывая слово по контексту (CBOW) либо контекст по данному слову (Skip-gram).

Также вышеперечисленные модели неспособны справляться с полисемией – многозначностью слова. Например, слово "ключ" может иметь разные значения в зависимости от контекста. На смену Word2Vec пришли более сложные и более глубокие архитектуры нейронных сетей, которые способны создавать векторные представления, содержащие больше информации о свойствах языка, и способные справляться с описанными выше проблемами.

Векторные представления слов являются основой для решения широкого класса задач обработки естественного языка. Долгое время для получения векторных представлений использовались методы, в которых не применялось глубокое обучение, либо нейронные сети не применялись вовсе.

В настоящее время для получения векторных представлений слов используется глубокое обучение. Примерами таких архитектур являются ELMo (Deep contextualized word representations) [1] и BERT (BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding) [2]. На момент своего выхода данные архитектуры демонстрировали state-of-the-art результаты в различных задачах обработки естественного языка.

Новизна архитектуры Transformer, лежащей в основе BERT, простота идеи и реализации, а также state-of-the-art результаты послужило мотивацей для изучения BERT. Но глубокое обучение требует больших вычислительных ресурсов, в случае

модели BERT - это ресурсы, которыми обладают только компании-гигианты уровня Google. В частности, для обучения оригинальной модели использовалось 64 тензорных процессора (англ. tensor processing unit). Важным вопросом для исследования становится ускорение получения state-of-the-art результатов, используя ограниченное количество ресурсов.

1.2 Актуальность

Векторные представления слов являются одним из важнейшим инструментом для задач обработки естественного языка, а именно для:

- анализа тональности текстов;
- чат-ботов;
- систем машинного перевода и т.д.

1.3 Цели и постановка задачи

Целью курсовой работы является ускорение процесса обучения модели BERT, то есть сделать модель вычислительно проще. Для достижения этой цели ставятся следующие задачи:

- изучение BERT;
- реализация модели BERT;
- обучение модели;
- валидация результатов на задаче классификации текстов;
- модификация модели с целью ускорения процесса обучения;

1.4 Терминология

Для удобства введём англоязычные термины, которым трудно подобрать аналог в русском языке:

- batch - пакет, набор данных, батч;
- residual connections - остаточные соединения;
- positional encoding - позиционное кодирование;

2 Методы вычисления векторных представлений слов с помощью нейронной сети Transformer

2.1 Transformer

Основой BERT является модель Transformer. Архитектура сети Transformer была представлена в статье “Attention is all you need” [4]. Используя стандартную парадигму Encoder-Decoder, Transformer представляет собой совершенно новую архитектуру нейронных сетей, основой которого является механизм внимания. Главным преимуществом перед рекуррентными нейроннами сетями является способность извлекать информацию из более длиных входных последовательностей.

В BERT используется только Encoder из Transformer, поэтому будет рассмотрен только Encoder.

2.2 Transformer Encoder

Рассмотрим Encoder Transformer. На Рисунке 1 Encoder выделен красным цветом. Encoder состоит из следующих элементов:

1. Входная последовательность токенов;
2. Таблица векторных представлений;
3. Positional Encoding;
4. Механизм Multi-head Attention;
5. Layer-normalization;
6. Residual connections;
7. Полносвязный слой;

Рассмотрим каждый слой по отдельности.

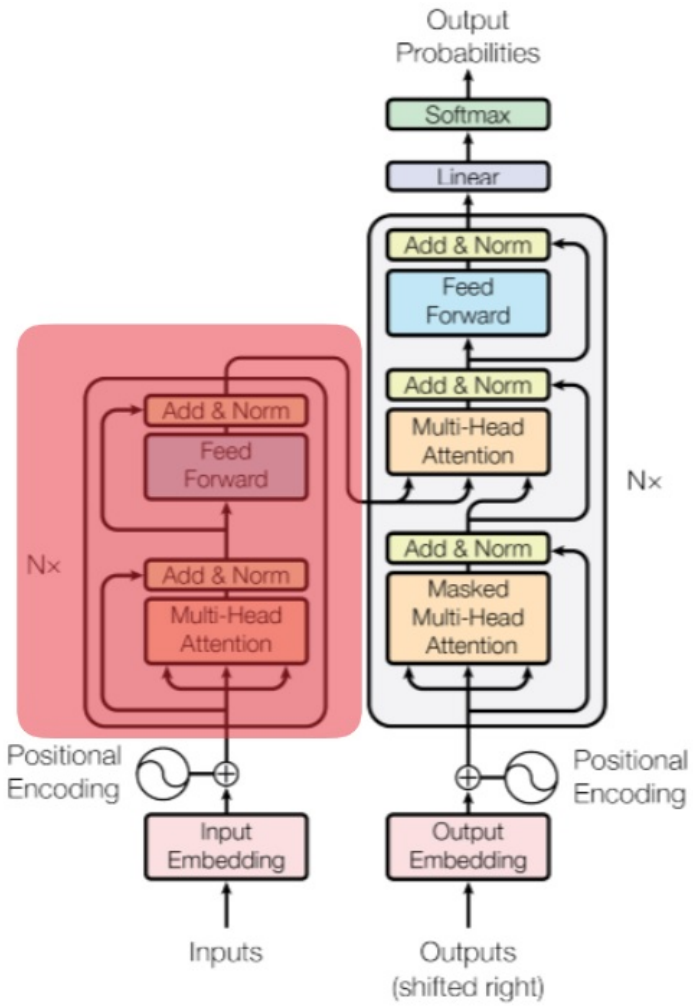

Рис. 1: Encoder-Decoder Transformer.

2.2.1 Входная последовательности токенов

Входная последовательность представляет собой вектор $(t_1, ..., t_n)$, где t_i – это отдельный токен, а вектор имеет фиксированную длину. Токен - это единица входной последовательности слов, разбитой каким-либо образом на части.

В качестве примера возьмём последовательность – “I want to believe” и ограничим длину последовательности двумя токенами. Пусть входная последовательность представляет собой два токена – “to believe”.

Рис. 2: Входная последовательность.

2.2.2 Таблица векторных представлений

Таблица векторных представлений представляет собой отображение токена в n-мерный вектор. На Рисунке 3 представлен пример, где исходному токен believe с порядковым номером 1 сопоставляется вектор $(9, 7, 3, 9)$ в таблице векторных представлений.

В качестве примера возьмем размерность векторного представления равной 4 и проинициализируем его значениями от 0 до 10 из равномерного распределения. В оригинальной реализация таблица векторных представлений инициализиурется с помощью нормального распределения.

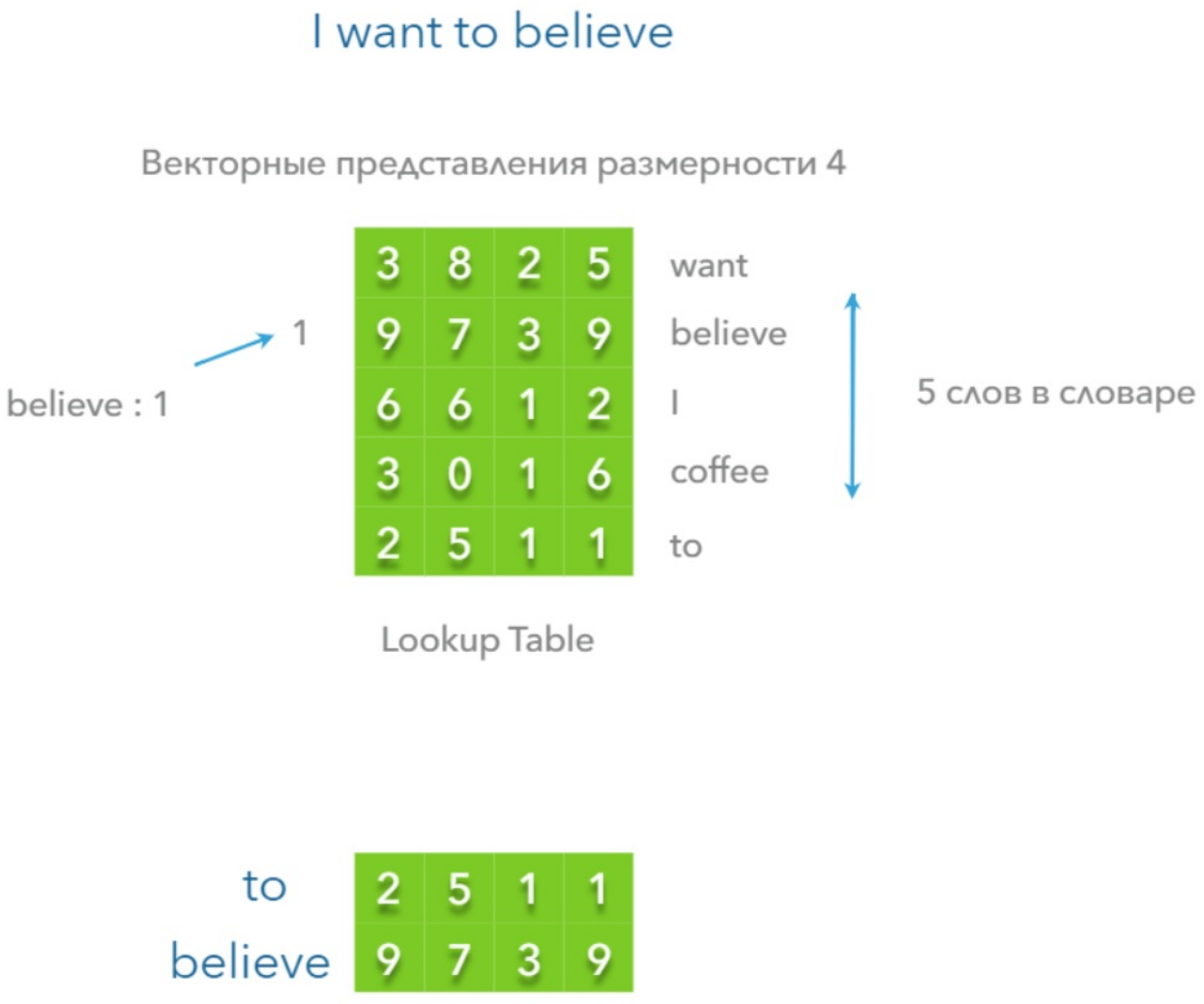

Рис. 3: Embedding lookup table.

2.2.3 Positional Encoding

Так как подобная архитектура не знает о том, в какой последовательности идут входные токены, то необходимо сообщить нейронной сети информацию о взаимном расположении символов во входной последовательности. Это делается с помощью Positional Encoding - входной вектор суммируется с вектором Positional Encoding.

Positional Encoding вычисляется с помощью функций косинус и синус, где i – это позиция в векторном представлении, pos – позиция слова в последовательности, d_{model} – размерность векторного представления:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

Для удобства демонстрации округлим значения Positional Encoding векторов и просуммируем с исходными векторными представлениями. На Рисунке 4 представлены описанные операции:

Рис. 4: Positional encoding.

2.2.4 Multi-head Attention

На Рисунке 5 представлен основной элемент Transformer - Multi-head Attention.

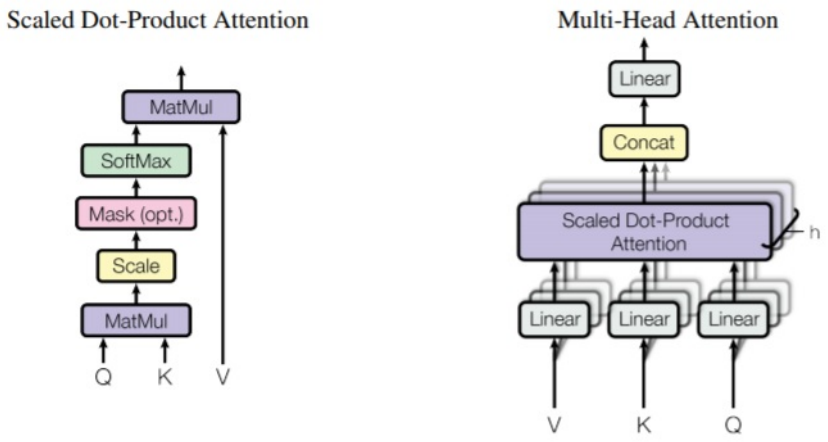

Рис. 5: Scaled Dot-Product Attention и Multi-Head Attention [4].

На вход Multi-Head Attention блоку подаются матрицы V, K, Q – в случае, если это первый слой, то эти матрицы одинаковы и совпадают со входной матрицей.

Далее к матрицам V, K, Q применяется линейное преобразование h раз. На Рисунке 6 показаны описанные операции:

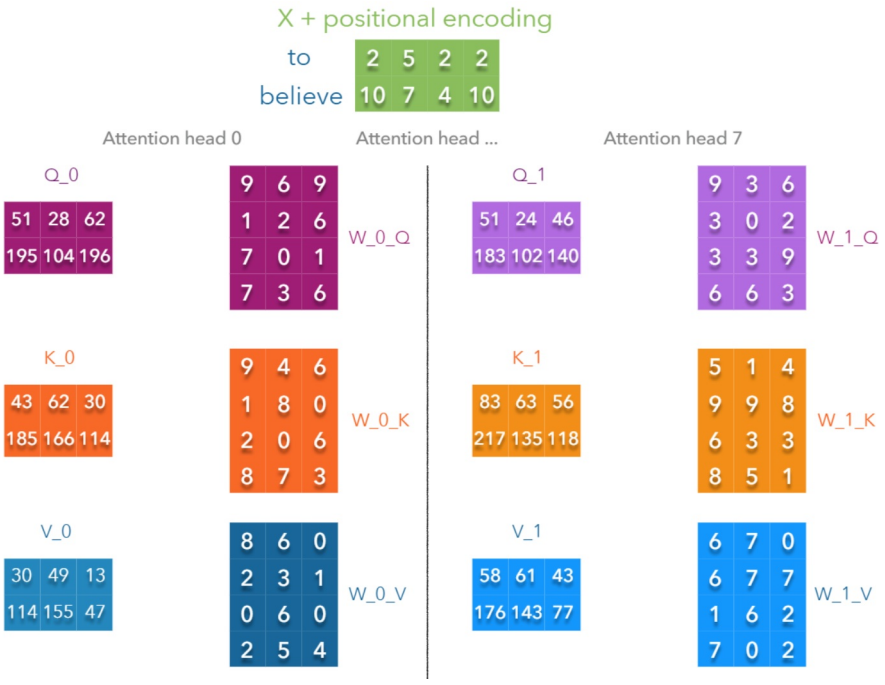

Рис. 6: Операция Linear в блоке Multi-Head Attention.

Затем к каждой полученной матрицей применяется блок Scaled Dot-Product Attention, тоже h раз:

Рис. 7: Операция Scaled Dot-Product Attention в блоке Multi-Head Attention.

Блок Scaled Dot-Product Attention начинается с матричного умножения Q и K матриц, далее следует операция Scale – деление полученных матриц на $1/d_k$, где d_k – одна из размерностей матрицы K. Далее применяется операция Mask и Softmax. Далее приосходит матричное умножение полученной матрицы и матрицы V.

Таким образом, блок Scaled Dot-Product Attention состоит в следующем:

$$Attention(Q, K, V) = softmax\left(\frac{QK}{\sqrt{d_k}}\right)V$$

Полученные матрицы конкатенируются:

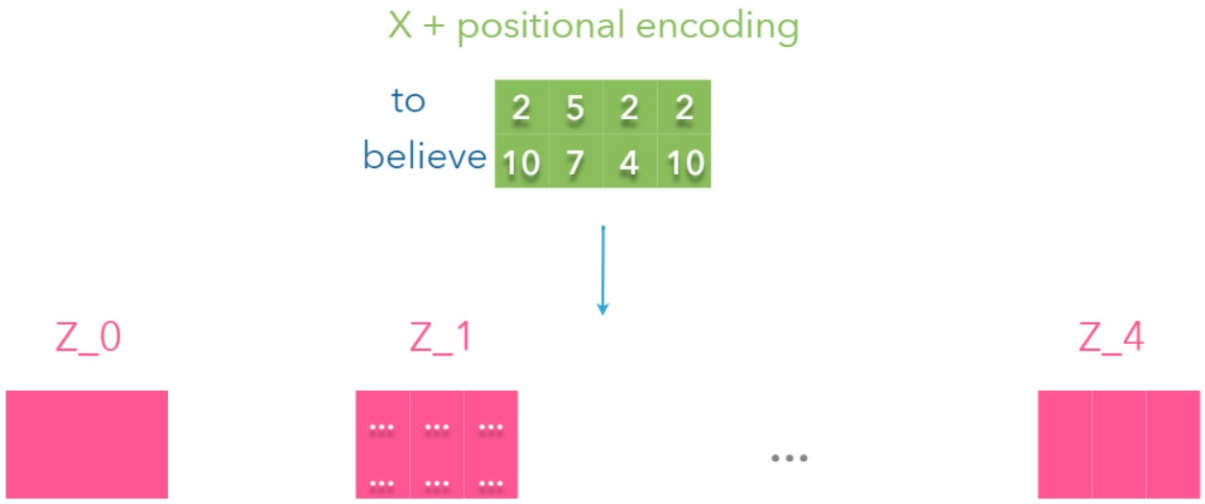

Рис. 8: Конкатенация матриц Z.

И к ним применяется линейное преобразование, таким образом получается матрица, совпадающая по размерам с исходной матрицей.

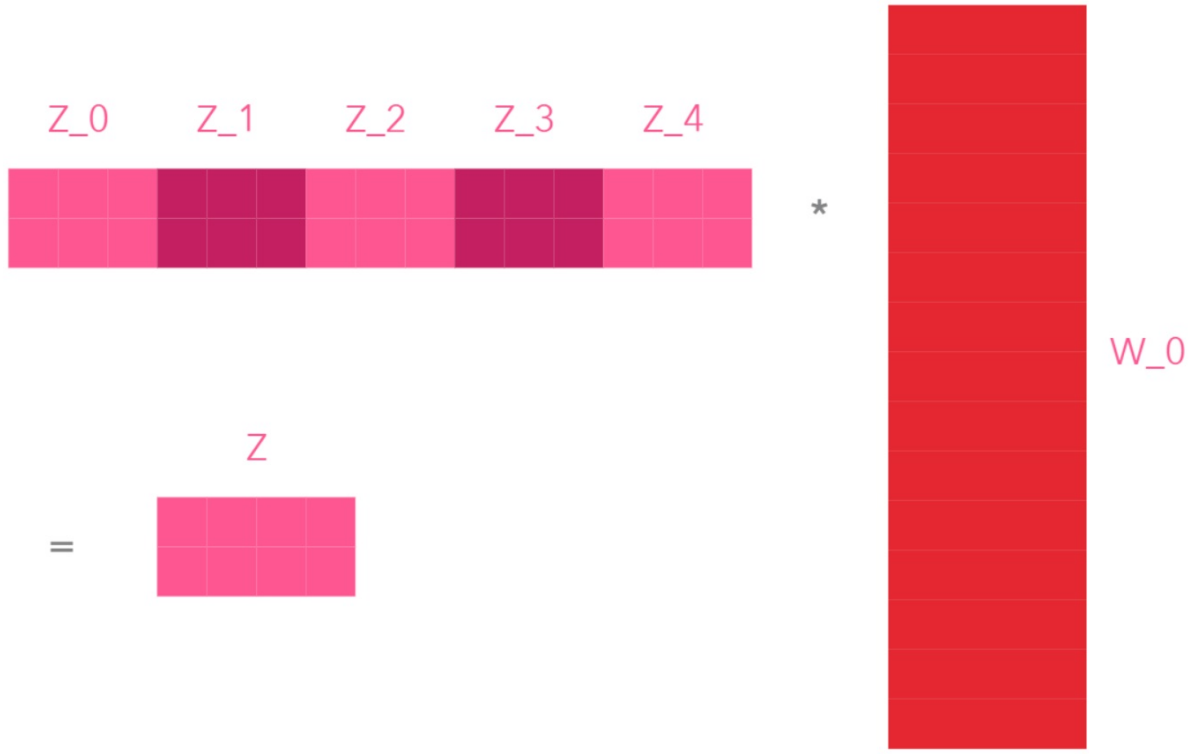

Рис. 9: Операция Linear в блоке Multi-Head Attention.

2.2.5 Residual connections

За блоком Multi-Head Attention следует Residual connections. Эта операция заключается в суммировании матриц, подававшихся на вход блокам Multi-Head Attention и Feed Forward с матрицами, полученными в результате применения данных блоков.

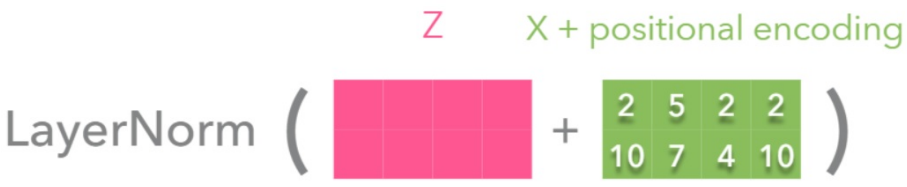

Рис. 10: Операция Add.

2.2.6 Layer normalization

К результату операции Add применяется Layer normalization [5]. Этот метод был разработан Джоффри Хинтоном. В отличие от техники batch-normalization, где считается статистика по батчам, в layer normalization статистика считается по отдельным измерениям входного вектора. Используются следующии выражения:

$$\mu_\beta = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_\beta^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\beta)^2$$

$$\hat{x}_i = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}}$$

$$y_i = \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$$

Рис. 11: Операция Add and Layer Norm.

2.2.7 Полносвязный слой

После блока Multi-Head Attention с последующими операциями Residual connections и Layer normalization следует полносвязный слой. Он описывается следующим выражением:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

Объединяя описанные операции получаем блок Multi-Head Attention + Add and Norm:

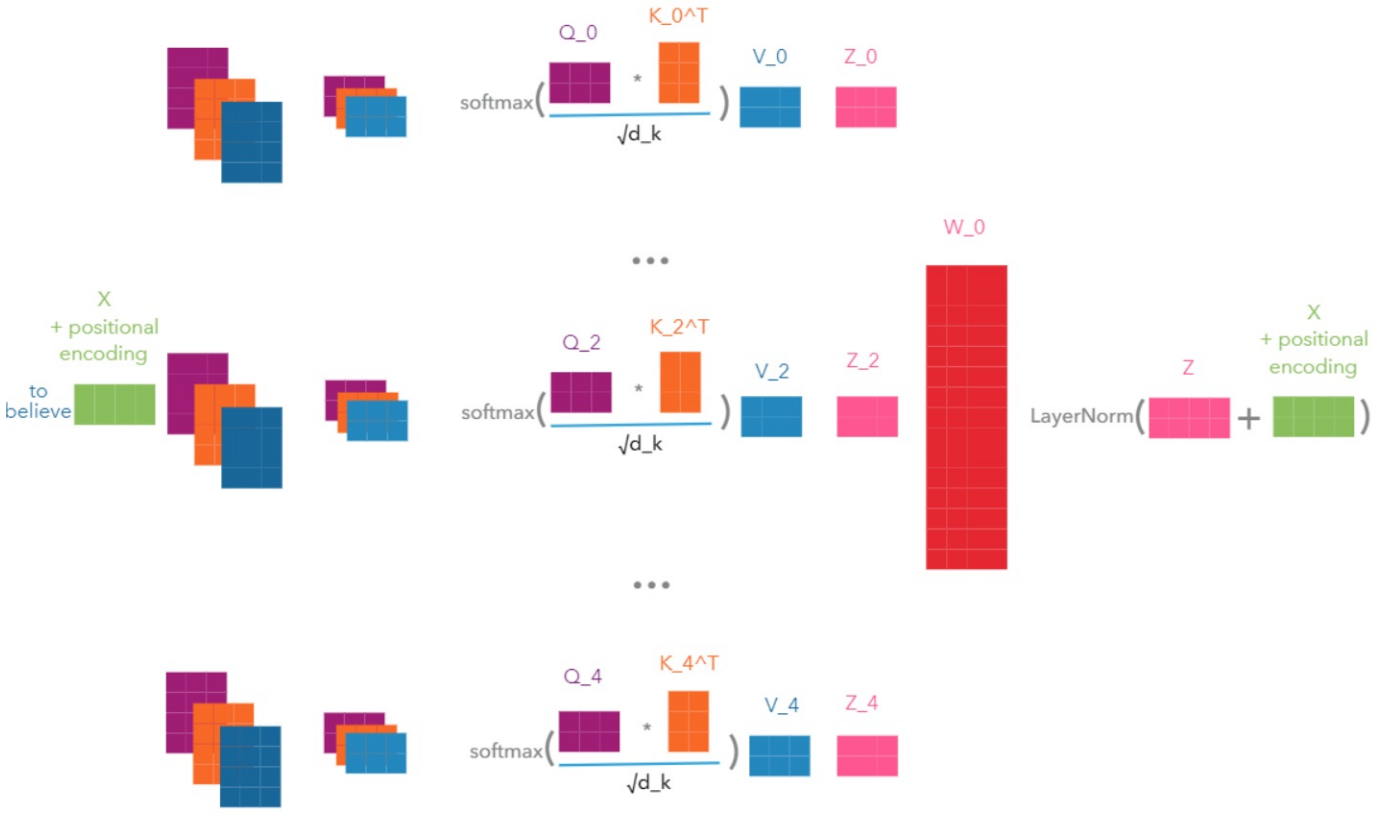

Рис. 12: блок Multi-Head Attention + Add and Norm.

2.2.8 Encoder Layer

Объединяя все описанные операции, получаем слой Encoder:

Рис. 13: Encoder Layer.

2.2.9 Encoder Transformer

Удобно соединить два слоя, поскольку размерность матрицы не меняется:

Рис. 14: Два слоя Encoder Layer.

Объединяя несколько слоев, получаем Encoder:

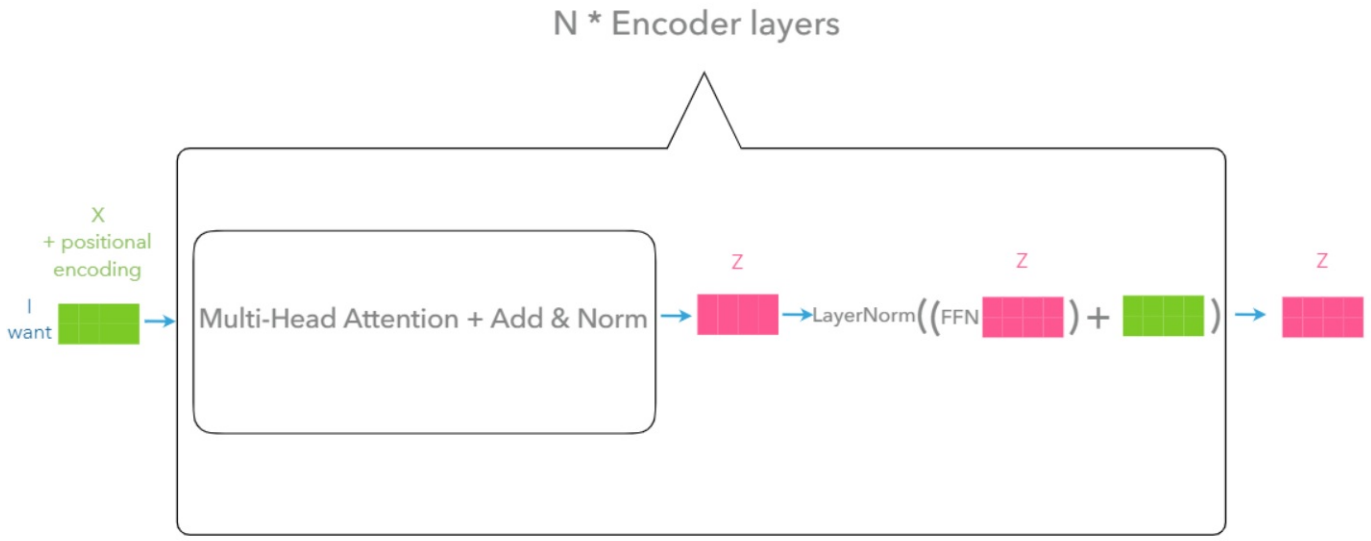

Рис. 15: Encoder.

2.3 Входной вектор

BERT использует Encoder Transformer, описанный выше. Отличается входной вектор. На Рисунке 16 представлен вход BERT.

Рис. 16: Вход BERT.

Помимо векторных представлений токенов используются positional encoding вектор и векторное представление сегментов.

2.3.1 Векторные представления сегментов

На вход BERT подаются пары предложений, разделенные специальным символом. Каждое предложение, в зависимости от того, является оно первым или вторым, имеет своё векторное представление.

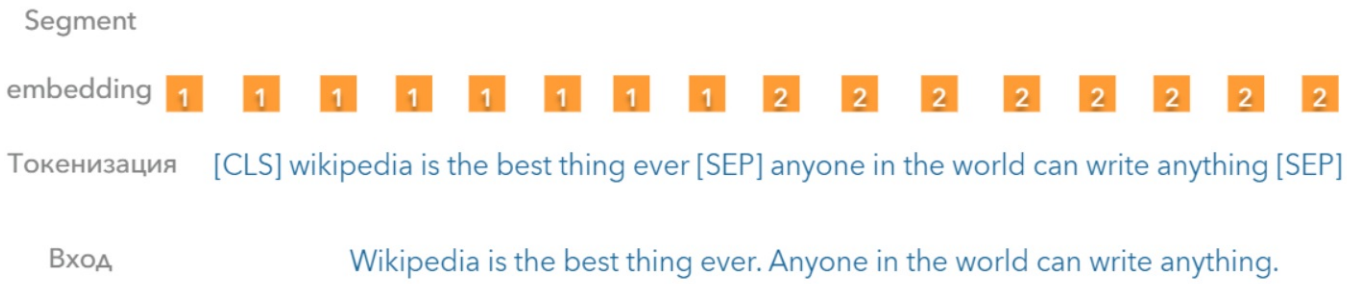

Рис. 17: Вход BERT – Segment Embedding.

2.4 Задачи

2.4.1 Masked Language Model

Для того, чтобы добиться лучших векторных представлений слов, обычную задачу языковой модели модифицируют. При использовании обычной языковой модели предсказывается слово в словаре по предыдущим. То есть нейроннная сеть обучается предсказывать распределение вероятностей элементов словаря по контексту:

$$P(w_t|w_{t-k}, ..., w_{t-1})$$

В BERT используется следующая стратегия:

- Выбирается 15% токенов входной последовательности;
- 80% из этих токенов заменяется на токен [MASK];
- 10% – на случайный токен;
- 10% – токен остается тем же самым;

Далее предсказывается токен, который был помечен [MASK].

Для примера возьмём предложение “Every day, once a day, give yourself a present”. И заменим токены по описаной стратегии:

“Every day once a day give yourself a present” → “Every day once a [MASK] give yourself a present”.

Таким образом, было выбрано 15% токенов – этим токеном оказался “day”. И с вероятностью 0.8 он был заменен на токен [MASK].

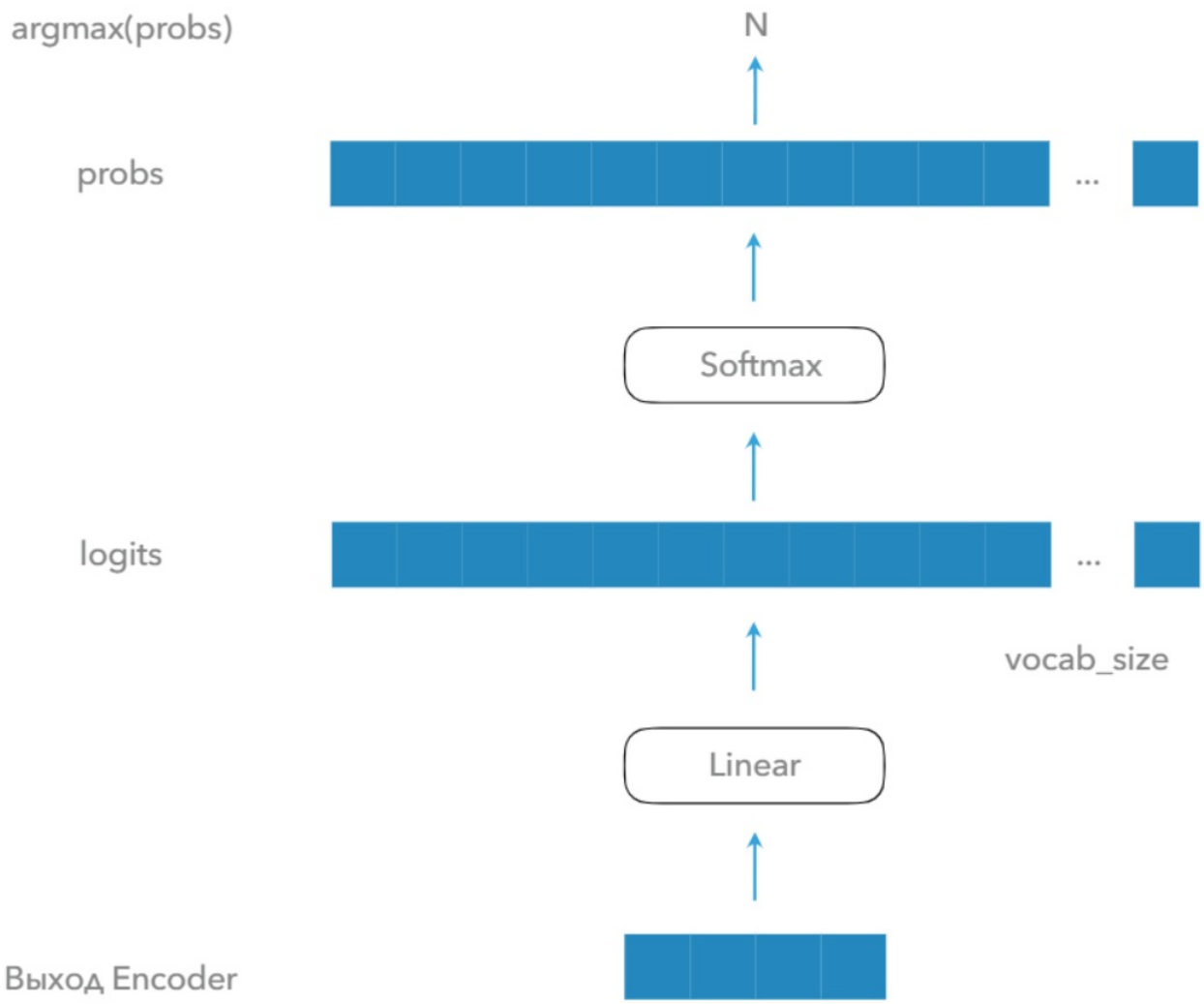

Рис. 18: Masked Language Model.

2.4.2 Next Sentence Prediction

В этой задаче 50% входных предложений заменяется на случайное, а другие 50% – остаются теми же. Решается задача бинарной классиификации – являются ли два предложения последовательными либо нет.

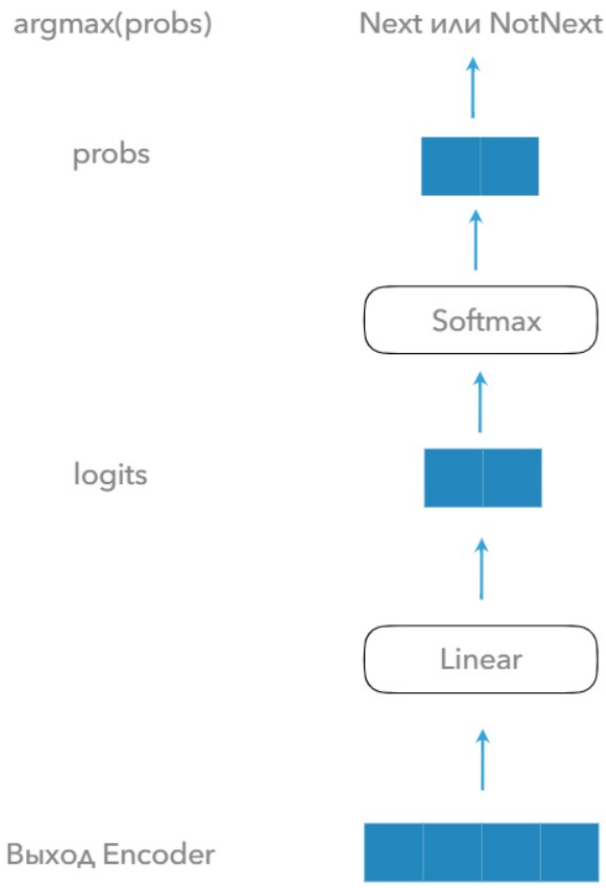

Рис. 19: Next Sentence Prediction.

2.5 Число обучаемых параметров BERT

Введём обозначения:

- d_{model} – размер векторного представления;
- v_{size} – размер словаря;
- n_{layers} – число слоёв;
- n_{heads} – число голов;

Посчитаем число обучаемых параметров на каждом этапе:

- $d_{model} \times v_{size} + 3 \times d_{model}$ – число параметров таблицы векторных представлений токенов и сегментов;
- $(d_{model} \times d_{model} + d_{model}) \times 3 + d_{model} \times d_{model} + 2 \times d_{model}$ – Multi-Head Attention;
- $4 \times d_{model} \times d_{model} + 4 \times d_{model} + 4 \times d_{model} \times d_{model} + d_{model} + 2 \times d_{model}$ – FFN;
- $2 \times d_{model} + 2 + d_{model} \times v_{size} + v_{size}$ – выход сети;

Объединяя все этапы, получаем:

$$N_p = d_{model} \times v_{size} + 3 \times d_{model} + ((d_{model} \times d_{model} + d_{model}) \times 3 + d_{model} \times d_{model} + 2 \times d_{model} + \\ + 4 \times d_{model} \times d_{model} + 4 \times d_{model} + 4 \times d_{model} \times d_{model} + d_{model} + 2 \times d_{model}) \times n_{layers} + 2 \times d_{model} + \\ 2 + d_{model} \times v_{size} + v_{size}$$

Сокращяя, получаем итоговую формулу, описывающее число обучаемых параметров в BERT:

$$N_p = 2 \times d_{model} \times v_{size} + 5 \times d_{model} + (12 \times d_{model} \times d_{model} + 13 \times d_{model}) \times n_{layers} + 2 + v_{size}$$

2.6 Число операций в BERT

Введём обозначения:

- d_{model} – размер векторного представления;
- v_{size} – размер словаря;
- n_{layers} – число слоёв;
- n_{heads} – число голов;
- n_{seq} – длина входной последовательности;

Делая предположение о том, что матричное умножение реализуется по алгоритму, имеющему сложность $O(n^3)$, и, учитывая только операции умножения в матричном умножении, посчитаем число операций, совершаемых на каждом этапе:

- $(n_{seq} \times d_{model} \times (d_{model}/n_{heads}) \times 3 + n_{seq} \times (d_{model}/n_{heads}) \times n_{seq} + n_{seq} \times n_{seq} \times (d_{model}/n_{heads})) \times n_{heads} + n_{seq} \times d_{model} \times d_{model}$ – Multi-Head Attention;
- $n_{seq} \times d_{model} \times (d_{model} \times 4) + n_{seq} \times (d_{model} \times 4) \times d_{model}$ – FFN;
- $n_{seq} \times d_{model} \times 2 + n_{seq} \times d_{model} \times n_{vocab}$ – выход сети;

Объединяя все этапы, получаем:

$$N_o = ((n_{seq} \times d_{model} \times (d_{model}/n_{heads}) \times 3 + n_{seq} \times (d_{model}/n_{heads}) \times n_{seq} + n_{seq} \times n_{seq} \times (d_{model}/n_{heads}) \times n_{heads} + n_{seq} \times d_{model} \times d_{model} + n_{seq} \times d_{model} \times (d_{model} \times 4) + n_{seq} \times (d_{model} \times 4) \times d_{model}) \times \times n_{layers} + n_{seq} \times d_{model} \times 2 + n_{seq} \times d_{model} \times n_{vocab}$$

Сокращяя, получаем итоговую формулу, описывающее число операций в BERT:

$$N_o = n_{seq} \times d_{model} \times n_{layers} \times (3 \times d_{model} + 2 \times n_{seq} + 9) + n_{seq} \times d_{model} \times (2 + n_{vocab})$$

3 Реализация и эксперимент

3.1 Язык программирования и библиотеки

В ходе реализации BERT были использованы следующие инструменты:

- Язык программирования Python
- PyTorch – фреймворк для глубокого обучения от Facebook
- math – библиотека, содержащая математические операции
- numpy – матричные операции
- pickle – сериализация и десериализация объектов для хранения
- os – работа с файловой системой
- tensorboard – визуализация работы нейронной сети
- Google Colab – облачный сервис с доступом к GPU

3.2 Набор данных

Для предобучения использовался набор данных WikiSplit Dataset состоящий из 989944 пар предложений из Википедии с размером словаря 632588 и количеством токенов – 33084465, для валидации – 5000 пар предложений, содержащих 166628 токенов словарь с 25251 токенами.

Для задачи анализа тональности текста использовался такой же набор данных как и в оригинальной статье. Набор данных для классификации содержит 6920 предложений, имеющих положительную или отрицательную метку, и 872 предложения в валидационном наборе.

3.3 Параметры

Для обучения использовалось 3 слоя, с количеством голов равным 3, входная последовательность ограничена 64 токенами, размера батча – 16, размер словаря – 66641. Обучение производилось в течение 16 эпох, использовался оптимизатор AdamW [6].

3.4 Результаты

3.4.1 Pre-training

В ходе предобучения модели была достигнута точность – 85.17 для задачи Next Sentence Prediction на тестовой выборке, значение общей целевой функции на тестовой выборке - 5.01.

Обучение модели происходило в течение 20 часов, каждую эпоху модель сохранялась.

3.4.2 Задача классификации текстов

Таблица 1: Точность на валидацинной выборке SST-2 в процентах.

Модель	Эпохи	Weight decay	SST-2(%)
Baseline	6	0.01	66.40
Pretrained 8 epoch	6	0.01	65.60
Baseline	13	0.05	68.71
Pretrained 8 epoch	13	0.05	70.16
Baseline	13	0.05	70.07
Pretrained 8 epoch	13	0.05	72.25
Baseline	17	0.05	70.18
Pretrained 14 epoch	17	0.05	75.69

В ходе решения задачи классификации текстов предобученные модели показывают точность выше, чем модели с начальной инициализацией весов.

3.4.3 Эксперимент по изменение параметров

В ходе эксперимента по изменению числа обучаемых параметров использовался словарь размером в 6353 слова, набор данных WikiSplit Dataset в 500 000 предложений, длина входной последовательности – 64, число голов – 3, число эпох обучения – 10.

Эксперимент заключался в исследовании влияния концентрации параметров в том или ином элементе BERT, при этом общее число обучаемых параметров в каждом случае оставалось неизменным.

Использовались следующие варианты для экспериментов:

- 1 слой, параметр d_{model} – 110, число параметров – 1 550 309
- 2 слоя, параметр d_{model} – 102, число параметров – 1 555 225.
- 3 слоя, параметр d_{model} – 96, число параметров – 1 562 131.
- 4 слоя, параметр d_{model} – 90, число параметров - 1 543 825.

На рисунках представлены результаты обучения в каждом случае, нумерация экспериментов идет по часовой стрелке.

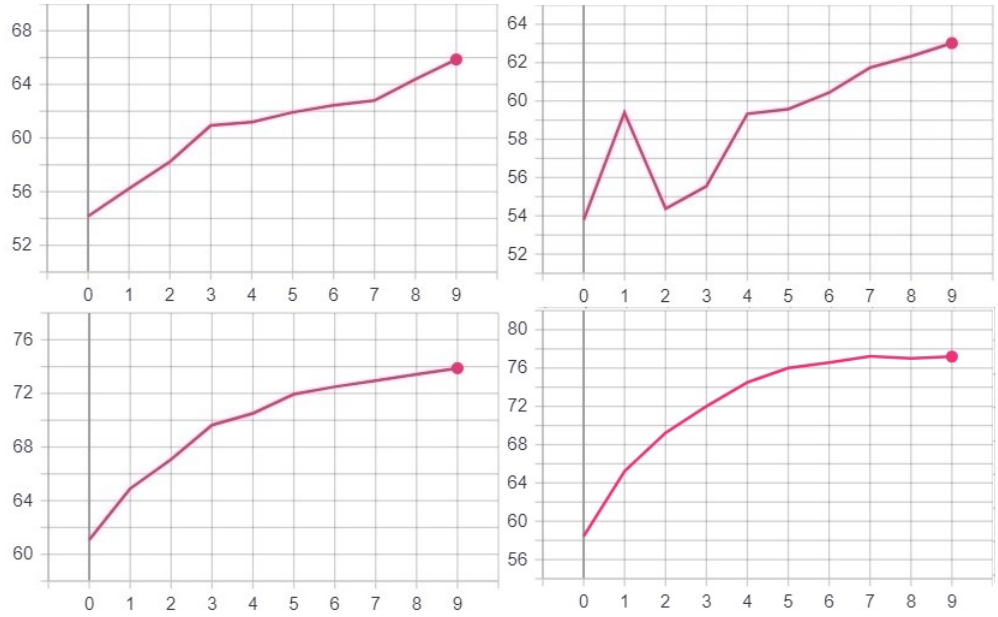

Рис. 20: Точность на тестовой выборке.

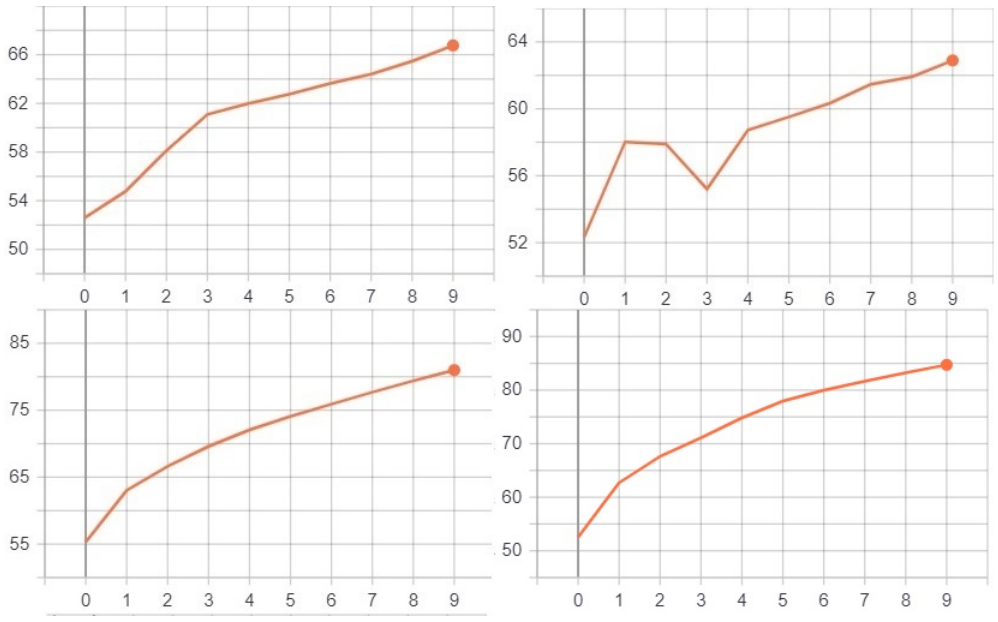

Рис. 21: Точность на тренировочной выборке.

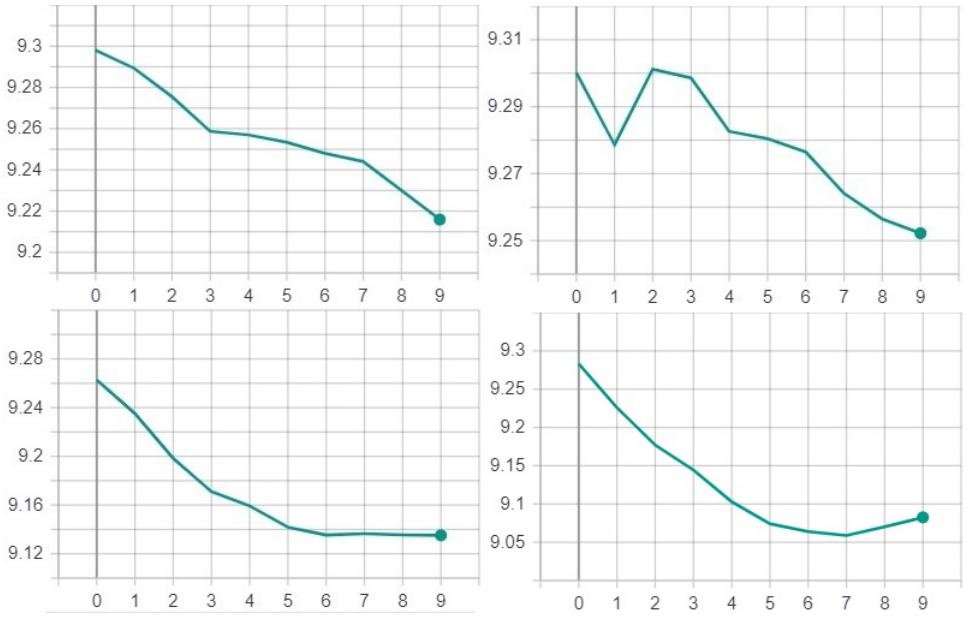

Рис. 22: Значение общей целевой функции на тестовой выборке.

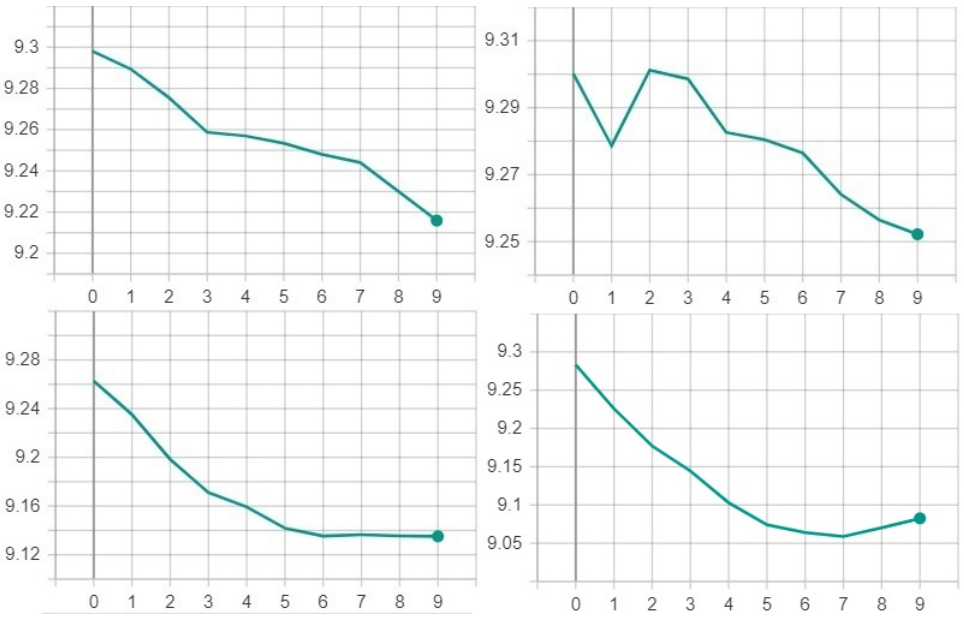

Рис. 23: Значение общей целевой функции на тестовой выборке.

Таблица 2: Результаты экспериментов по изменению числа обучаемых параметров.

Число слоёв	d_{model}	Число параметров	Время обучения (мин:сек)	Точность на тестовой выборке(%)	Точность на тренировочной выборке(%)	Значение общей целевой функции на тестовой выборке	Значение общей целевой функции на тренировочной выборке
1	110	1 550 309	30:42	66	67	9.22	9.22
2	102	1 555 225	34:30	63	63	9.25	9.25
3	96	1 562 131	38:09	74	81	9.14	9.14
4	90	1 543 825	51:16	77	85	9.09	9.09

В таблице 2 представлены результаты. Исходя из результатов, можно сделать вывод о том, что модель с большим количеством слоёв при одинаковом числе параметров, даёт результаты лучше, чем модель с меньшим числом слоёв. То есть модель показывает лучше точность на задаче предсказания являются ли предложения последовательными и имеет ниже значения целевой функции, но при этом время обучения возрастает с увеличением числа слоёв.

3.4.4 Эксперимент с изменением числа операций

В ходе эксперимента с изменением числа операций в BERT использовалось аналигично предыдущему эксперименту – словарь размером в 6353 слова, набор данных WikiSplit Dataset в 500 000 предложений, длина входной последовательности – 64, число голов – 3.

Эксперимент заключался в исследовании влияния концентрации совершаемых операций в том или ином элементе BERT, при этом время на обучение и число операций оставались неизменными.

Использовались следующие варианты для экспериментов:

- 1 слой, параметр d_{model} – 110
- 2 слоя, параметр d_{model} – 104
- 3 слоя, параметр d_{model} – 98
- 4 слоя, параметр d_{model} – 94

На рисунках представлены результаты обучения в каждом случае, нумерация экспериментов идет по часовой стрелке.

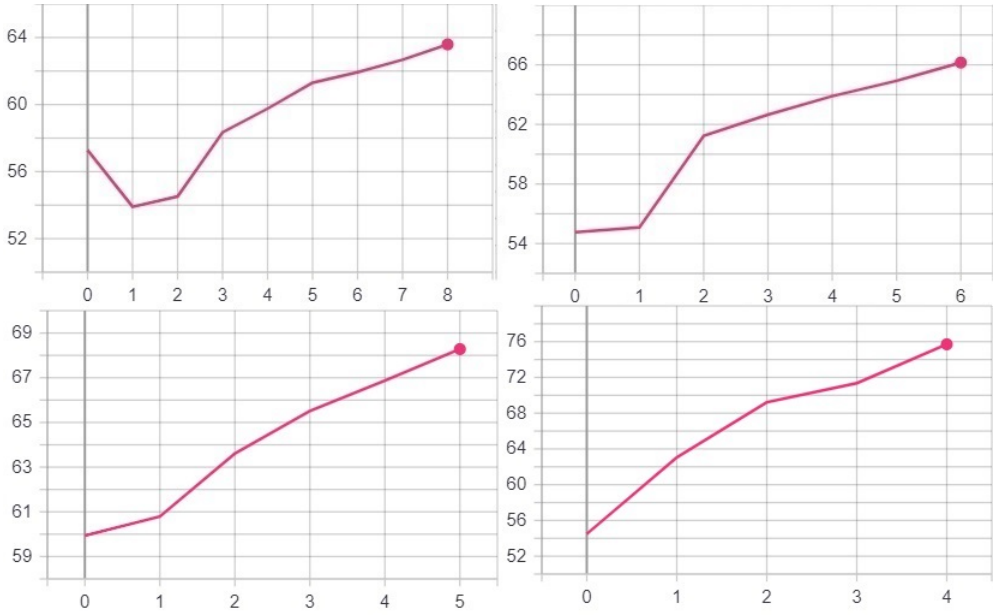

Рис. 24: Точность на тестовой выборке.

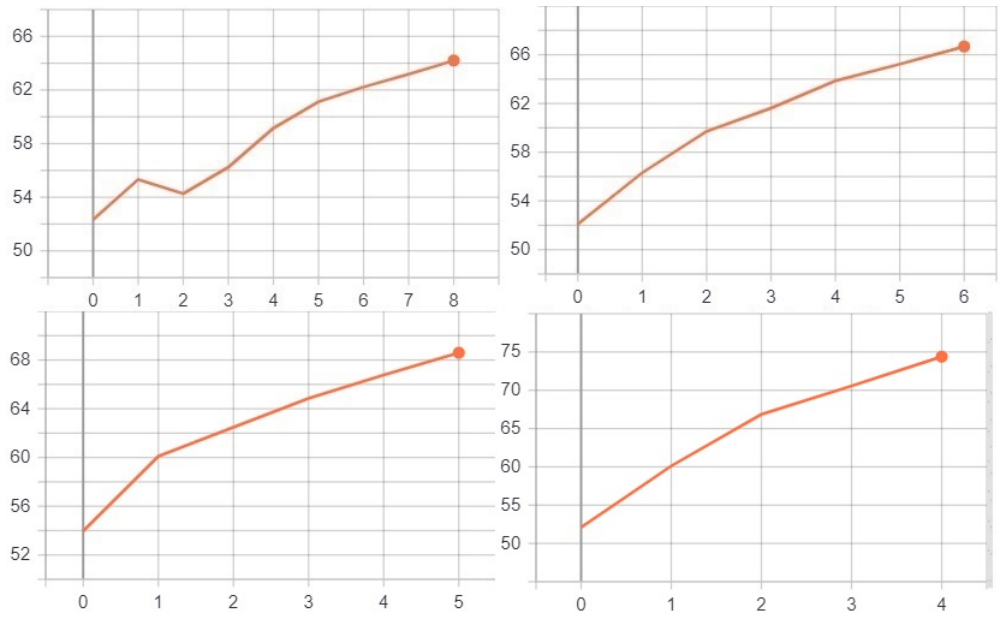

Рис. 25: Точность на тренировочной выборке.

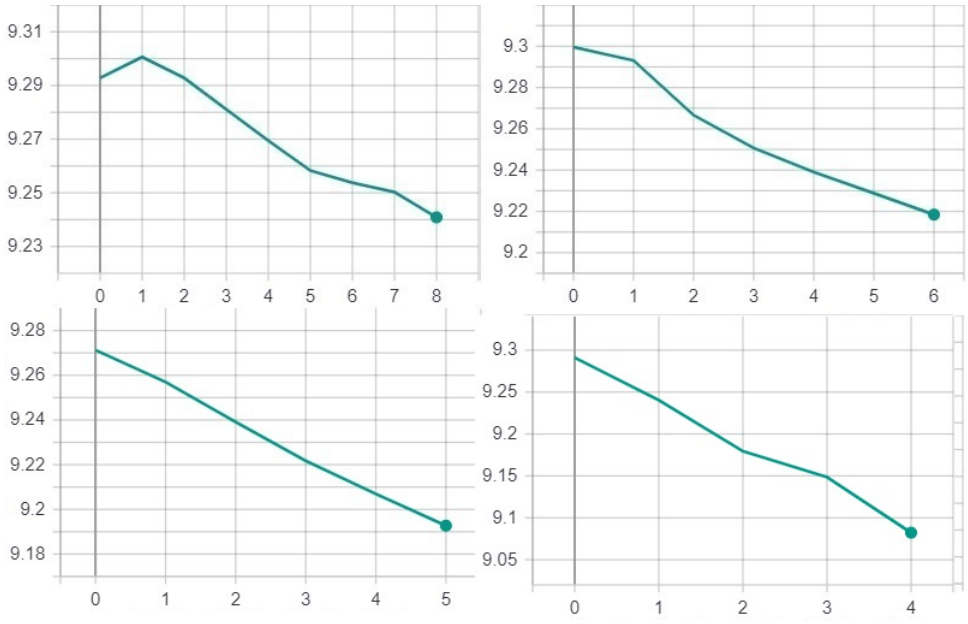

Рис. 26: Значение общей целевой функции на тестовой выборке.

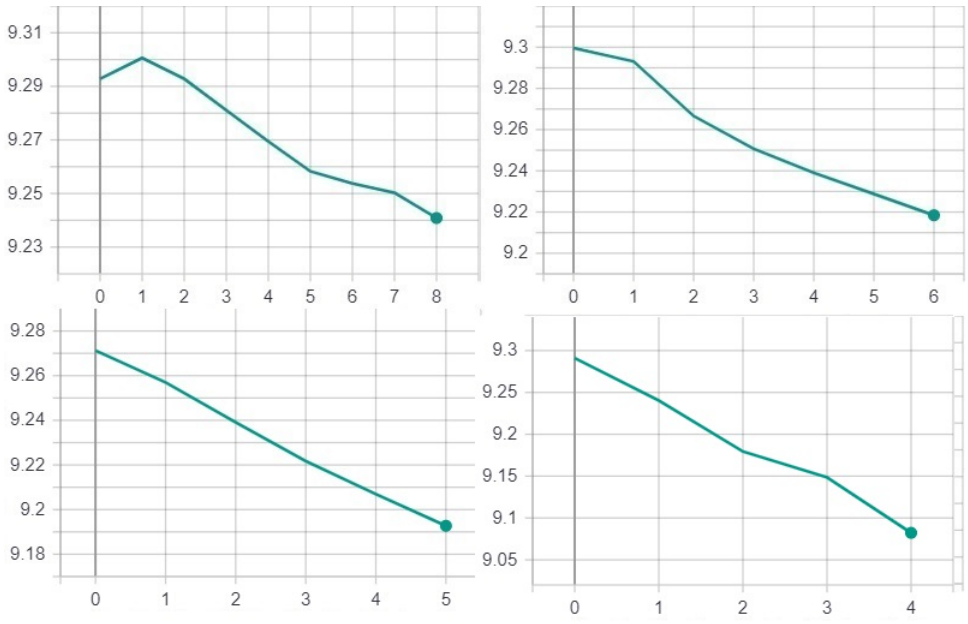

Рис. 27: Значение общей целевой функции на тестовой выборке.

Таблица 3: Результаты экспериментов с изменением числа операций.

Число слоёв	d_{model}	Число эпох	Время обучения (мин:сек)	Точность на тестовой выборке(%)	Точность на тренировочной выборке(%)	Значение общей целевой функции на тестовой выборке	Значение общей целевой функции на тренировочной выборке
1	110	8	30:00	64	64	9.24	9.24
2	104	6	30:00	66	66	9.22	9.22
3	98	5	30:00	68	68	9.19	9.19
4	94	4	30:00	74	75	9.10	9.08

В таблице 3 представлены результаты. На основе результатов можно сделать вывод о том, что модель с большим количеством слоёв при одинаковом числе операций достигает лучших результатов - то есть модель показывает лучше точность на задаче предсказания являются ли предложения последовательными и имеет ниже значения целевой функции, при этом время обучения в ходе эксперимента не меняется с увеличением числа слоёв.

4 Заключение

В ходе курсовой работы был изучен и реализован BERT, проведено предобучение на наборе данных WikiSplit Dataset, предобученная модель была использована для задачи классификации текстов на наборе данных SST-2, для которой удалось добиться точности в 75.69 %.

Также были проведены эксперименты с изменением числа обучаемых параметров и числа операций в BERT. Из экспериментов был сделан вывод о том, что модели с большим числом слоёв показывают результаты лучше в плане точности классификации и скорости сходимости при двух разных условиях экспериментов - при одинаковом числе параметров в модели и при одинаковом числе операций.

Список литературы

[1] Deep contextualized word representations. Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer, 2018.

[2] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova, 2018.

[3] Efficient Estimation of Word Representations in Vector Space. Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, 2013.

[4] Attention Is All You Need. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, 2017.

[5] Layer Normalization. Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton, 2016.

[6] Decoupled Weight Decay Regularization. Ilya Loshchilov, Frank Hutter, 2017.

Приложение 1

```
import torch
import random
from collections import Counter
from math import sqrt
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torch.optim import Adam
from sklearn.model_selection import train_test_split
import math
import numpy as np
import pickle


class BERTDataset:
    def __init__(self, corpus_path, vocab, seq_len):
        """
        :param corpus_path: path to text corpus
        :param vocab: vocab object
        :param seq_len: len of input sequence
        """
        self.vocab = vocab
        self.seq_len = seq_len

        self.corpus_path = corpus_path

        with open(corpus_path, "r") as f:
```

```
        self.lines = [line.replace("\n", "").split("\\t") for line in f]

def __len__(self):
    return len(self.lines)

def __getitem__(self, item):
    t1, t2, is_next_label = self.random_sent(item)
    t1_random, t1_label = self.random_word(t1)
    t2_random, t2_label = self.random_word(t2)

    t1 = [self.vocab.cls_index] + t1_random + [self.vocab.sep_index]
    t2 = t2_random + [self.vocab.sep_index]

    t1_label = [self.vocab.pad_index] \
               + t1_label + [self.vocab.pad_index]
    t2_label = t2_label + [self.vocab.pad_index]

    segment_label = ([1 for _ in range(len(t1))] \
                     + [2 for _ in range(len(t2))])[:self.seq_len]
    bert_input = (t1 + t2)[:self.seq_len]
    bert_label = (t1_label + t2_label)[:self.seq_len]

    padding = [self.vocab.pad_index for _ in \
               range(self.seq_len - len(bert_input))]

    bert_input += padding
    bert_label += padding
    segment_label += padding
```

```
        output = {"bert_input": bert_input,
                  "bert_label": bert_label,
                  "segment_label": segment_label,
                  "is_next": is_next_label}

        return {key: torch.tensor(value) for key, value in output.items()}

    def random_word(self, sentence):
        tokens = sentence.split()
        output_label = []

        for i, token in enumerate(tokens):
            prob = random.random()
            if prob < 0.15:
                prob /= 0.15

                if prob < 0.8:
                    tokens[i] = self.vocab.mask_index
                elif prob < 0.9:
                    tokens[i] = random.randrange(len(self.vocab))
                else:
                    tokens[i] = self.vocab.token_to_index\
                        .get(token, self.vocab.unk_index)

                output_label.append(self.vocab.token_to_index\
                                    .get(token, self.vocab.unk_index))
            else:
                tokens[i] = self.vocab.token_to_index\
                    .get(token, self.vocab.unk_index)
```

```
                output_label.append(self.vocab.pad_index)

        return tokens, output_label

    def random_sent(self, index):
        t1, t2 = self.get_corpus_line(index)

        if random.random() > 0.5:
            return t1, t2, 1
        else:
            return t1, self.get_random_line(), 0

    def get_corpus_line(self, item):
        return self.lines[item][0], self.lines[item][1]

    def get_random_line(self):
        return self.lines[random.randrange(len(self.lines))][1]


class Vocab:
    def __init__(self, text):
        """
        :param text: input text file
        """
        self.specials = ["<pad>", "<unk>", "<sep>", "<cls>", "<mask>"]

        self.pad_index = 0
        self.unk_index = 1
        self.sep_index = 2
```

```
        self.cls_index = 3
        self.mask_index = 4

        self.index_to_token = list(self.specials)

        counter = Counter()

        for line in text:
            words = line.replace("\n", "").replace("\\t", "").split()

            for word in words:
                counter[word] += 1

        words_and_frequencies = sorted(counter.items())

        for word, freq in words_and_frequencies:
            if (freq > 200):
                self.index_to_token.append(word)

        self.token_to_index = {token: i for i, token \
                               in enumerate(self.index_to_token)}

    def __len__(self):
        return len(self.index_to_token)


class ScaledDotProductAttention(nn.Module):
    def __init__(self, d_k):
        """
```

```
        :param d_k: int scaling factor
        """
        super(ScaledDotProductAttention, self).__init__()

        self.scaling = 1 / (sqrt(d_k))

    def forward(self, q, k, v, mask):
        """
        :param q:  An float tensor
        with shape of [b_s, seq_len, d_model / n_head]
        :param k: An float tensor
        with shape of [b_s, seq_len, d_model / n_head]
        :param v: An float tensor
        with shape of [b_s, seq_len, d_model / n_head]

        :return: An float tensor
        with shape of [b_s, seq_len, d_model / n_head]
        """
        attention = torch.bmm(q, k.transpose(1, 2)) * self.scaling
        attention = attention.masked_fill(mask == 0, -1e9)

        attention = F.softmax(attention, dim=2)

        output = torch.bmm(attention, v)

        return output


class SingleHeadAttention(nn.Module):
```

```
def __init__(self, d_model, d_k, d_v):
    """
    :param d_model: Int
    :param d_k: Int = d_model / n_head
    :param d_v: Int = d_model / n_head
    """
    super(SingleHeadAttention, self).__init__()

    self.q_linear = nn.Linear(d_model, d_k)
    self.k_linear = nn.Linear(d_model, d_k)
    self.v_linear = nn.Linear(d_model, d_v)

    self.attention = ScaledDotProductAttention(d_k)

def forward(self, q, k, v, mask):
    """
    :param q: An float tensor with shape of [b_s, seq_len, d_model]
    :param k: An float tensor with shape of [b_s, seq_len, d_model]
    :param v: An float tensor with shape of [b_s, seq_len, d_model]

    :return: An float tensor
    with shape of [b_s, seq_len, d_model / n_heads]
    """
    proj_q = self.q_linear(q)
    proj_k = self.k_linear(k)
    proj_v = self.v_linear(v)

    output = self.attention(proj_q, proj_k, proj_v, mask)
```

```
        return output


class MultiHeadAttention(nn.Module):
    def __init__(self, n_head, d_model):
        """
        :param n_head: Int number of heads
        :param d_model: Int
        """
        super(MultiHeadAttention, self).__init__()

        d_v = int(d_model / n_head)
        d_k = int(d_model / n_head)

        self.attention = nn.ModuleList([SingleHeadAttention\
                        (d_model, d_k, d_v) for _ in range(n_head)])

        self.Linear = nn.Linear(n_head * d_v, d_model)

    def forward(self, q, k, v, mask):
        """
        :param q: An float tensor with shape of [b_s, seq_len, d_model]
        :param k: An float tensor with shape of [b_s, seq_len, d_model]
        :param v: An float tensor with shape of [b_s, seq_len, d_model]

        :return: An float tensor with shape of [b_s, seq_len, d_model]
        """
        results = []
```

```
        for i, single_attention in enumerate(self.attention):
            attention_out = single_attention(q, k, v, mask)
            results.append(attention_out)

        concat = torch.cat(results, dim=2)

        linear_output = self.Linear(concat)

        return linear_output


class TokenEmbedding(nn.Embedding):
    def __init__(self, vocab_size, emb_size):
        super().__init__(vocab_size, emb_size)


class SegmentEmbedding(nn.Embedding):
    def __init__(self, emb_size):
        super().__init__(3, emb_size)


class PositionalEmbedding(nn.Module):
    def __init__(self, d_model, max_len=512):
        super().__init__()

        pe = torch.zeros(max_len, d_model).float()
        pe.require_grad = False

        position = torch.arange(0, max_len).float().unsqueeze(1)
```

```
        div_term = torch.pow(10000, torch.arange\
            (0, d_model, 2).float() / d_model)

        pe[:, 0::2] = torch.sin(position / div_term)
        pe[:, 1::2] = torch.cos(position / div_term)

        self.pe = pe.unsqueeze(0)

    def forward(self, x):
        return self.pe[:, :x.size(1)].to('cuda')


class BERTEmbedding(nn.Module):
    def __init__(self, vocab_size, emb_size):
        """
        :param vocab_size: Int size of vocabulary
        :param emb_size: Int size of embedding
        """
        super(BERTEmbedding, self).__init__()

        self.v_s = vocab_size
        self.e_s = emb_size

        self.token = TokenEmbedding(self.v_s, self.e_s)
        self.segment = SegmentEmbedding(self.e_s)
        self.position = PositionalEmbedding(self.e_s)

    def forward(self, seq, segment_label):
        """
```

```
        :param seq: An long tensor with shape of [b_s, seq_len]

        :return: An float tensor with shape of [b_s, seq_len, emb_size]
        """

        return self.token(seq) + self.segment(segment_label)\
               + self.position(seq)


class GELU(nn.Module):
    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi)\
                * (x + 0.044715 * torch.pow(x, 3))))


class PositionWise(nn.Module):
    def __init__(self, size, inner_size):
        """
        :param size: Int input size
        :param inner_size: Int inner size of position wise
        """
        super(PositionWise, self).__init__()

        self.fc = nn.Sequential(
            nn.Linear(size, inner_size),
            GELU(),
            nn.Linear(inner_size, size)
        )
```

```
        self.layer_norm = nn.LayerNorm(size)

    def forward(self, input):
        """
        :param input: An float tensor with shape of [b_s, seq_len, emb_size]

        :return: An float tensor with shape of [b_s, seq_len, emb_size]
        """
        residual = input

        result = self.fc(input)

        return self.layer_norm(result + residual)


class Encoder(nn.Module):
    def __init__(self, embeddings, d_model, n_heads, n_layers, vocab_s):
        """
        :param embeddings: An float embeddings tensor \
        with shape [b_s, seq_len, d_model]
        :param d_model: Int size of input
        :param n_heads: Int number of heads
        :param vocab_s: Int size of vocabulary
        """
        super(Encoder, self).__init__()

        self.embeddings = embeddings
        self.vocab_s = vocab_s
```

```
        self.transformer_blocks = nn.ModuleList\
            ([nn.Sequential(MultiHeadAttention(n_heads, d_model),
              nn.LayerNorm(d_model),
              PositionWise(d_model, d_model * 4)) for _ in
              range(n_layers)])

    def forward(self, x, segment_label):
        """
        :param input: An long tensor with shape of [b_s, seq_len]

        :return: An float tensor with shape of [b_s, seq_len, vocab_size]
        """

        mask = (x > 0).unsqueeze(1).repeat(1, x.size(1), 1)
        input = self.embeddings(x, segment_label)

        for multi_head_block, layer_norm, position_wise \
                in self.transformer_blocks:
            input = layer_norm(input +\
                    multi_head_block(q=input, k=input, v=input, mask=mask))
            input = position_wise(input)

        return input


class Model(nn.Module):
    def __init__(self, n_heads, n_layers, vocab_size, emb_size):
        """
        :param n_heads: Int number of heads
```

```
        :param vocab_size: Int size of vocabulary
        :param emb_size: Int embedding size
        """
        super(Model, self).__init__()

        self.embed = BERTEmbedding(vocab_size, emb_size)

        self.d_model = self.embed.e_s
        self.v_s = self.embed.v_s

        self.encoder = Encoder(self.embed, self.d_model,\
                               n_heads, n_layers, self.v_s)
        self.next_sentence = NextSentencePrediction(self.d_model)
        self.mask_lm = MaskedLanguageModel(self.d_model, self.v_s)

    def forward(self, x, segment_label):
        """
        :param x: An float tensor with shape of [b_s, seq_len]
        :param segment_label: An float tensor with shape of [b_s, seq_len]
        """
        prediction = self.encoder(x, segment_label)

        return self.next_sentence(prediction), self.mask_lm(prediction)


class NextSentencePrediction(nn.Module):
    def __init__(self, d_model):
        """
        :param d_model: Int
```

```
        """
        super().__init__()
        self.linear = nn.Linear(d_model, 2)
        self.softmax = nn.LogSoftmax(dim=-1)

    def forward(self, x):
        return self.softmax(self.linear(x[:, 0]))


class MaskedLanguageModel(nn.Module):
    def __init__(self, d_model, vocab_size):
        """
        :param d_model: Int
        :param vocab_size: Int size of vocabulary
        """
        super().__init__()
        self.linear = nn.Linear(d_model, vocab_size)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        return self.softmax(self.linear(x))


class AdamW(torch.optim.Optimizer):
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8,
                 weight_decay=0):
        defaults = dict(lr=lr, betas=betas, eps=eps,
                        weight_decay=weight_decay)
        super(AdamW, self).__init__(params, defaults)
```

```
def step(self, closure=None):
    loss = None
    if closure is not None:
        loss = closure()

    for group in self.param_groups:
        for p in group['params']:
            if p.grad is None:
                continue
            grad = p.grad.data

            state = self.state[p]

            # State initialization
            if len(state) == 0:
                state['step'] = 0
                state['exp_avg'] = torch.zeros_like(p.data)
                state['exp_avg_sq'] = torch.zeros_like(p.data)

            exp_avg, exp_avg_sq = state['exp_avg'], state['exp_avg_sq']
            beta1, beta2 = group['betas']

            state['step'] += 1

            exp_avg.mul_(beta1).add_(1 - beta1, grad)
            exp_avg_sq.mul_(beta2).addcmul_(1 - beta2, grad, grad)

            denom = exp_avg_sq.sqrt().add_(group['eps'])
```

```
                bias_correction1 = 1 - beta1 ** state['step']
                bias_correction2 = 1 - beta2 ** state['step']
                step_size = group['lr'] * \
                            math.sqrt(bias_correction2) / bias_correction1

                # w = w - wd * lr * w
                if group['weight_decay'] != 0:
                    p.data.add_(-group['weight_decay']\
                                * group['lr'], p.data)

                # w = w - lr * w.grad
                p.data.addcdiv_(-step_size, exp_avg, denom)

        return loss


class ScheduledOptim():
    '''A simple wrapper class for learning rate scheduling'''

    def __init__(self, optimizer, d_model, n_warmup_steps):
        self._optimizer = optimizer
        self.n_warmup_steps = n_warmup_steps
        self.n_current_steps = 0
        self.init_lr = np.power(d_model, -0.5)

    def step_and_update_lr(self):
        "Step with the inner optimizer"
        self._update_learning_rate()
```

```
        self._optimizer.step()

    def zero_grad(self):
        "Zero out the gradients by the inner optimizer"
        self._optimizer.zero_grad()

    def _get_lr_scale(self):
        return np.min([
            np.power(self.n_current_steps, -0.5),
            np.power(self.n_warmup_steps, -1.5) * self.n_current_steps])

    def _update_learning_rate(self):
        ''' Learning rate scheduling per step '''

        self.n_current_steps += 1
        lr = self.init_lr * self._get_lr_scale()

        for param_group in self._optimizer.param_groups:
            param_group['lr'] = lr


dataset_path = "/content/gdrive/My Drive/data.txt"

with open(dataset_path, "r") as f:
    vocab = Vocab(f)

# with open("/content/vocab.pickle", "wb") as f:
#     pickle.dump(vocab, f)
```

```
# with open('/content/vocab.pickle', 'rb') as f:
#     vocab = pickle.load(f)


print(len(vocab))

seq_len = 64
emb_size = 94
epochs = 10
n_layers = 4

dataset = BERTDataset(dataset_path, vocab, seq_len)

train, test = train_test_split(dataset, test_size=0.2)

train_data_loader = DataLoader(train, batch_size=128, shuffle=True)
test_data_loader = DataLoader(test, batch_size=128)
# model = torch.load("/content/model_epoch_7.pth").to('cuda')
model = Model(3, n_layers, len(vocab), emb_size).to('cuda')
masked_criterion = nn.CrossEntropyLoss(ignore_index=0)
next_criterion = nn.CrossEntropyLoss()
optim = AdamW(model.parameters(), lr=1e-5, weight_decay=0.01)
optim_schedule = ScheduledOptim(optim, emb_size, n_warmup_steps=10000)

train_loss = []
test_loss = []
train_acc = []
test_acc = []
```

```
# writer = SummaryWriter()

for l in range(epochs):
    print("Train")

    avg_loss = 0.0
    avg_next_loss = 0.0
    avg_mask_loss = 0.0
    total_correct = 0
    total_element = 0

    for i, data in enumerate(train_data_loader):
        # print(i / len(train_data_loader))
        data = {key: value.to('cuda') for key, value in data.items()}

        next_sent_output, mask_lm_output = model.forward\
            (data["bert_input"], data["segment_label"])

        next_loss = next_criterion(next_sent_output, data["is_next"])
        mask_loss = masked_criterion(mask_lm_output.transpose(1, 2),\
                                     data["bert_label"])

        # print("Next loss = " + str(next_loss))
        # print("Mask loss = " + str(mask_loss))

        loss = next_loss + mask_loss

        correct = next_sent_output.argmax(dim=-1)\
            .eq(data["is_next"]).sum().item()
```

```
        avg_next_loss += next_loss.item()
        avg_mask_loss += mask_loss.item()
        avg_loss += loss.item()
        total_correct += correct
        total_element += data["is_next"].nelement()

        optim_schedule.zero_grad()
        loss.backward()
        optim_schedule.step_and_update_lr()

    train_loss.append(avg_loss)
    train_acc.append(total_correct * 100.0 / total_element)

    print('Avg loss train ' + str(avg_loss / (i + 1)) + " epoch " + str(l))
    print('Acc train ' + str(total_correct * 100.0\
                    / total_element) + " epoch " + str(l))

    tb.save_value('Avg loss train', 'avg_loss_train',\
                  l, avg_loss / (i + 1))
    tb.save_value('Avg next loss train', 'avg_next_loss_train', l\
                  , avg_next_loss / (i + 1))
    tb.save_value('Avg mask loss train', 'avg_mask_loss_train', l\
                  , avg_mask_loss / (i + 1))
    tb.save_value('Acc train', 'acc_train', l, total_correct \
                  * 100.0 / total_element)

    avg_loss = 0.0
    avg_next_loss = 0.0
```

```
avg_mask_loss = 0.0
total_correct = 0
total_element = 0

print("Test")

for i, data in enumerate(test_data_loader):
    data = {key: value.to('cuda') for key, value in data.items()}

    next_sent_output, mask_lm_output = model.forward\
        (data["bert_input"], data["segment_label"])

    next_loss = next_criterion(next_sent_output, data["is_next"])
    mask_loss = masked_criterion\
        (mask_lm_output.transpose(1, 2), data["bert_label"])

    loss = next_loss + mask_loss
    loss = next_loss + mask_loss

    correct = next_sent_output.argmax(dim=-1)\
        .eq(data["is_next"]).sum().item()

    avg_next_loss += next_loss.item()
    avg_mask_loss += mask_loss.item()
    avg_loss += loss.item()
    total_correct += correct
    total_element += data["is_next"].nelement()

test_loss.append(avg_loss)
```

```
test_acc.append(total_correct * 100.0 / total_element)

print('Avg loss test ' + str(avg_loss / (i + 1)) + " epoch " + str(l))
print('Acc test ' + str(total_correct * 100.0 / total_element) \
+ " epoch " + str(l))

tb.save_value('Avg loss test',\
    'avg_loss_test', l, avg_loss / (i + 1))
tb.save_value('Avg next loss test',\
    'avg_next_loss_test', l, avg_next_loss / (i + 1))
tb.save_value('Avg mask loss test',\
    'avg_mask_loss_test', l, avg_mask_loss / (i + 1))
tb.save_value('Acc test', 'acc_test',\
    l, total_correct * 100.0 / total_element)

# torch.save(model.cuda(), "model_epoch_" + str(l))

print()
```