

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	3
1 Методы вычисления векторных представлений слов с помощью нейронной сети Transformer	6
1.1 Transformer	6
1.2 Transformer Encoder	6
1.2.1 Входная последовательности токенов	7
1.2.2 Таблица векторных представлений	8
1.2.3 Positional Encoding	9
1.2.4 Multi-head Attention	10
1.2.5 Residual connections	12
1.2.6 Layer normalization	13
1.2.7 Полносвязный слой	13
1.2.8 Encoder Layer	15
1.2.9 Encoder Transformer	16
1.3 Входной вектор	17
1.3.1 Векторные представления сегментов	18
1.4 Задачи	18
1.4.1 Masked Language Model	18
1.4.2 Next Sentence Prediction	20
1.5 Число обучаемых параметров BERT	20
1.6 Число операций в BERT	21
2 Реализация и эксперимент	23
2.1 Язык программирования и библиотеки	23
2.2 Набор данных	23
2.3 Параметры	23
2.4 Результаты	24

2.4.1	Pre-training	24
2.4.2	Задача классификации текстов	24
2.4.3	Эксперимент по изменение параметров	25
2.4.4	Эксперимент с изменением числа операций	28
3	Вычислительно эффективные методы получения векторных представлений слов с помощью нейронной сети Transformer	31
3.1	Исследование вычислительно эффективных методов получения векторных представлений слов с помощью нейронной сети Трансформер .	31
3.2	Разработка модификации архитектуры нейронной сети Трансформер .	33
3.3	Программная реализация модификации нейронной сети Трансформер	38
3.4	Результаты	39
3.4.1	Pre-training	39
3.4.2	Pre-training на одной итерации нейронной сети Transformer с модификацией	42
3.4.3	ELECTRA	46
3.4.4	ELECTRA Pre-training	49
3.4.5	ELECTRA Pre-training с модификацией	51
ЗАКЛЮЧЕНИЕ		53
СПИСОК ЛИТЕРАТУРЫ		54
ПРИЛОЖЕНИЯ		56

ВВЕДЕНИЕ

Обработка естественного языка – это важная область искусственного интеллекта, поскольку алгоритмы обработки естественного языка позволяют компьютерам обрабатывать, генерировать и понимать человеческую речь. Приложения этой области встречаются повсеместно: чат-боты, обработка электронной почты, обработка медицинских отчетов, машинный перевод и.т.д. Одним из самых эффективных инструментов обработки естественного языка является машинное обучение, причем в последние годы все больше задач решается с помощью глубокого обучения (или глубоких нейронных сетей). Глубокое обучение в свою очередь часто использует понятие векторного представления, которое выступает как средство представления единицы естественного языка в понятной для компьютера форме.

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 за-

висимости 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 способныеправляться 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 наилучшие результаты в различных задачах обработки естественного языка.

Новизна архитектуры Transformer, лежащей в основе BERT, простота идеи и реализации, а также наилучшие результаты в области 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 наилучших результатов в области, используя ограниченное количество ресурсов.

Векторные представления слов являются одними из важнейших инструментов для задач обработки естественного языка, а именно для:

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

Целью магистерской диссертации 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.

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.

1 Методы вычисления векторных представлений слов с помощью нейронной сети Transformer

1.1 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 нейронными сетями является способность извлекать информацию из более длинных входных последовательностей.

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

1.2 Transformer Encoder

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

- входная последовательность токенов;
- таблица векторных представлений;
- positional Encoding;
- механизм Multi-head Attention;
- layer-normalization;
- residual connections;
- полносвязный слой.

Рассмотрим каждый слой по отдельности.

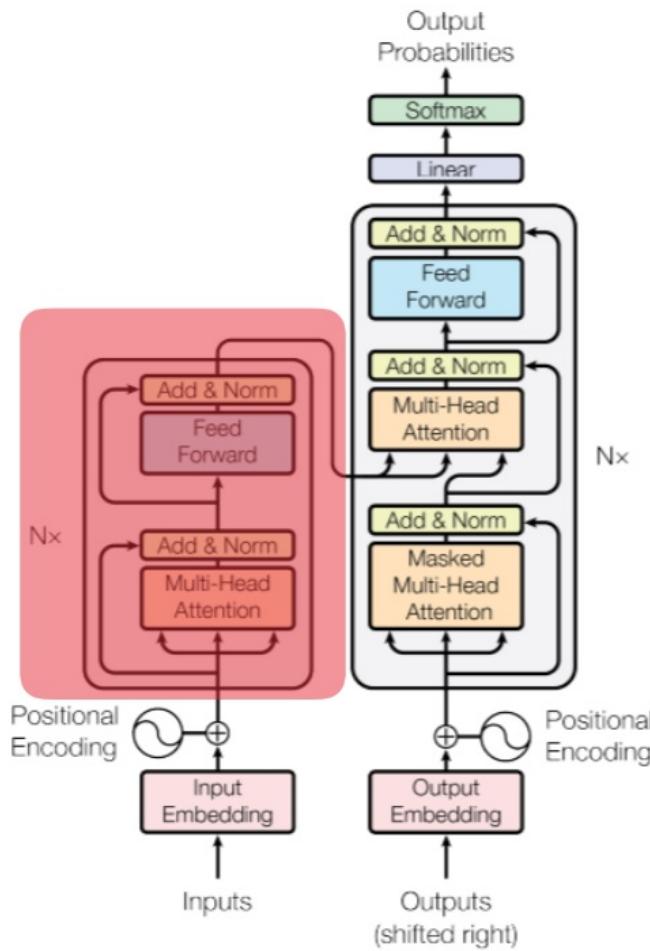


Рисунок 1 – Encoder-Decoder Transformer

1.2.1 Входная последовательности токенов

Входная последовательность представляет собой вектор (t_1, \dots, 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”.

Входная последовательность to believe

Рисунок 2 – Входная последовательность

1.2.2 Таблица векторных представлений

Таблица векторных представлений представляет собой отображение токена в n -мерный вектор. На рисунке 3 представлен пример, где исходному токену 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 реализации таблица векторных представлений инициализируется с помощью нормального распределения.

I want to believe

Векторные представления размерности 4

believe : 1	1	<table border="1"><tr><td>3</td><td>8</td><td>2</td><td>5</td></tr><tr><td>9</td><td>7</td><td>3</td><td>9</td></tr><tr><td>6</td><td>6</td><td>1</td><td>2</td></tr><tr><td>3</td><td>0</td><td>1</td><td>6</td></tr><tr><td>2</td><td>5</td><td>1</td><td>1</td></tr></table>	3	8	2	5	9	7	3	9	6	6	1	2	3	0	1	6	2	5	1	1	want	↔ 5 слов в словаре
3	8	2	5																					
9	7	3	9																					
6	6	1	2																					
3	0	1	6																					
2	5	1	1																					
			believe																					
			I																					
			coffee																					
			to																					

Lookup Table

to	2	5	1	1
believe	9	7	3	9

Рисунок 3 – Embedding lookup table

1.2.3 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:



Рисунок 4 – Positional encoding

1.2.4 Multi-head Attention

На рисунке 5 представлен основной элемент Transformer – Multi-head Attention.

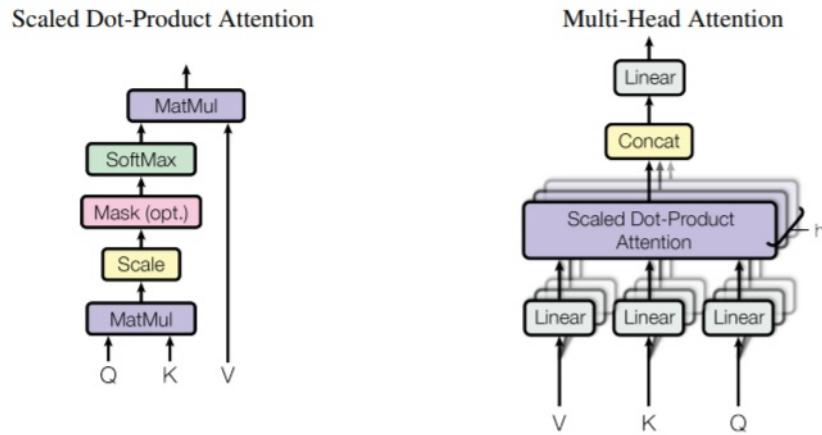


Рисунок 5 – Scaled Dot-Product Attention и Multi-Head Attention [4]

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:

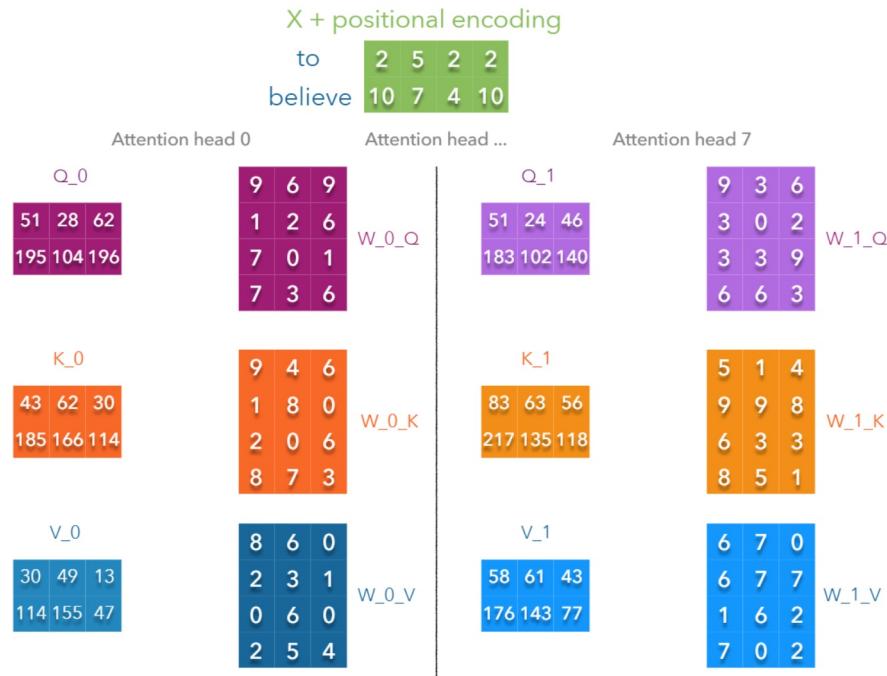


Рисунок 6 – Операция Linear в блоке Multi-Head Attention

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

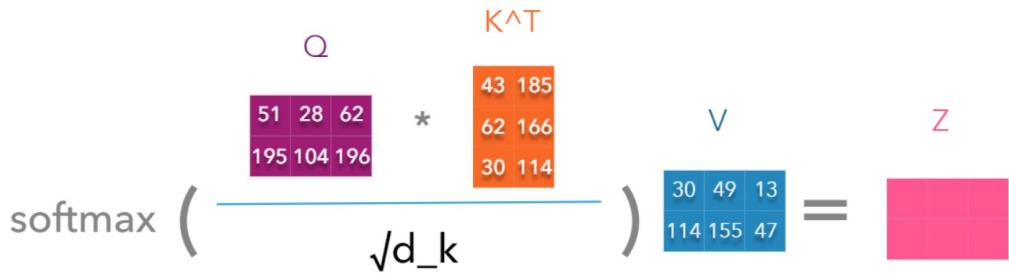


Рисунок 7 – Операция Scaled Dot-Product Attention в блоке Multi-Head Attention

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 происходит матричное умножение полученной матрицы и матрицы V .

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

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

Полученные матрицы конкатенируются:

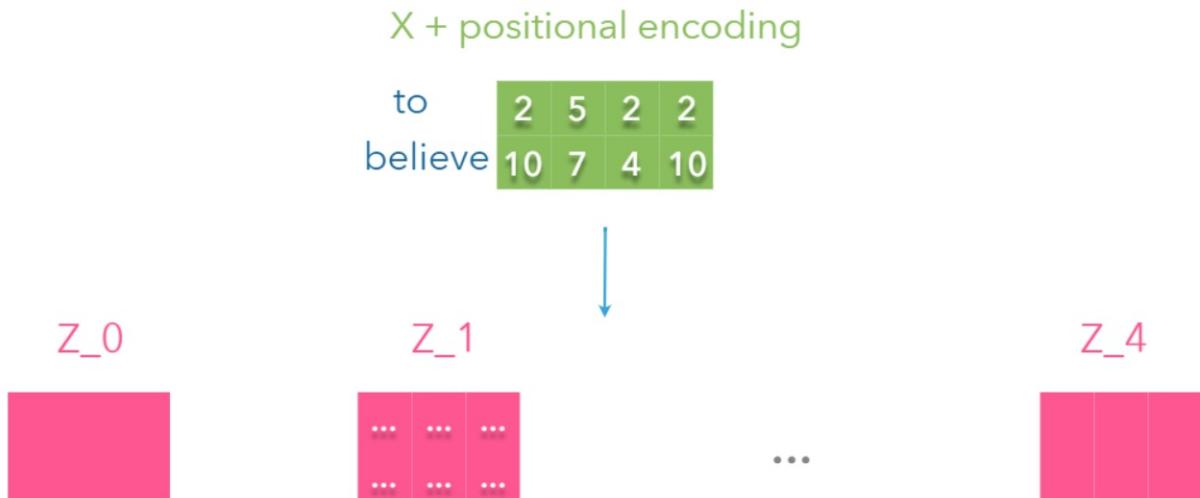


Рисунок 8 – Конкатенация матриц Z

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

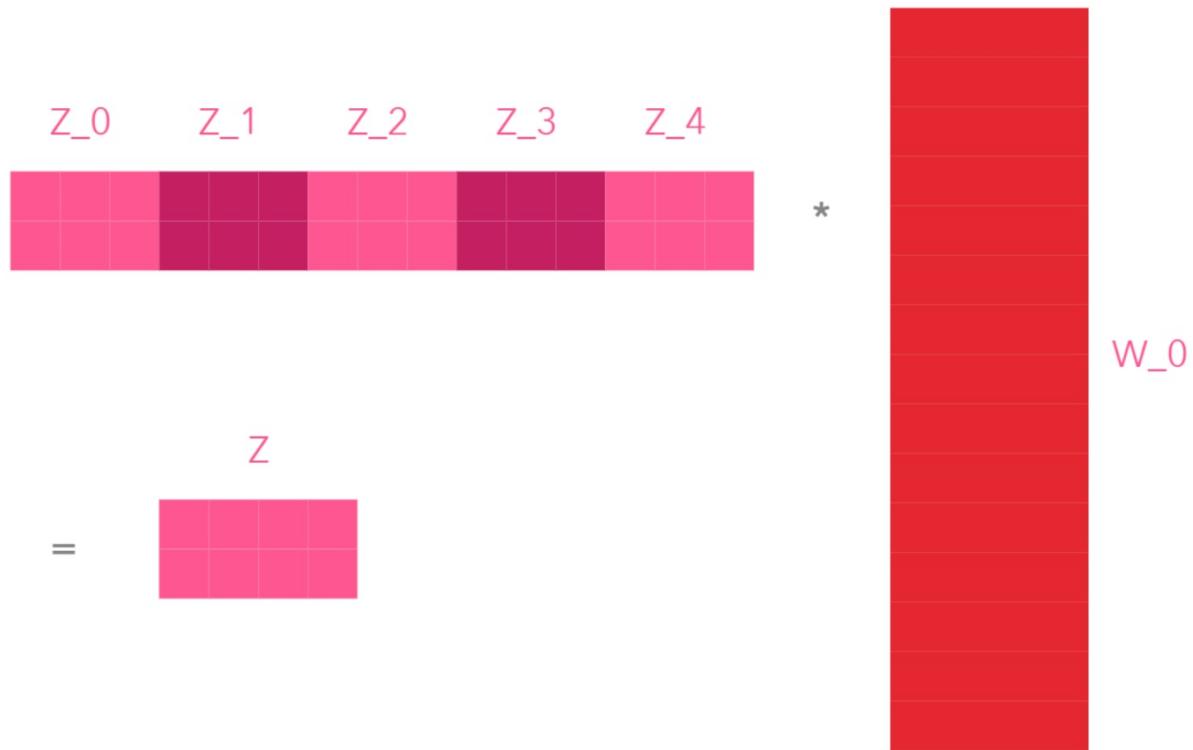


Рисунок 9 – Операция Linear в блоке Multi-Head Attention

1.2.5 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.

$$\text{LayerNorm} \left(\begin{matrix} Z \\ + X + \text{positional encoding} \end{matrix} \right)$$

The diagram illustrates the residual connection operation. It shows the sum of the output of the LayerNorm block and the input X plus positional encoding.

Рисунок 10 – Операция Add

1.2.6 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 следующие выражения:

$$\mu_\beta = \frac{1}{m} \sum_{i=1}^m x_i$$
$$\sigma_\beta^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_\beta)^2$$
$$\hat{x}_i = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}}$$
$$y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i)$$



Рисунок 11 – Операция Add and Layer Norm

1.2.7 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:

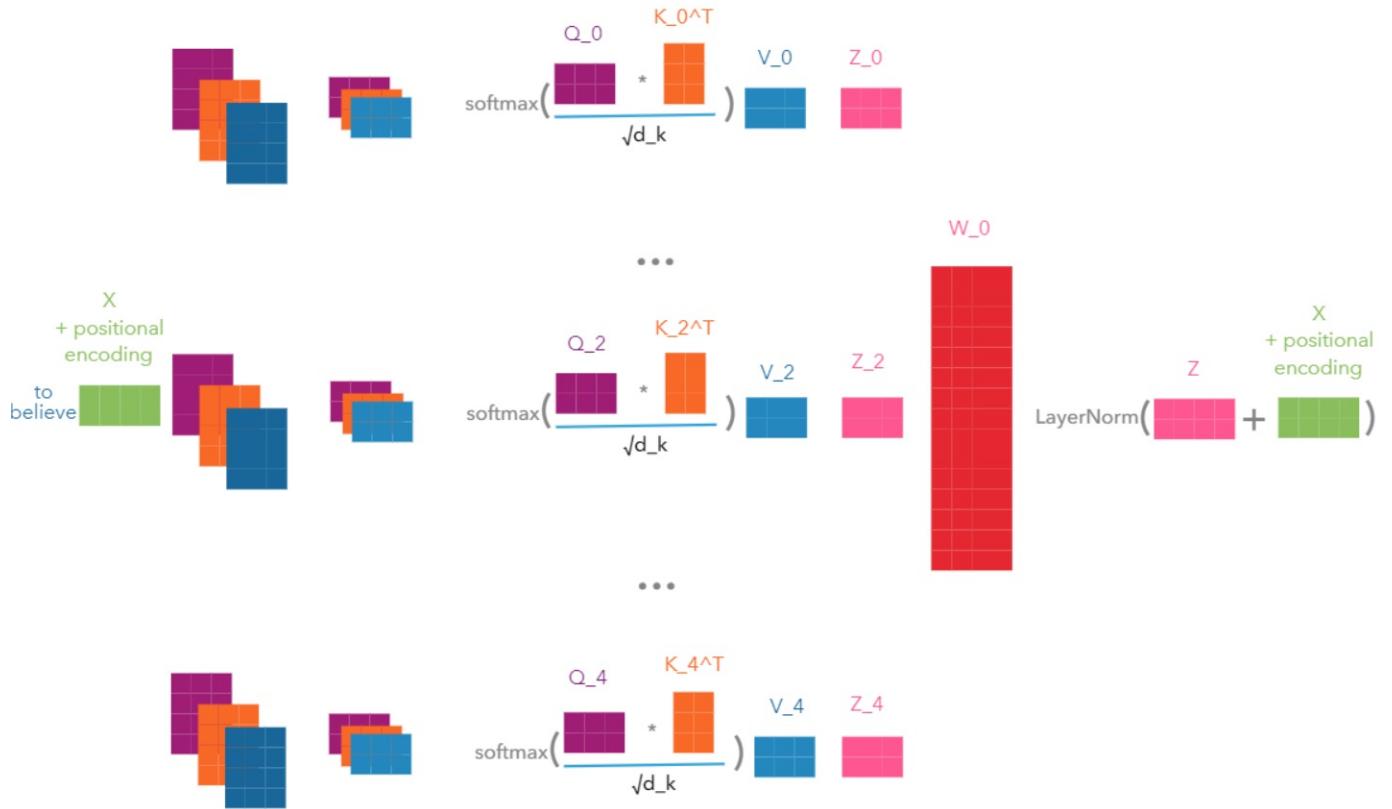


Рисунок 12 – блок Multi-Head Attention + Add and Norm

1.2.8 Encoder Layer

Объединяя все описанные операции, получаем слой Encoder:

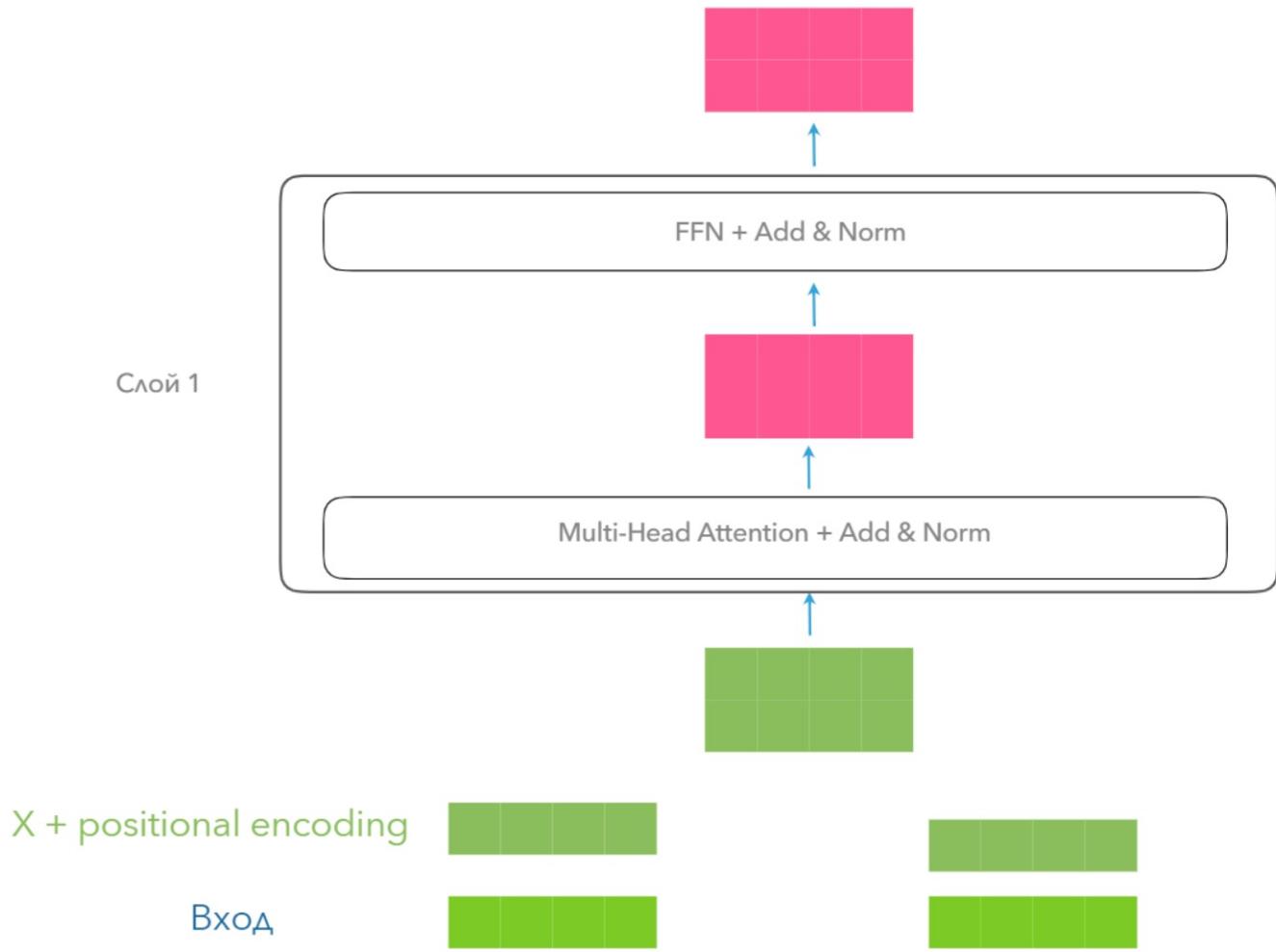


Рисунок 13 – Encoder Layer

1.2.9 Encoder Transformer

Удобно соединить два слоя, поскольку размерность матрицы не меняется:

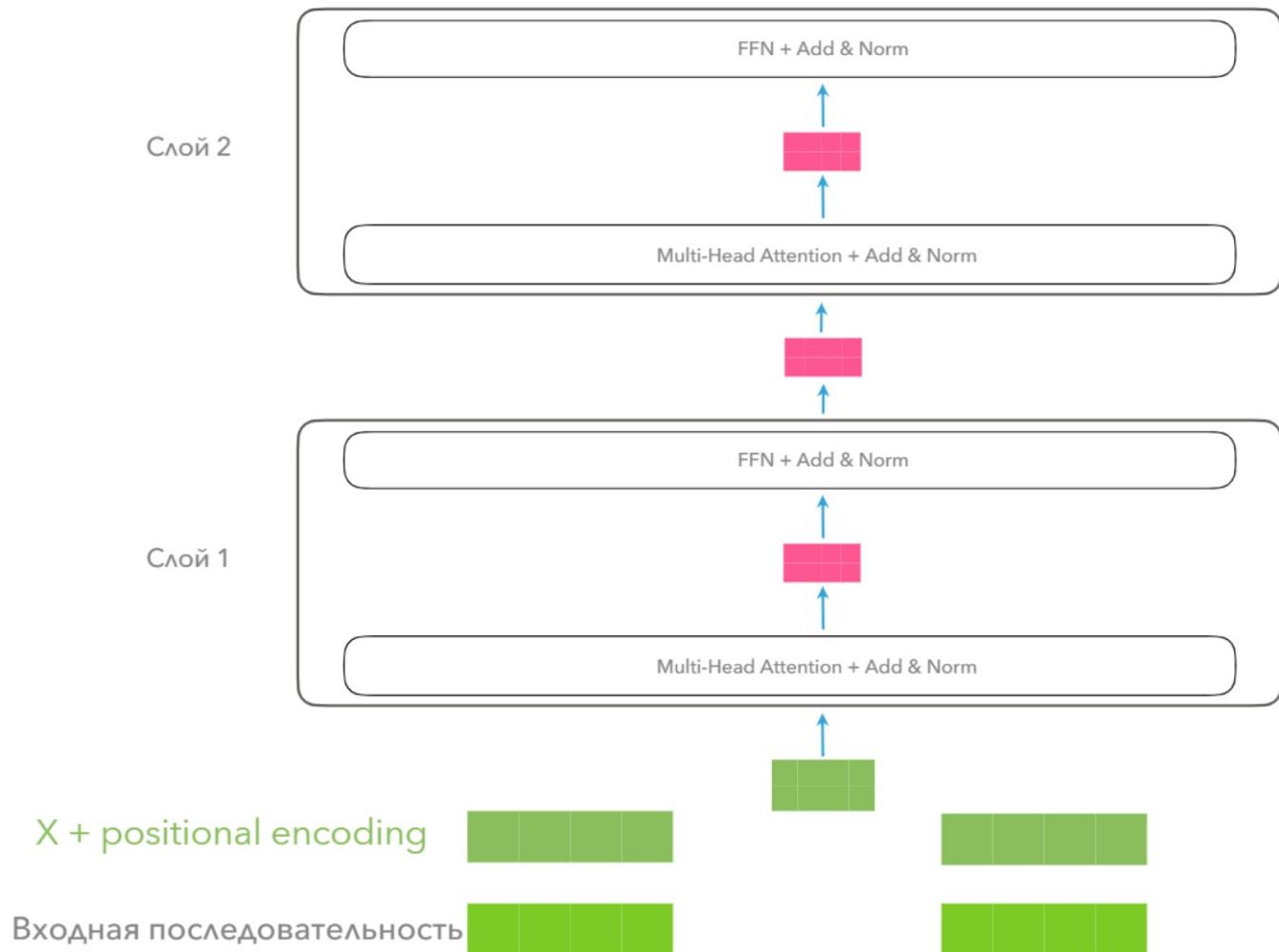


Рисунок 14 – Два слоя Encoder Layer

Объединяя несколько слоев, получаем Encoder:

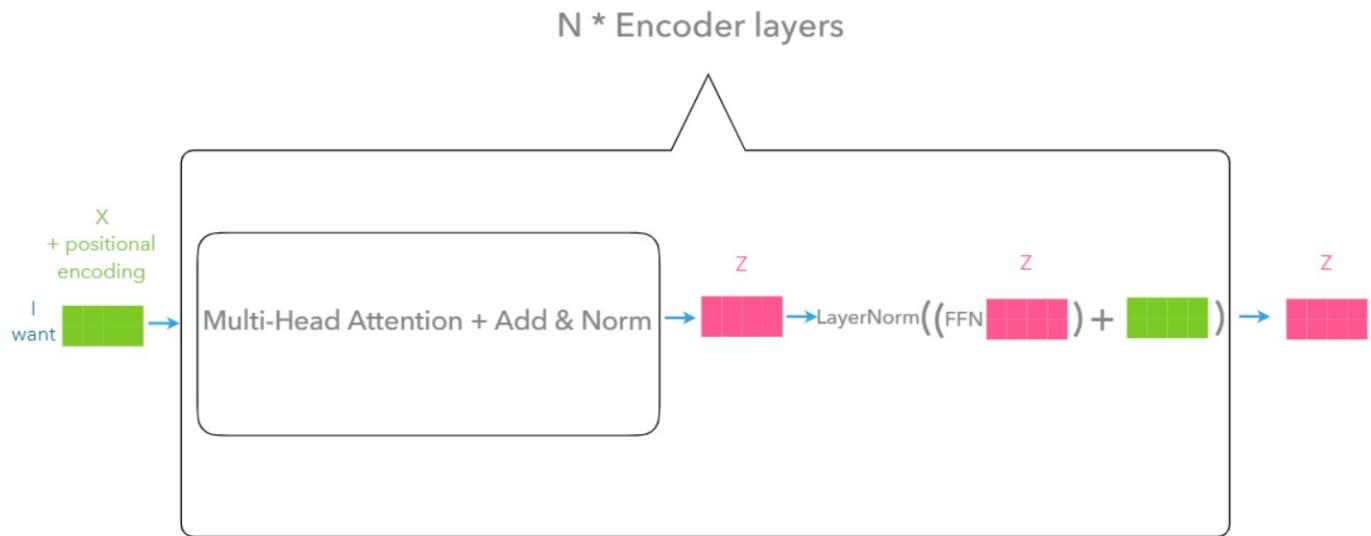


Рисунок 15 – Encoder

1.3 Входной вектор

BERT использует Encoder Transformer, описанный выше. Отличается входной вектором. На рисунке 16 представлен вход BERT.

Segment embedding	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Positional encoding	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Векторное представление	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

Токенизация [CLS] wikipedia is the best thing ever [SEP] anyone in the world can write anything [SEP]

Вход

Wikipedia is the best thing ever. Anyone in the world can write anything.

Рисунок 16 – Вход BERT

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

1.3.1 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.

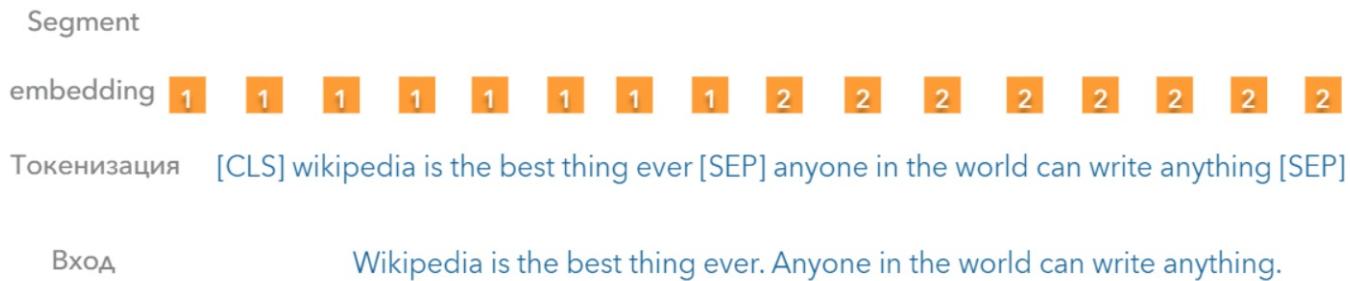


Рисунок 17 – Вход BERT – Segment Embedding

1.4 Задачи

1.4.1 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 нейронная сеть обучается предсказывать распределение вероятностей элементов словаря по контексту:

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

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].

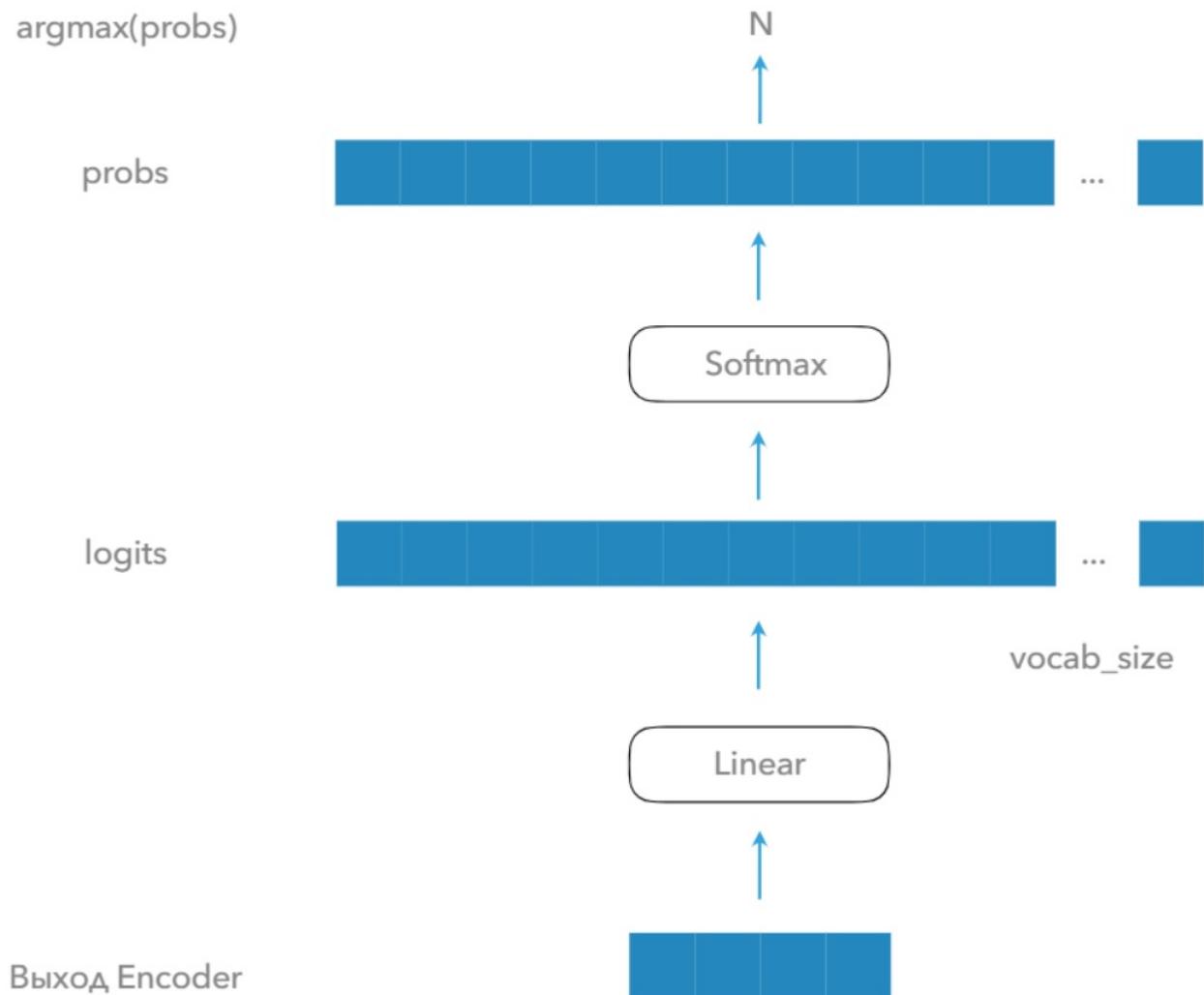


Рисунок 18 – Masked Language Model

1.4.2 Next Sentence Prediction

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

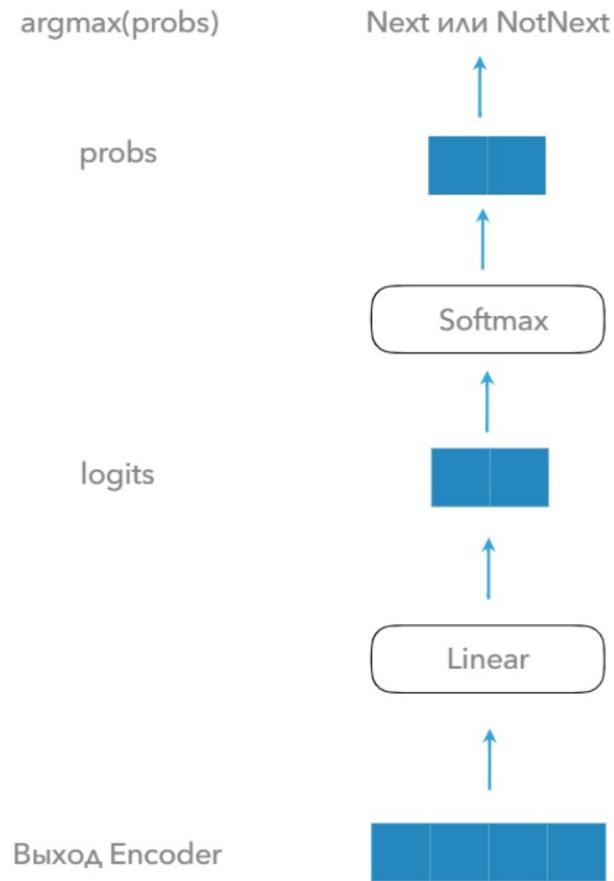


Рисунок 19 – Next Sentence Prediction

1.5 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:

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

Сокращая, получаем итоговую формулу, описывающую число обучаемых параметров в BERT:

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

1.6 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:

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

Сокращая, получаем итоговую формулу, описывающую число операций в BERT:

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

2 Реализация и эксперимент

2.1 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.

2.2 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.

2.3 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].

2.4 Результаты

2.4.1 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.

2.4.2 Задача классификации текстов

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

Таблица 1 – Точность на валидационной 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

2.4.3 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.

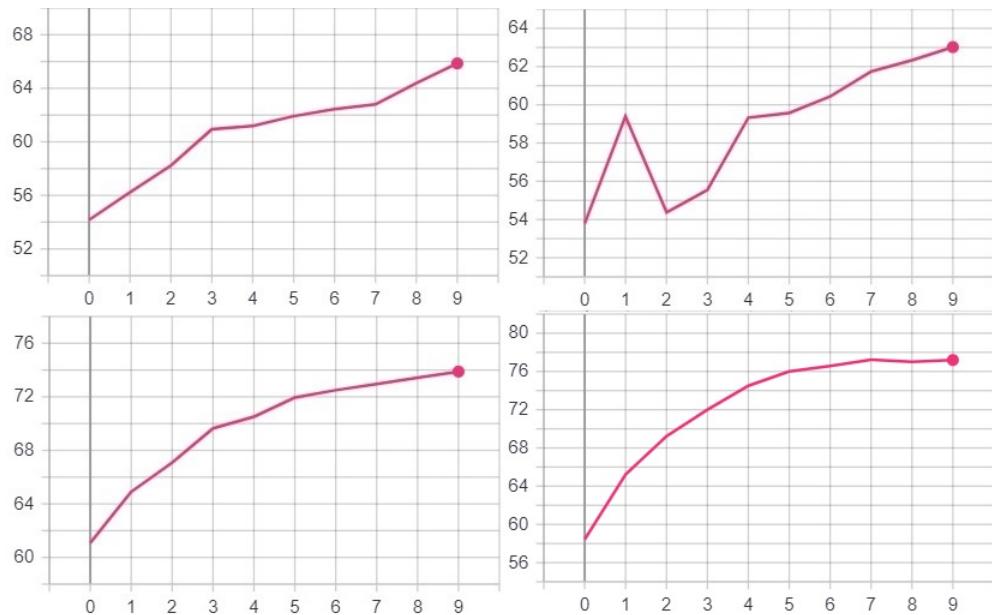


Рисунок 20 – Точность на тестовой выборке

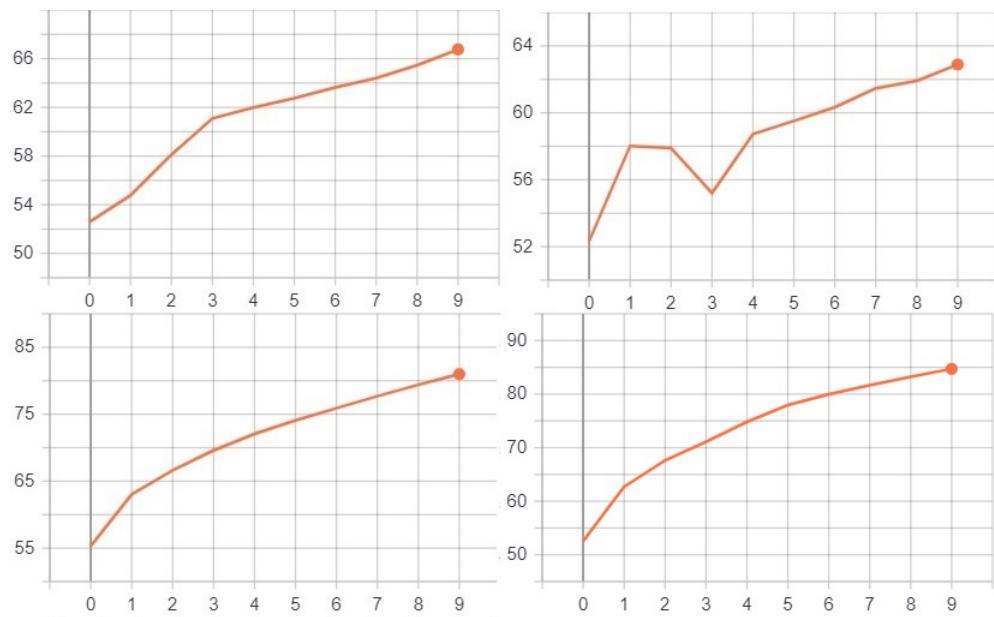


Рисунок 21 – Точность на тренировочной выборке

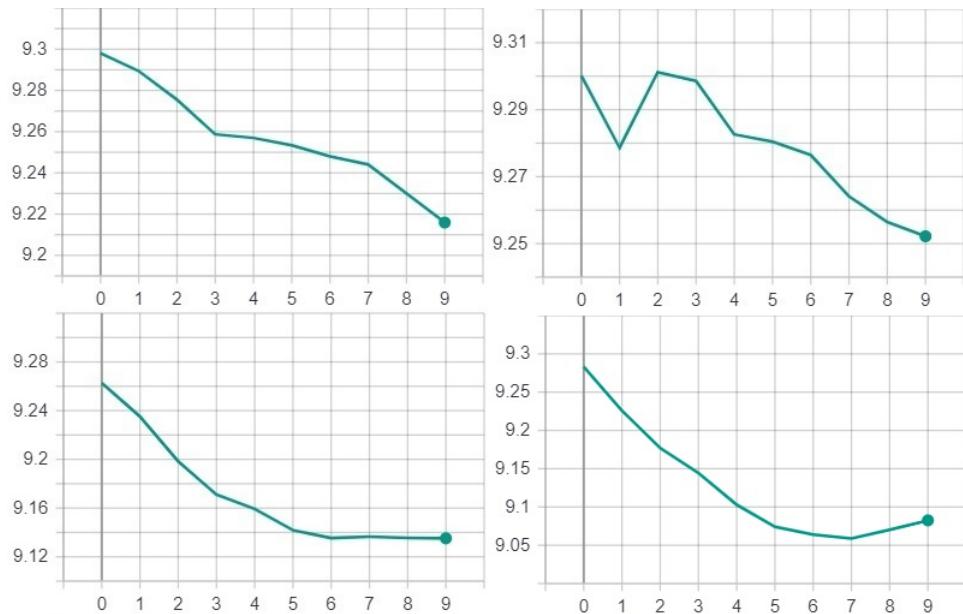


Рисунок 22 – Значение общей целевой функции на тестовой выборке

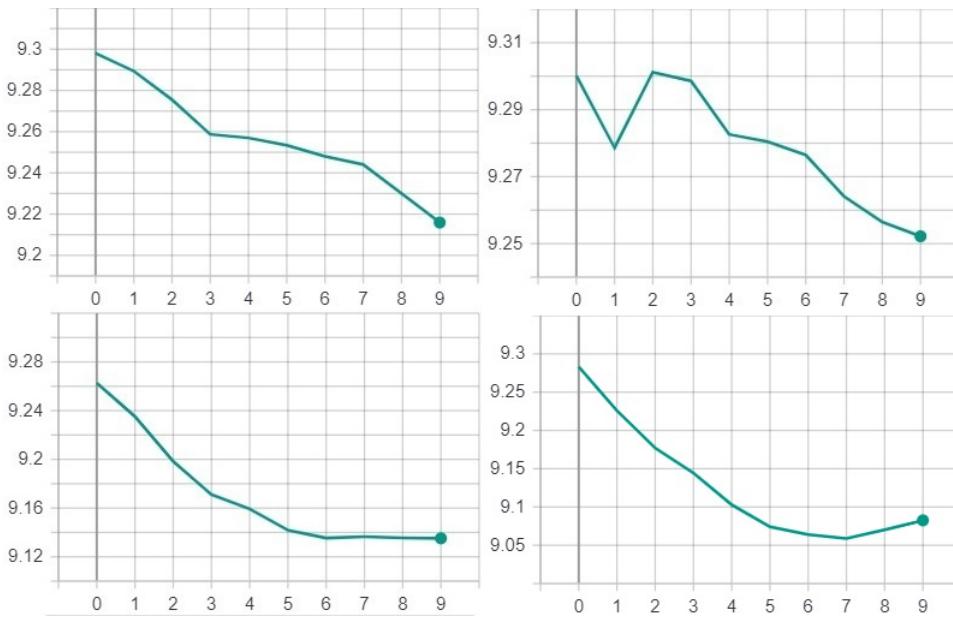


Рисунок 23 – Значение общей целевой функции на тестовой выборке

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 лучшую точность на задаче предсказания являются ли предложения последовательными и имеет ниже значения целевой функции, но при этом время обучения возрастает с увеличением числа слоёв.

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

2.4.4 Эксперимент с изменением числа операций

В ходе эксперимента с изменением числа операций в BERT использовалось аналогично 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.

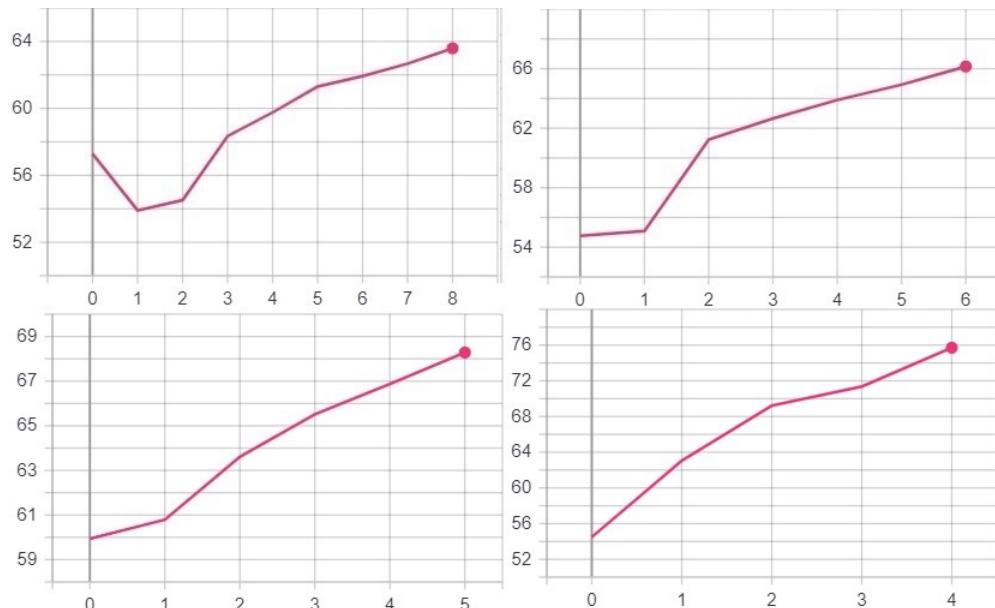


Рисунок 24 – Точность на тестовой выборке

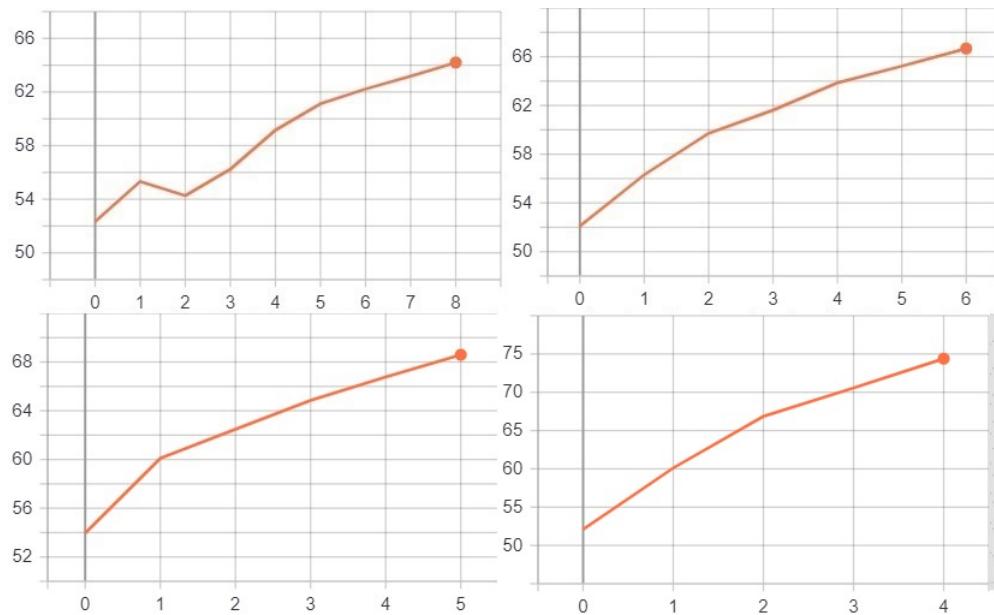


Рисунок 25 – Точность на тренировочной выборке

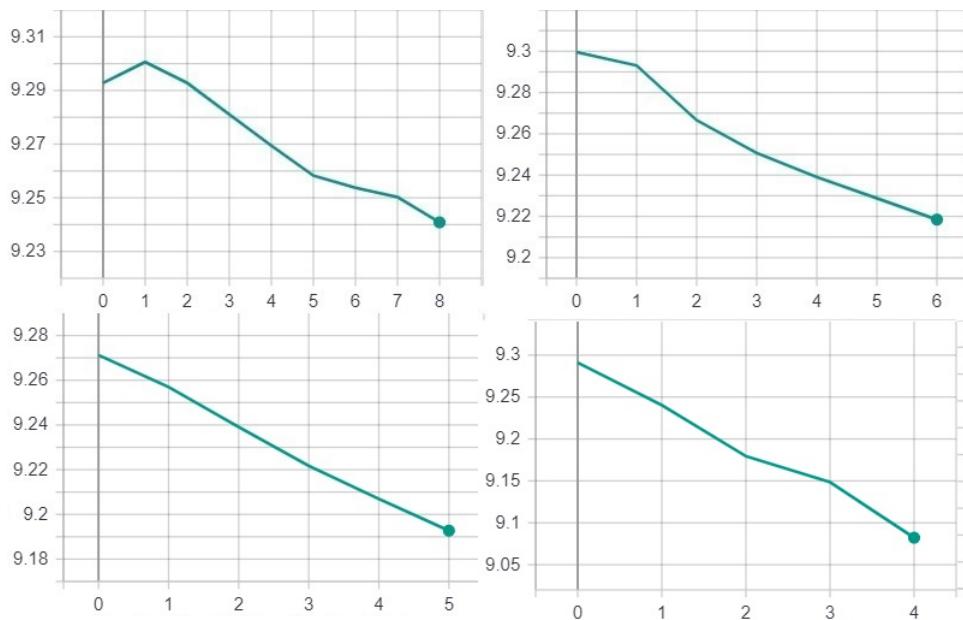


Рисунок 26 – Значение общей целевой функции на тестовой выборке

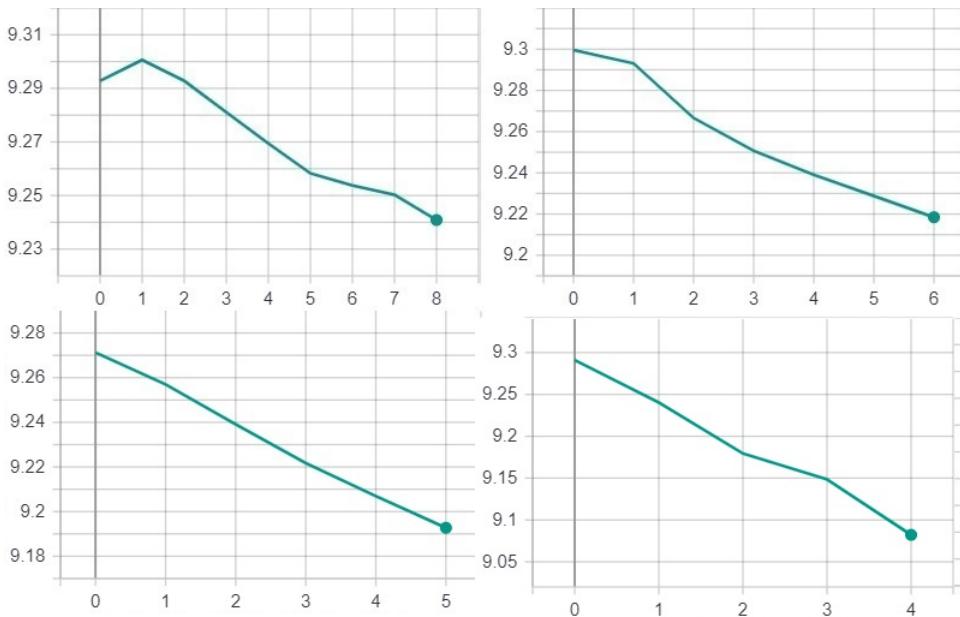


Рисунок 27 – Значение общей целевой функции на тестовой выборке

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 лучшее 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.

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

3 Вычислительно эффективные методы получения векторных представлений слов с помощью нейронной сети Transformer

3.1 Исследование вычислительно эффективных методов получения векторных представлений слов с помощью нейронной сети Трансформер

В ходе исследования методов, позволяющих вычислительно эффективно получать векторные представления слов с помощью нейронной сети Трансформер, было выделено 4 основных подхода:

1. Распределенное обучение на кластере из тензорных или графических процессоров с использованием специальных алгоритмов оптимизации:

– "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes" [7] – сокращение времени обучения нейронной сети BERT за счет использования кластера из 1024 тензорных процессоров и специального алгоритма оптимизации LAMB, позволяющего эффективно обучать нейронную сеть с использованием очень больших пакетов данных.

2. Модификации архитектуры:

– "Efficient Training of BERT by Progressively Stacking" [8] – для ускорения обучения более глубокую нейронную сеть инициализируют предобученной менее глубокой нейронной сетью, данная модификация позволяет ускорить обучение на 25% и позволяет обучать модель BERT-base за 338 часов на кластере из 4 видеокарт NVIDIA Tesla P40;

– "A Lite BERT for Self-supervised Learning of Language Representations" [9] – используются 2 модификации архитектуры нейронной сети, что позволяет ускорить обучения на 20% для BERT-base, полученная модель имеет в 18 раз меньше параметров и практически не уступает по качеству;

– "ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators" [10] – в основе модификации архитектуры лежит использование двух нейрон-

ных сетей Encoder из Transformer, одна из них выполняет роль генератора – она решает задачу Masked Language Model, описанную в главе 2.4.1, вторая нейронная сеть используется как дискриминатор – она классифицирует токен, сгенерированный генератором, как оригинальный, если он совпадает с исходным токеном, подаваемым на вход нейронной сети, либо как замещенный, если он отличается от исходного.

3. Квантизация – уменьшение числовой точности весов модели:

- ”Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT” [11] – квантизация позволяет сократить размер весов модели в 13 раз.

4. Дистилляция – техника сокращения размера модели, при которой маленькая модель обучается воспроизводить поведение большой модели: [12]

- ”DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter” [13] – маленькая модель учится на распределении выхода большой предобученной модели, метод позволяет обучить BERT за 90 часов на кластере из 8 видеокарт 16GB V100 GPU с незначительной потерей качества.

Был проанализировано также сколько вычислительных ресурсов требуется для каждой модели из статей выше. Тензорные [14] и графические [15] [16] процессоры имеют разную производительность, на основании данных о производительности различных версий тензорных и графических процессоров была получена следующая таблица (таблица 4).

Таблица 4 – Сравнение производительности тензорных и графических процессоров

GPU/TPU	Производительность	Производительность	Доля от производительности GPU V100
1 чип TPU v2	1x	0.78x	1.28
1 чип TPU v3	3x	2x	0.42
P40 GPU	1.1x	1.16x	1.16
V100 GPU	1.28x	1x	1

Итоговое сравнение всех моделей:

Таблица 5 – Сравнение времени обучения для моделей

Модель	GPU/TPU, число процессоров	Тип	Время обучения, часов	Прирост скорости, %	Число видеокарт V100	Время обучения на V100, часов
Stacking	4	P40 GPU	338	25	16	96
ALBERT	64	TPU v3	-	20	17	96
DistilBERT	8	V100 GPU	90	300	7	96
ELECTRA small	1	V100 GPU	96	4500	1	96
BERT base	16	TPU v2	96	baseline	21	96

Данные методы имеют свои недостатки. Использование кластера из процессоров и специальных алгоритмов оптимизации сокращает время обучения, но не сокращает требуемые вычислительные ресурсы. Дистилляция не позволяет обучить нейронную сеть на новом корпусе данных, поскольку для дистилляции требуется большая предобученная модель. Методы, использующие квантизацию часто сталкиваются с проблемами нестабильного обучения. Оптимальным вариантом для последующей реализации становится модификация архитектуры.

3.2 Разработка модификации архитектуры нейронной сети Трансформер

В ходе работы разработана модификация архитектуры нейронной сети. Идея модификации состоит в последовательном обучении и увеличении размера слоев нейронной сети, такой подход носит название – curriculum learning. Поскольку нейронные сети с меньшим количеством параметров обучается быстрее, то предлагается в процессе обучения постепенно увеличивать размерность слоев в ширину с использованием частично связанных слоев – модификации полносвязных слоев, в которой часть соединений между нейронами отсутствует. Данная модификация применяется ко всем полносвязным слоям в архитектуре BERT за исключением последнего слоя.

На рисунке 28 представлен процесс обучения нейронной сети – при первой итерации используются два нейрона первого слоя и один нейрон второго слоя,

остальные нейроны не участвуют в процессе обучения на данном этапе. После обучения нейронной сети в данной конфигурации добавляются два нейрона в первом слое и один нейрон во втором, причем первый нейрон во втором слое не связан с третьим и четвертым нейронами первого слоя. По аналогии происходит третья итерация процесса обучения, на этой итерации нейронная сеть достигает конечного размера.

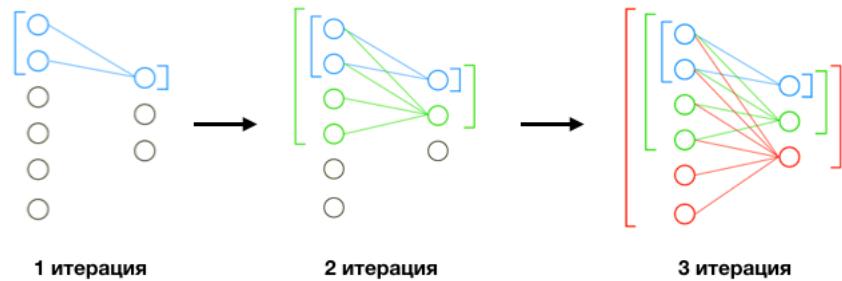


Рисунок 28 – Увеличение размерности нейронной сети

Так как наращивание размерности слоев происходит постепенно и часть нейронов не участвует в обучении, необходима модификация операции конкатенации в блоке Multi-head Attention.

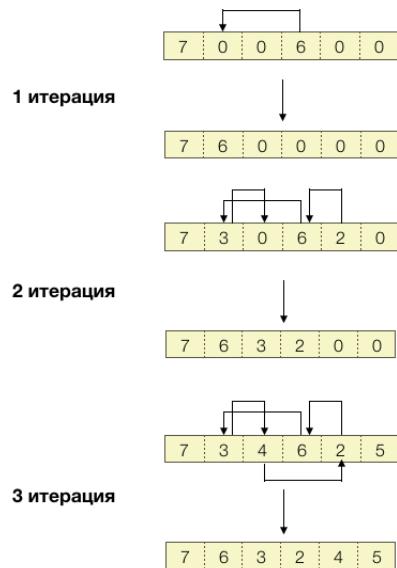


Рисунок 29 – Модификации в операции конкатенации

Модификации блока Multi-Head Attention, включающие в себя частично связанные слои и изменение порядка значений после конкатенации, представлены на рисунках 30–35 для каждой из трех итераций.

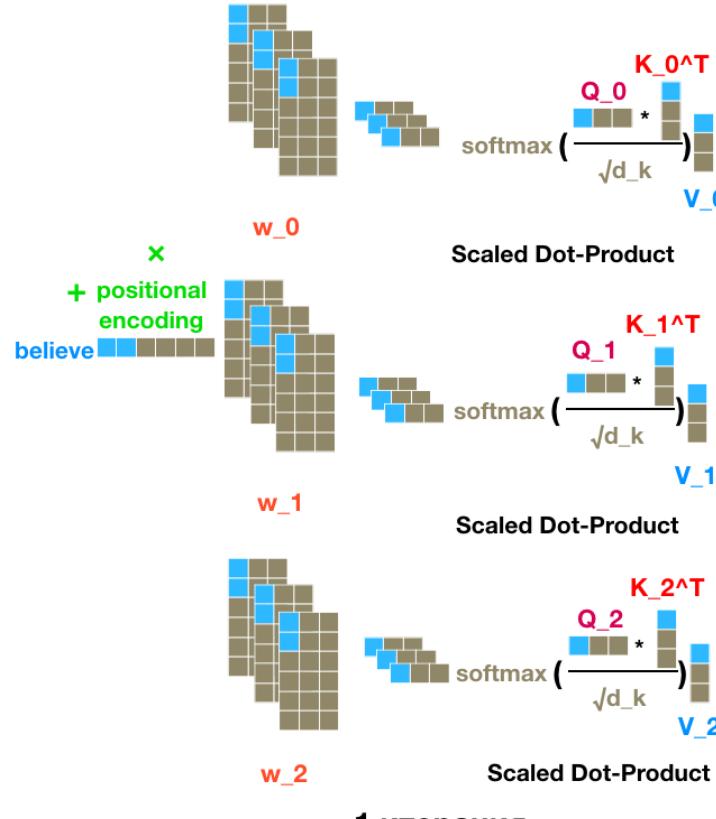


Рисунок 30 – Операция Scaled Dot-Product в блоке Multi-Head Attention на первой итерации

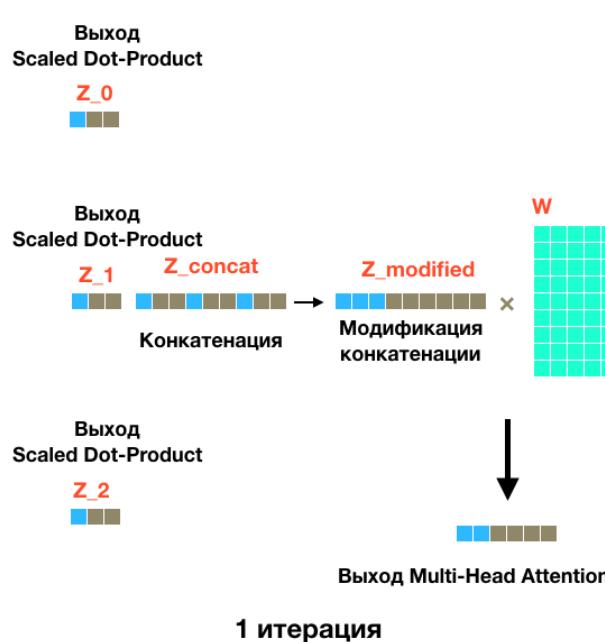


Рисунок 31 – Операция конкатенации в блоке Multi-Head Attention на первой итерации

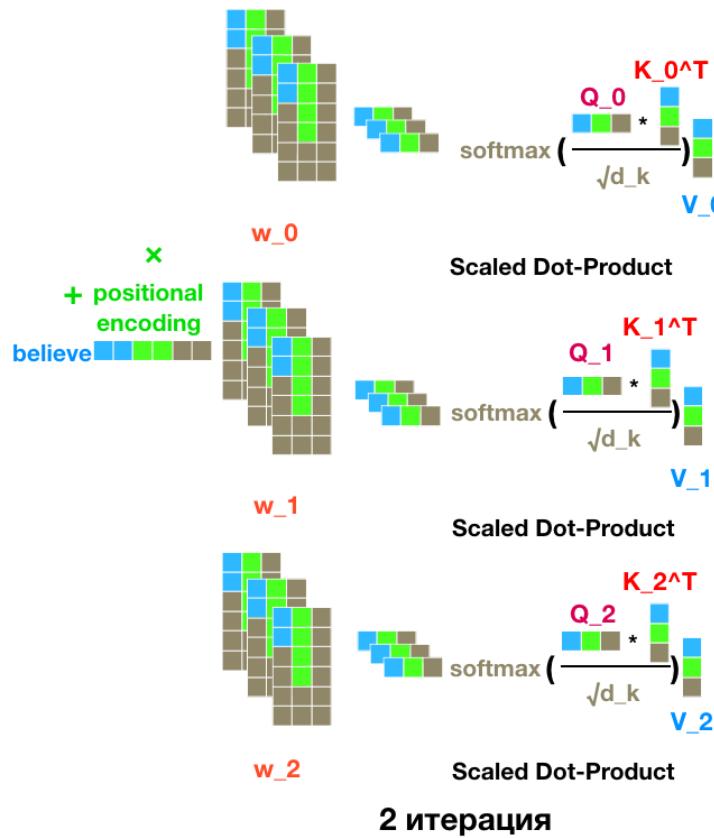


Рисунок 32 – Операция Scaled Dot-Product в блоке Multi-Head Attention на второй итерации

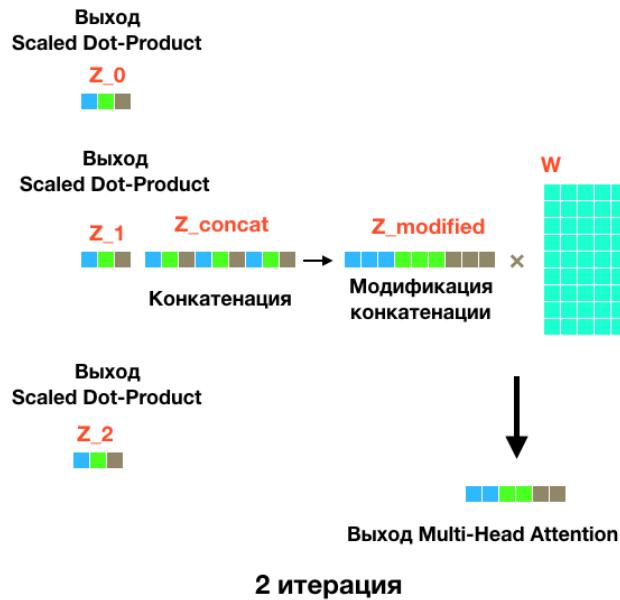


Рисунок 33 – Операция конкатенации в блоке Multi-Head Attention на второй итерации

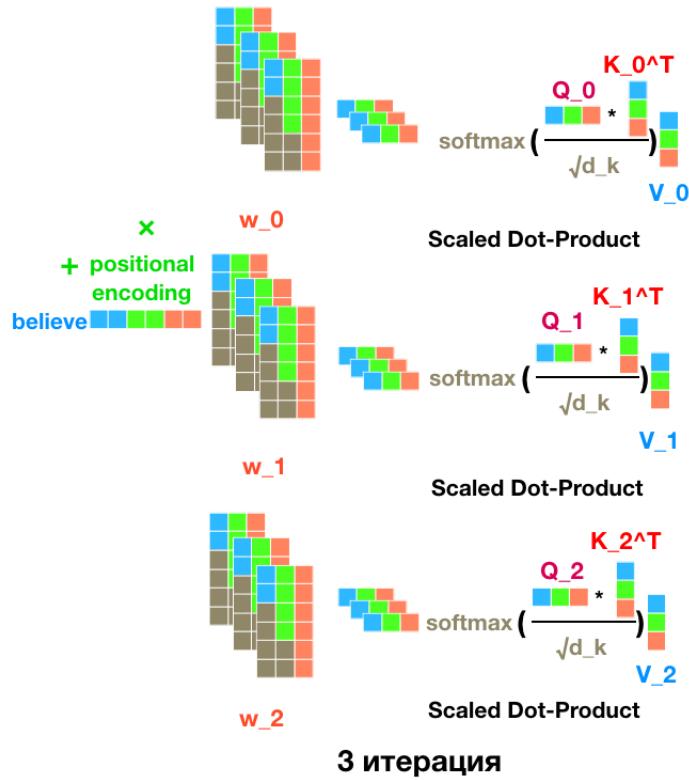


Рисунок 34 – Операция Scaled Dot-Product в блоке Multi-Head Attention на третьей итерации

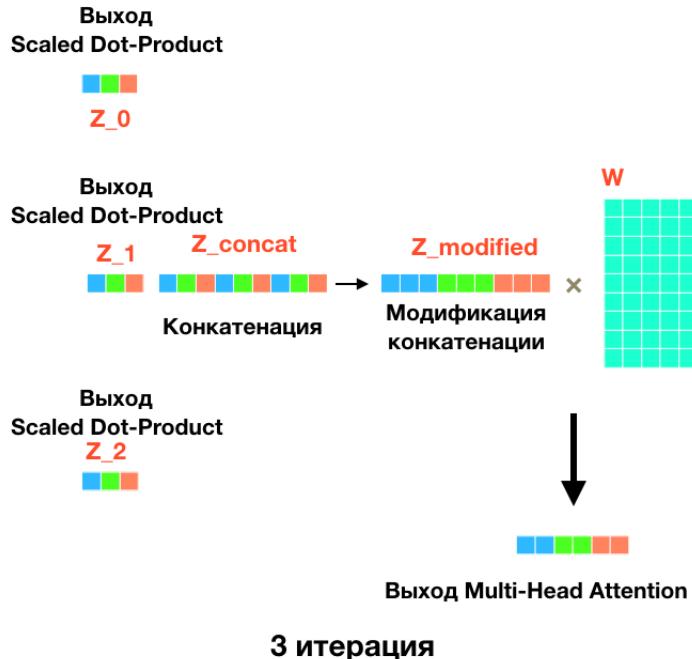


Рисунок 35 – Операция конкатенации в блоке Multi-Head Attention на третьей итерации

3.3 Программная реализация модификации нейронной сети Трансформер

В ходе программной реализации было создан класс, позволяющий создавать маски для полносвязных слоев, состоящих из нулей и единиц. Нули отключают связь между нейронами в слое.

За основу был взят стандартный класс Linear из модуля nn фреймворка для глубокого обучения Pytorch, модификация в операции конкатенации была реализована с помощью высокоеффективных на GPU операций torch.sort и torch.index_select. Маска позволяет отключать связь между нейронами как на шаге прямого распространения, так и обратного. На рисунке 36 продемонстрирована работа операции Mask.

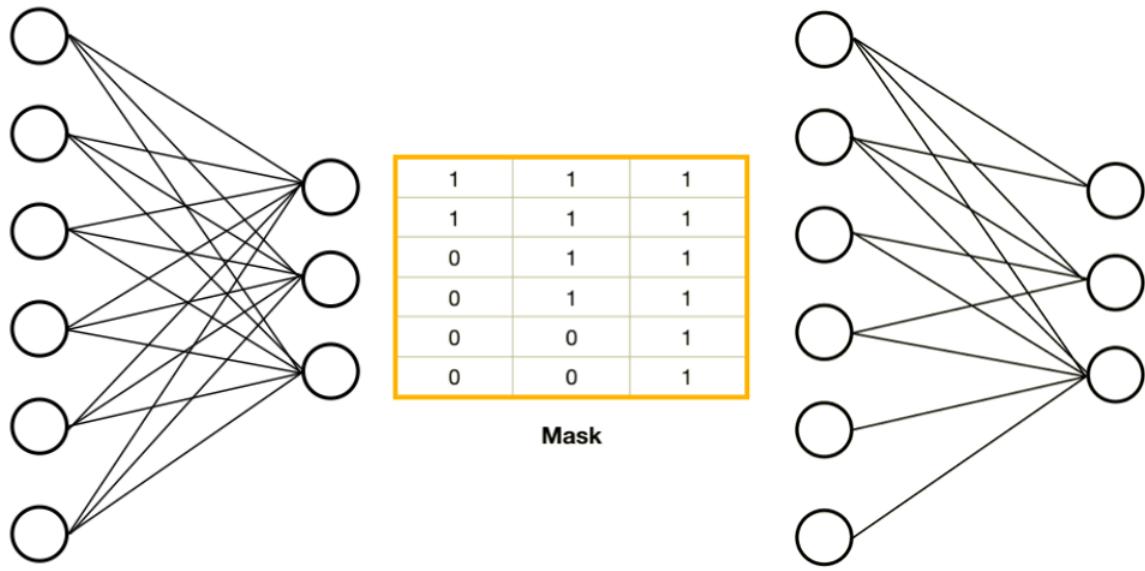


Рисунок 36 – Операция Mask

Время необходимое стандартной реализации нейронной сети Трансформер для совершения шага прямого и обратного распространения оказалось на 6% процентов меньше, чем время необходимое для совершения тех же операций предложенной модификацией. Несмотря на оптимизацию с помощью высокоеффективных на GPU операций, наиболее вычислительно затратной операцией оказалась модификация в операции конкатенации.

3.4 Результаты

3.4.1 Pre-training

Для того, чтобы оценить влияние предложенной модификации на обучение нейронной сети Transformer был проведен следующий эксперимент – оригинальная нейронная сеть Transformer и нейронная сеть с модификацией были предобучены на наборе данных WikiSplit Dataset состоящем из 500 000 пар предложений, для корректности сравнения результатов предобучения все параметры и гиперпараметры были одинаковыми для обеих нейронных сетей. Время обучения нейронных сетей также было одинаковым для обеих нейронных сетей, нейронная сеть без модификаций совершила 13 эпох обучения, нейронная сеть без модификации успела совершить 12 эпох обучения за то же время.

Результаты предобучения представлены на рисунках 37–40.

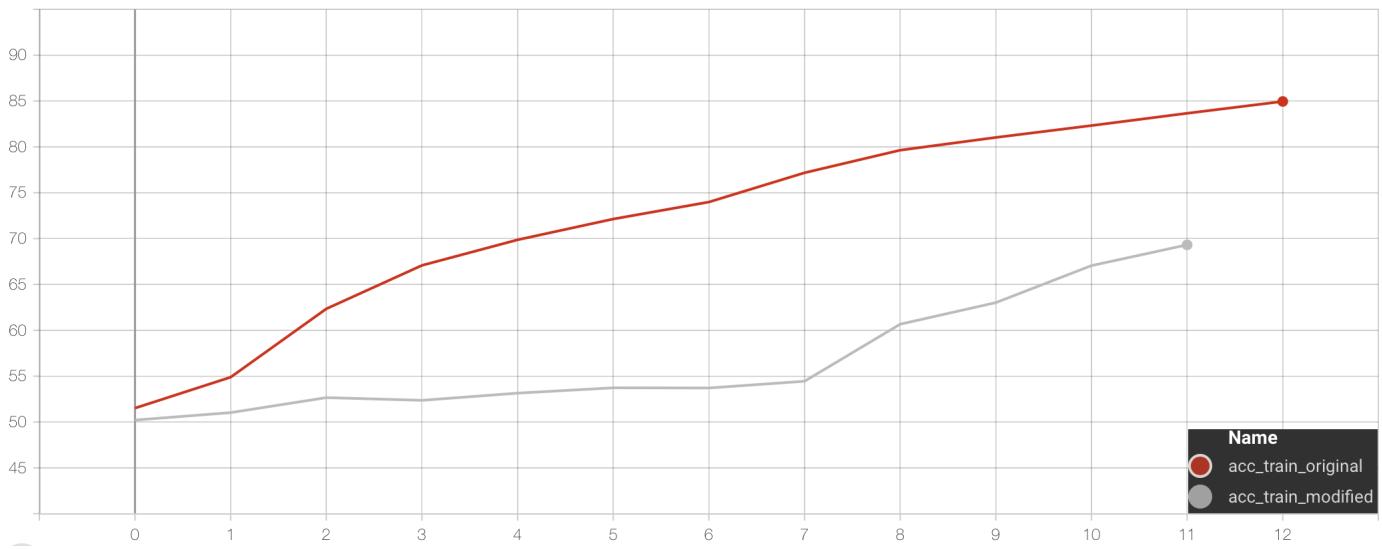


Рисунок 37 – Точность классификации для задачи Next Sentence Prediction на тренировочной выборке исходной модели – acc_train_original, модели с модификацией – acc_train_modified

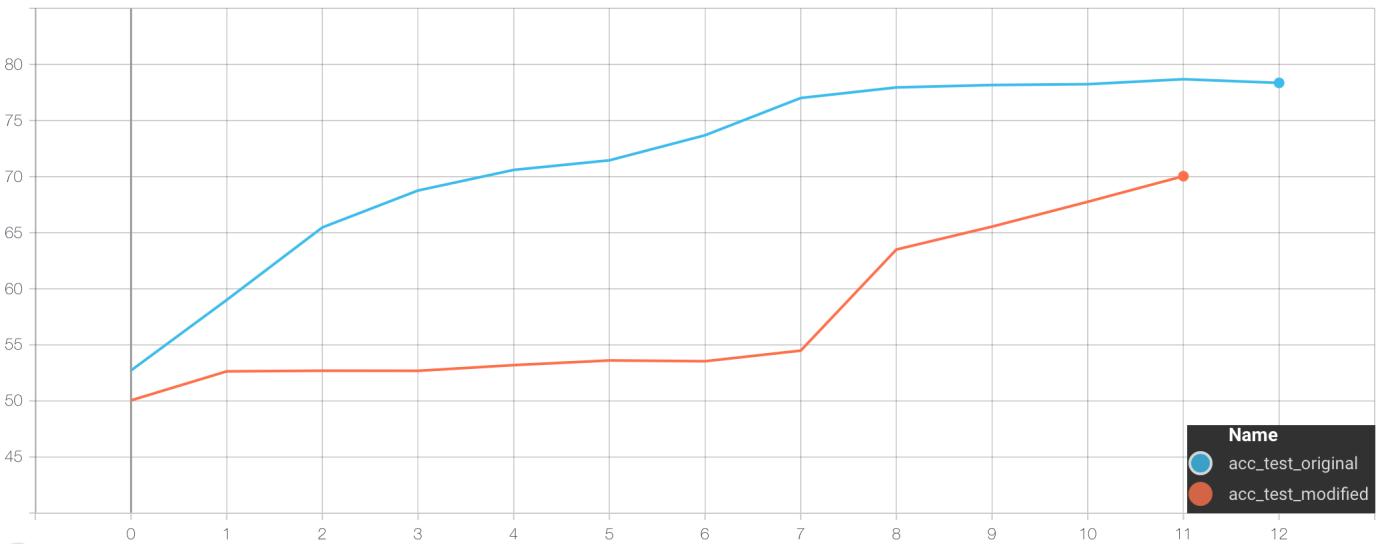


Рисунок 38 – Точность классификации для задачи Next Sentence Prediction на тестовой выборке исходной модели – acc_test_original, модели с модификацией – acc_test_modified

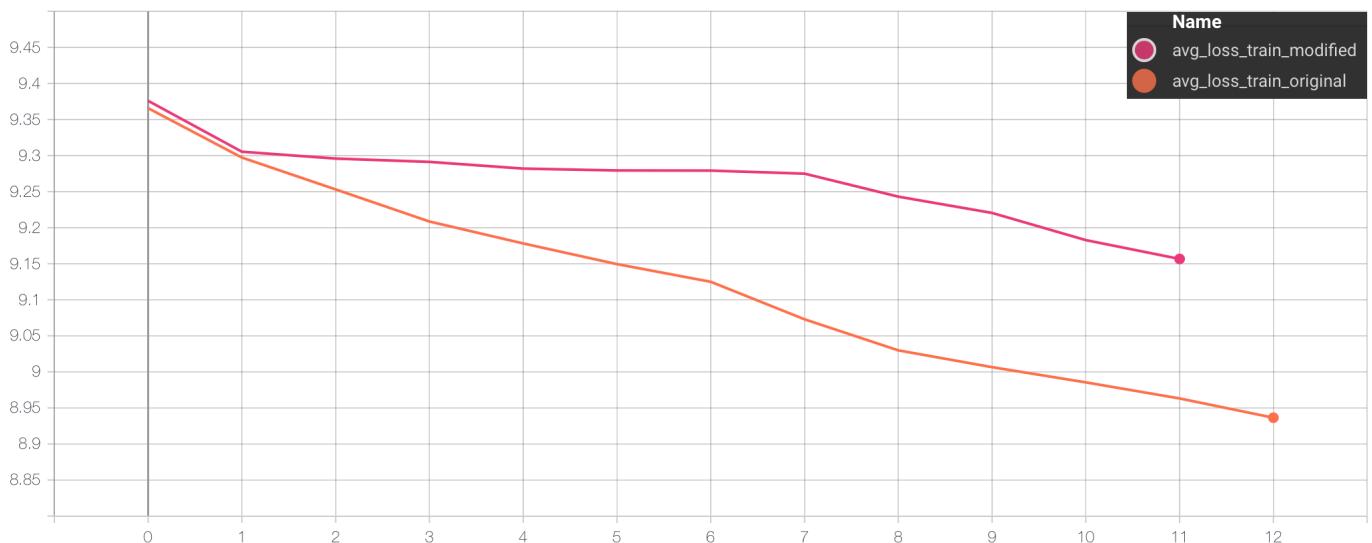


Рисунок 39 – Значение общей целевой функции на тренировочной выборке исходной модели – avg_loss_train_original, модели с модификацией – avg_loss_train_modified

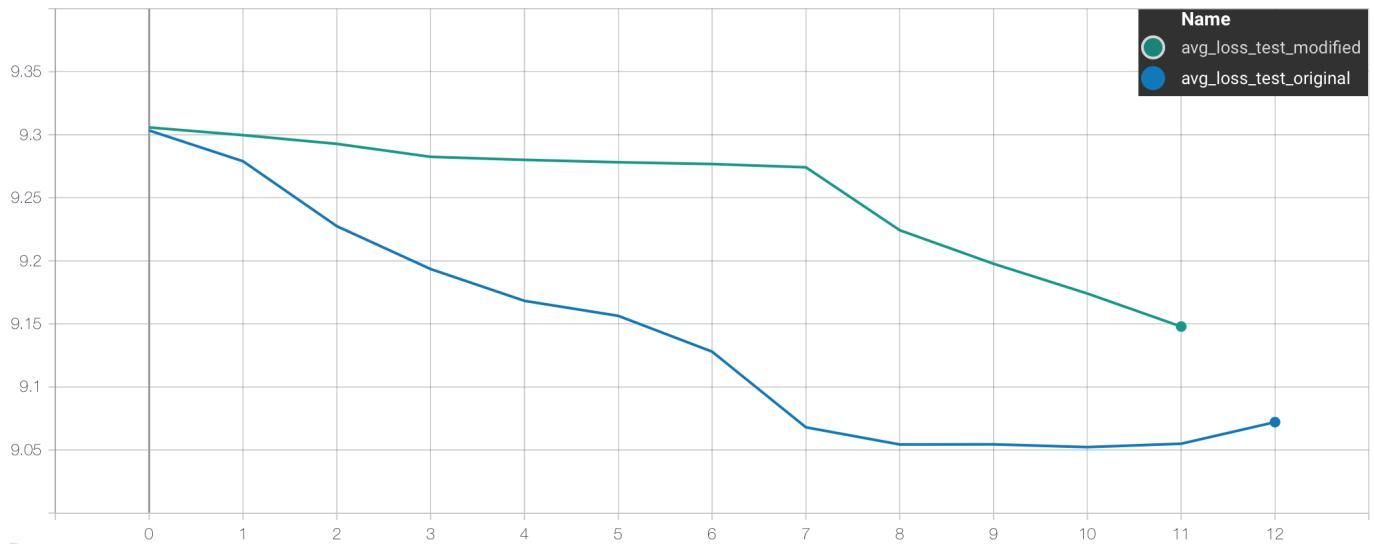


Рисунок 40 – Значение общей целевой функции на тестовой выборке исходной модели – avg_loss_test_original, модели с модификацией – avg_loss_test_modified

Таблица 6 – Результаты предобучения

Значение	Нейронная сеть без модификации	Нейронная сеть с модификацией	Относительная разница
Точность классификации для задачи Next Sentence Prediction, тренировочная выборка	84.95 %	69.32 %	-18 %
Точность классификации для задачи Next Sentence Prediction, тестовая выборка	78.35 %	70.04 %	-11 %
Общее значение целевой функции, тренировочная выборка	8.938	9.158	-3 %
Общее значение целевой функции, тестовая выборка	9.071	9.149	-1 %

Нейронная сеть с модификацией показала результат хуже, чем оригинальная нейронная сеть Transformer. Но как видно из графика значений целевой функции на рисунке 39, разница между оригинальной нейронной сетью и сетью с модификацией незначительна, в этом может быть потенциал для ускорения обучения нейронной сети.

3.4.2 Pre-training на одной итерации нейронной сети Transformer с модификацией

В ходе проведения эксперимента, описанного в предыдущей главе, возникла гипотеза о том, что для ускорения обучения оригинальной нейронной сети Transformer может быть использовано в качестве инициализации начальное предобучение с помощью первой итерации предобучения нейронной сети Transformer с модификацией.

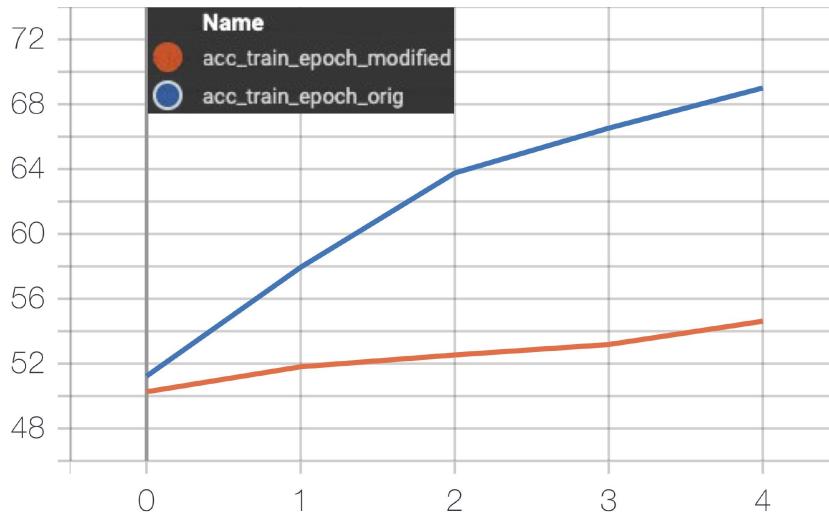


Рисунок 41 – Точность классификации для задачи Next Sentence Prediction на тренировочной выборке исходной модели – acc_train_orig, модели с модификацией – acc_train_modified

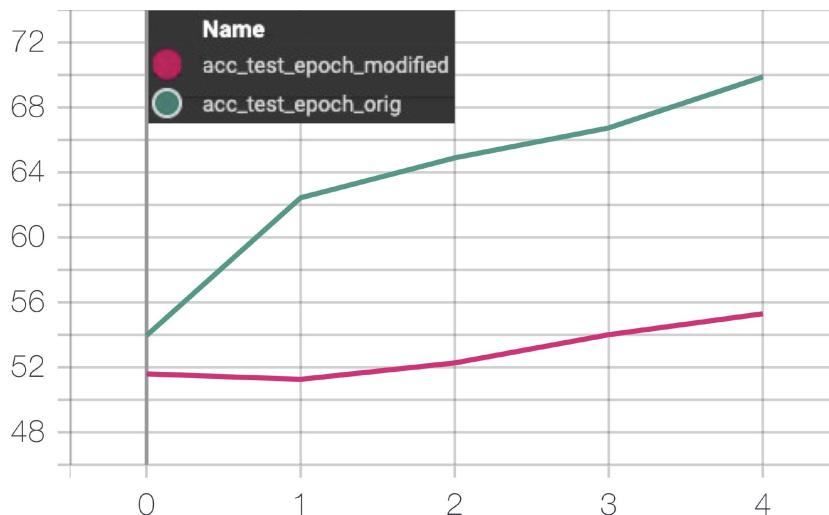


Рисунок 42 – Точность классификации для задачи Next Sentence Prediction на тестовой выборке исходной модели – acc_test_orig, модели с модификацией – acc_test_modified

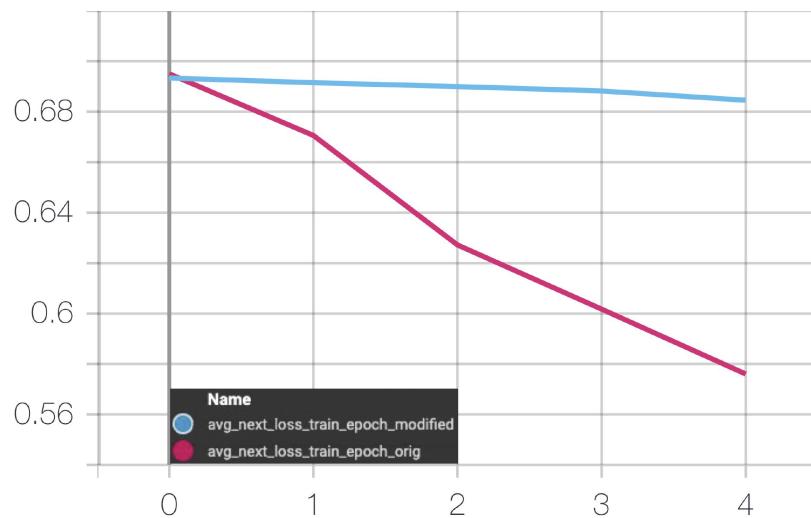


Рисунок 43 – Значение целевой функции для задачи Next Sentence Prediction на тренировочной выборке исходной модели – avg_next_loss_test_orig, модели с модификацией – avg_next_loss_test_modified

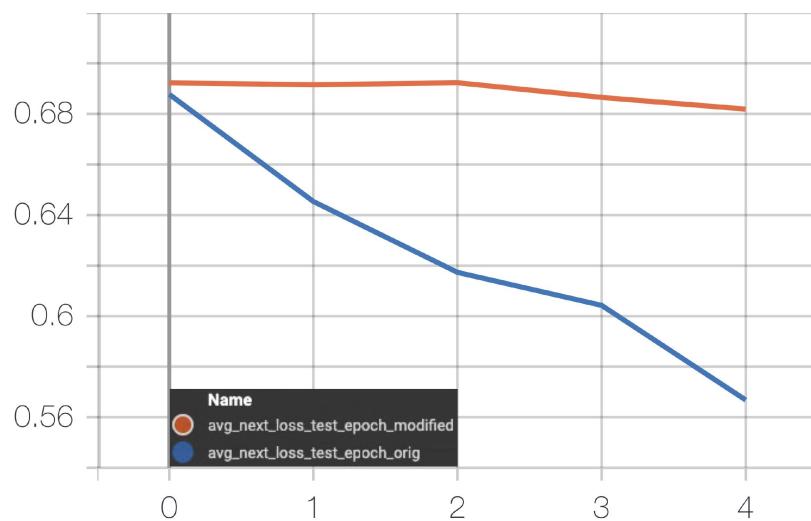


Рисунок 44 – Значение целевой функции для задачи Next Sentence Prediction на тестовой выборке исходной модели – avg_next_loss_test_orig, модели с модификацией – avg_next_loss_test_modified

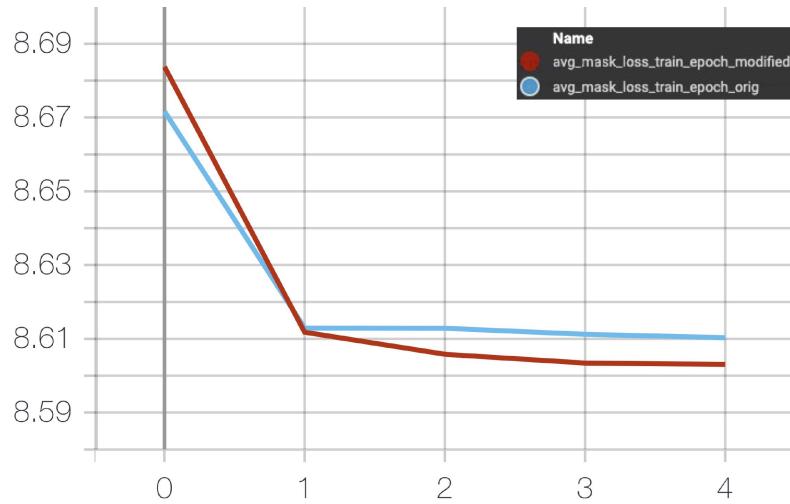


Рисунок 45 – Значение целевой функции для задачи Masked Language Model на тренировочной выборке исходной модели – avg_mask_loss_train_orig, модели с модификацией – avg_mask_loss_train_modified

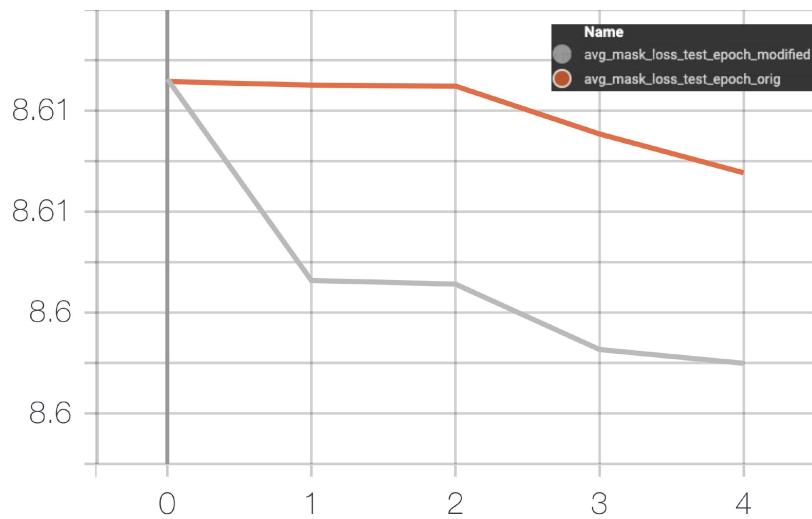


Рисунок 46 – Значение целевой функции для задачи Masked Language Model на тестовой выборке исходной модели – avg_mask_loss_test_orig, модели с модификацией – avg_mask_loss_test_modified

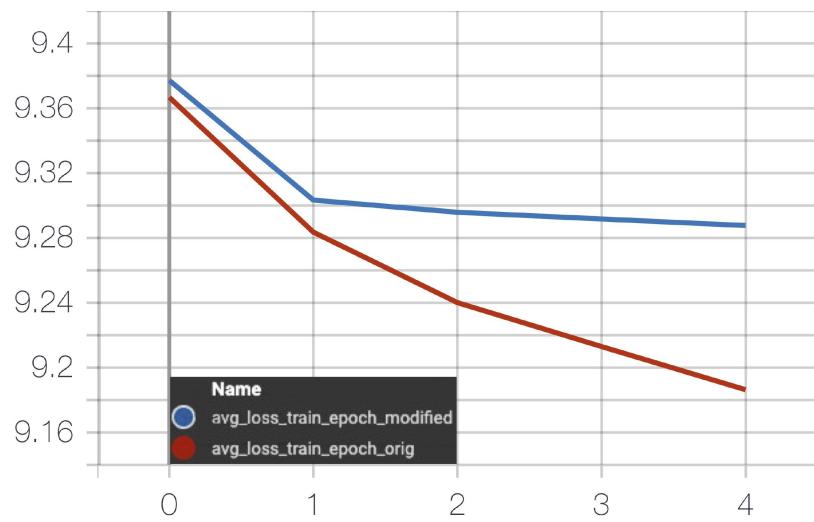


Рисунок 47 – Значение общей целевой функции на тренировочной выборке исходной модели – avg_loss_test_orig, модели с модификацией – avg_loss_test_modified

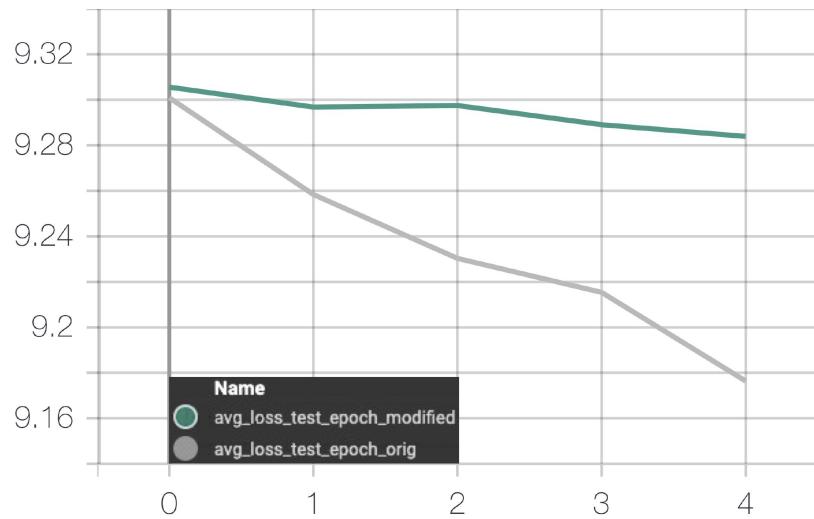


Рисунок 48 – Значение общей целевой функции на тестовой выборке исходной модели – avg_loss_test_orig, модели с модификацией – avg_loss_test_modified

Таблица 7 – Результаты предобучения

Значение	Нейронная сеть без модификации	Нейронная сеть с модификацией	Относительная разница
Точность классификации для задачи Next Sentence Prediction, тренировочная выборка	69 %	54.62 %	-21 %
Точность классификации для задачи Next Sentence Prediction, тестовая выборка	69.88%	55.3 %	-21 %
Общее значение целевой функции, тренировочная выборка	9.186	9.288	-1 %
Общее значение целевой функции, тестовая выборка	9.176	9.284	-1 %
Значение целевой функции для задачи Masked Language Model, тренировочная выборка	8.61	8.603	+1 %
Значение целевой функции для задачи Masked Language Model, тестовая выборка	8.61	8.602	+1 %
Значение целевой функции для задачи Next Sentence Prediction, тренировочная выборка	0.576	0.6846	-20 %
Значение целевой функции для задачи Next Sentence Prediction, тестовая выборка	0.5668	0.6819	-19%

По результатам эксперимента нейронная сеть с модификацией показала результат незначительно лучше на задаче Masked Language Model, но общий результат предобучения оказался хуже значительно.

3.4.3 ELECTRA

Как было сказано ранее, нейронная сеть Transformer требует значительных вычислительных ресурсов. Но в марте 2020 года был представлен новый подход ELECTRA [10], позволяющий достигать близких к наилучшим в области результатов, используя небольшое количество вычислительных ресурсов.

Новизна подхода заключается с использовании в архитектуре двух нейронных сетей, первая нейронная сеть выступает в роли генератора, вторая – в роли дискриминатора. И генератор, и дискриминатор представляют собой Encoder Transformer. Похожая структура используется в генеративно-состязательных сетях [17], но в ELECTRA не используется состязательное обучение.

Генератор в ELECTRA принимает на вход токены, среди которых часть замаскирована токеном MASK, на выходе – та же последовательность токенов, только вместо токенов MASK генератор подставляет такие токены, которые по его мнению должны присутствовать на месте замаскированных токенов. Далее эта сгенерированная последовательность токенов подается на вход дискриминатору, который для каждого токена предсказывает – был ли он сгенерирован дискриминатором, либо это токен, который присутствовал в оригинальной последовательности. Таким образом устраняется недостаток нейронной сети BERT, поскольку для обучения нейронной сети используются все токены, а не 15 % токенов, как это было в оригинальной архитектуре.

Общая целевая функция для обучения нейронной сети выглядит следующим образом:

$$\min_{\theta_G, \theta_D} \sum_{\mathbf{x} \in X} L_{MLM}(\mathbf{x}, \theta_G) + \lambda L_{Disc}(\mathbf{x}, \theta_D),$$

где X – это весь корпус текста, а $\mathbf{x} = [x_1, x_2, \dots, x_n]$ – входная последовательность токенов.

$L_{MLM}(\mathbf{x}, \theta_G)$ – целевая функция, отвечающая за обучение генератора:

$$L_{MLM}(\mathbf{x}, \theta_G) = \mathbb{E} \left(\sum_{i \in \mathbf{m}} -\log p_G(x_i | \mathbf{x}^{masked}) \right),$$

где $\mathbf{m} = [m_1, m_2, \dots, m_n]$ – случайно выбранное множество целых чисел от 1 до n используемых в качестве индексов для маскировки входных токенов токеном [MASK], а \mathbf{x}^{masked} – токены, замаскированные с помощью данных индексов.

$L_{Disc}(\mathbf{x}, \theta_D)$ – целевая функция, отвечающая за обучение дискриминатора:

$$L_{Disc}(\mathbf{x}, \theta_D) = \mathbb{E} \left(\sum_{t=1}^n -\mathbb{1}(x_t^{crt} = x_t) \log D(\mathbf{x}^{crt}, t) - \mathbb{1}(x_t^{crt} \neq x_t) \log (1 - D(\mathbf{x}^{crt}, t)) \right),$$

где \mathbf{x}^{crt} – токены, сгенерированные генератором вместо токенов [MASK].

После окончания обучения нейронной сети оставляют только дискриминатор и дообучают на наборе данных для конкретной задачи.

3.4.4 ELECTRA Pre-training

Для сравнения эффективности оригинальной архитектуры и ELECTRA был проведен эксперимент, состоящий из предобучения нейронных сетей на наборе данных WikiSplit Dataset и сравнения достигнутых значений целевых функций. Поскольку целевая функция оригинальной нейронной сети BERT и целевая функция генератора в ELECTRA эквивалентны, можно провести корректное сравнение эффективности.

В своей работе авторы утверждают, что наиболее эффективным образом работает архитектура, в которой размер генератора составляет $1/4\text{--}1/2$ от размера дискриминатора [10]. Поэтому для эксперимента были выбраны 4 конфигурации моделей: оригинальный BERT с 2 слоями в Encoder, ELECTRA с 2 слоями в генераторе и 4 слоями в дискриминаторе, ELECTRA с 1 слоем в генераторе и 4 слоями в дискриминаторе, ELECTRA с 1 слоем в генераторе и 2 слоями в дискриминаторе.

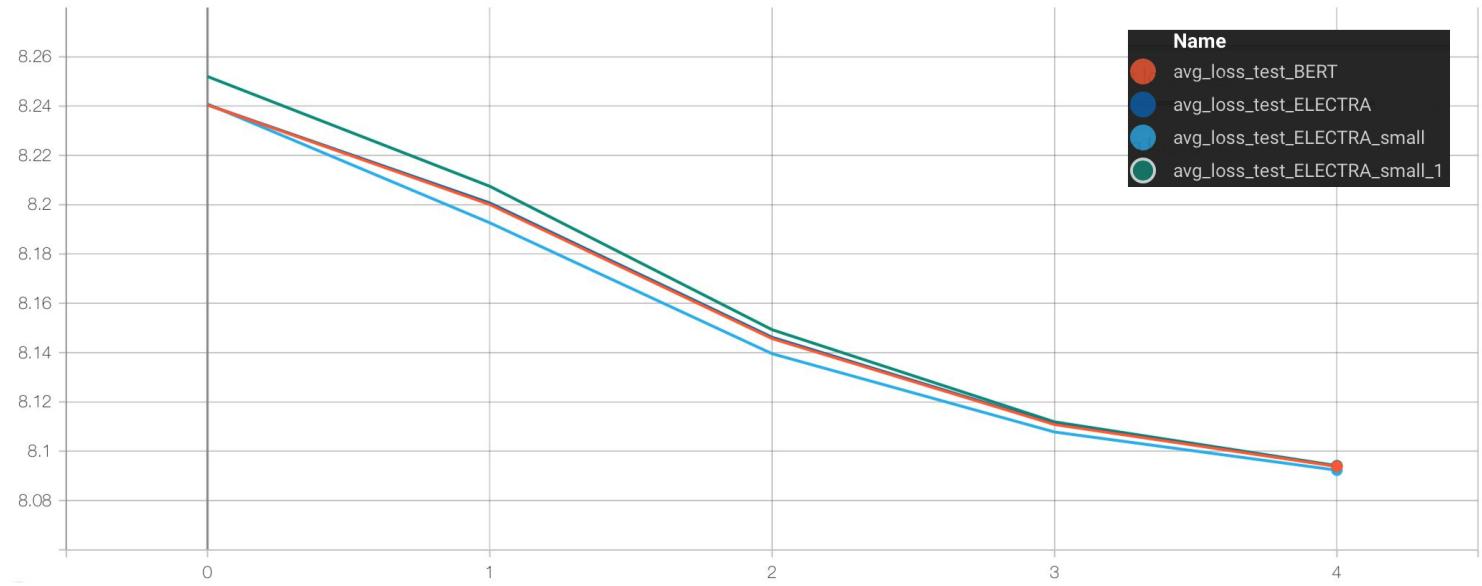


Рисунок 49 – Значение общей целевой функции на тренировочной выборке исходной модели – `avg_loss_test_orig`, модели с модификацией – `avg_loss_test_modified`

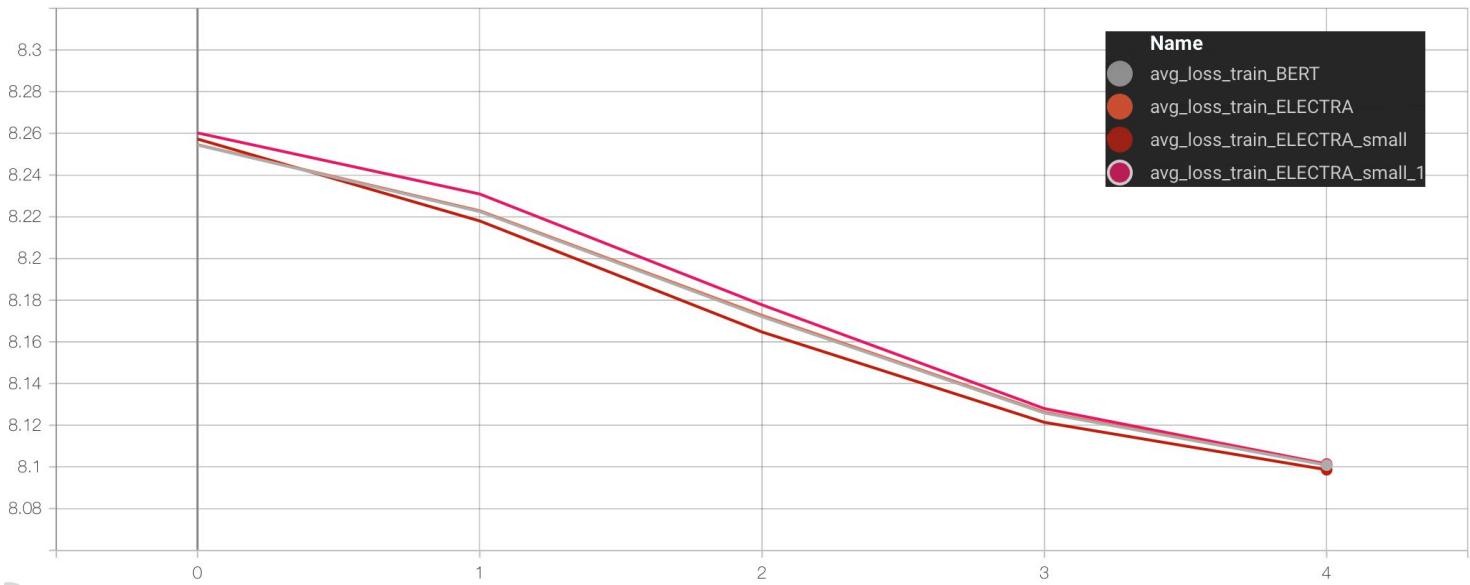


Рисунок 50 – Значение общей целевой функции на тренировочной выборке исходной модели – avg_loss_test_orig, модели с модификацией – avg_loss_test_modified

Таблица 8 – Результаты предобучения.

Конфигурация	Значение целевой функции на тренировочной выборке	Значение целевой функции на тестовой выборке
Оригинальный BERT с 2 слоями в Encoder	8.614	8.614
ELECTRA с 2 слоями в генераторе и 4 слоями в дискриминаторе	8.614	8.614
ELECTRA с 1 слоем в генераторе и 4 слоями в дискриминаторе	8.614	8.613
ELECTRA с 1 слоем в генераторе и 2 слоями в дискриминаторе	8.614	8.613

По результатам эксперимента нейронные сети BERT и ELECTRA в различных конфигурациях показали примерно сравнимые результаты, для следующего эксперимента была выбрана конфигурация ELECTRA с 1 слоем в генераторе и 2 слоями в дискриминаторе, как содержащая наименьшее количество параметров.

3.4.5 ELECTRA Pre-training с модификацией

В ходе следующего эксперимента было проведено предобучение оригинальной нейронной сети BERT и ELECTRA, предобучение производилось с использованием предложеной в работе модификацией.

Гипотеза, на основании которой проводился данный эксперимент состоит в следующем. Генератор и дискриминатор в ELECTRA согласно авторам сильно отличаются по количеству параметров, предложенный в работе метод может позволить им стабильнее обучаться благодаря постепенности наращивания сложности модели, к тому же и генератору, и дискриминатору подаются одинаковые векторные представления слов, в этом случае предложенная модификация также может помочь разным по глубине нейронным сетям быстрее обучаться.

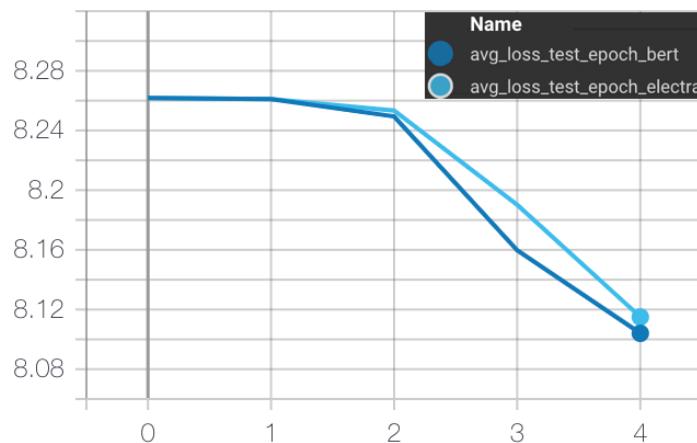


Рисунок 51 – Значение целевой функции на тестовой выборке на первой итерации BERT – avg_loss_test_epoch_bert, ELECTRA – avg_loss_test_epoch_electra

По результатам предобучения ELECTRA показала результат хуже, чем оригинальный BERT. Причем по результатам первых итераций предобучения ELECTRA обучалась лучше, чем BERT, но итоговый результат к четвертой итерации у BERT оказался лучше.

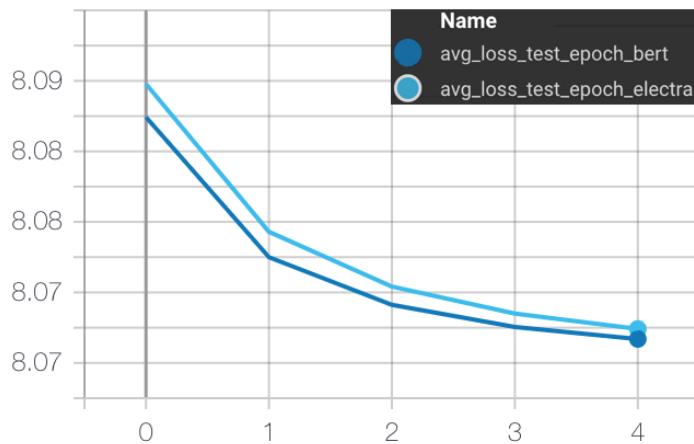


Рисунок 52 – Значение целевой функции на тестовой выборке на второй итерации BERT – avg_loss_test_epoch_bert, ELECTRA – avg_loss_test_epoch_electra

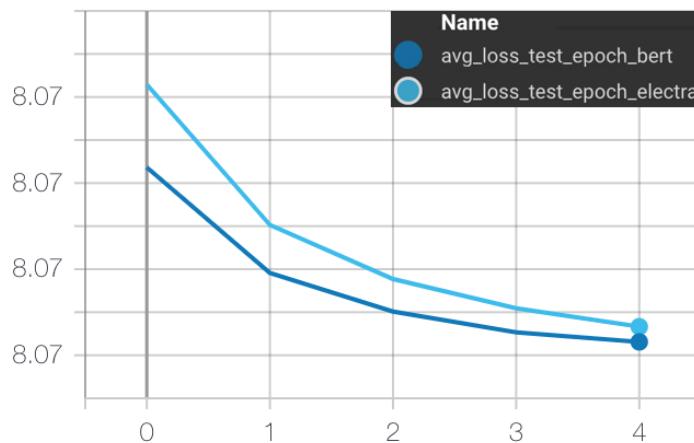


Рисунок 53 – Значение целевой функции на тестовой выборке на третьей итерации BERT – avg_loss_test_epoch_bert, ELECTRA – avg_loss_test_epoch_electra

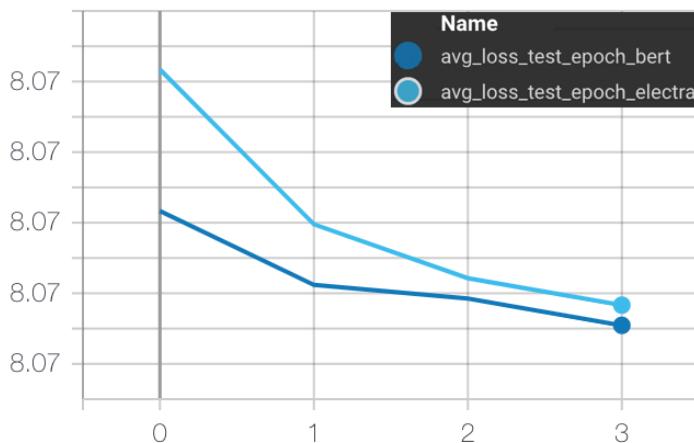


Рисунок 54 – Значение целевой функции на тестовой выборке на четвертой итерации BERT – avg_loss_test_epoch_bert, ELECTRA – avg_loss_test_epoch_electra

ЗАКЛЮЧЕНИЕ

В ходе работы была изучена и реализована нейронная сеть BERT, проведено предобучение нейронной сети 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.

В ходе работы был проведен анализ существующих методов ускорения нейронной сети Transformer. В ходе анализа были рассмотрены методы использующие распределенное обучение на кластере из тензорных или графических процессоров с применением специальных алгоритмов оптимизации, методы на основе модификации архитектуры, а также квантизации и дистилляции. Оптимальным вариантом для последующей реализации была выбрана модификация архитектуры.

Была предложена модификация нейронной сети, нацеленная на сокращение требуемых вычислительных ресурсов. Идея модификации состоит в последовательном обучении и увеличении размера слоев нейронной сети, такой подход носит название – curriculum learning. Были проведены эксперименты с применением данной модификации к нейронным сетям BERT и ELECTRA.

Рассмотренная в работе проблема вычислительной эффективности нейронной сети Transformer важна, идея лежащая в основе предложенного метода является новой, но экспериментальные результаты не показали положительного влияния модификации.

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, Łukasz 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.
7. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, Cho-Jui Hsieh, 2019.
8. Efficient Training of BERT by Progressively Stacking. Linyuan Gong, Di He, Zhuohan Li, Tao Qin, Liwei Wang, Tieyan Liu, 2019.
9. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, Radu Soricut, 2019.
10. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. Kevin Clark, Minh-Thang Luong, Quoc V. Le, Christopher D. Manning, 2020.
11. Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT. Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, Kurt Keutzer, 2019.

12. Distilling the Knowledge in a Neural Network. Geoffrey Hinton, Oriol Vinyals, Jeff Dean, 2015.
13. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. Victor Sanh, Lysandre Debut, Julien Chaumond, Thomas Wolf, 2019.
14. <https://cloud.google.com/blog/products/ai-machine-learning/now-you-can-train-ml-models-faster-and-lower-cost-cloud-tpu-pods>.
15. <https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf>.
16. <http://images.nvidia.com/content/pdf/tesla/184427-Tesla-P40-Datasheet-NV-Final-Letter-Web.pdf>.
17. Generative Adversarial Networks. Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, 2014.

ПРИЛОЖЕНИЯ

```
import math
import random
import sys
import time
from collections import Counter
from math import sqrt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from sklearn.model_selection import train_test_split
from torch.optim import AdamW
from torch.utils.data import DataLoader
from tqdm import tqdm

torch.set_default_tensor_type(torch.cuda.FloatTensor)

class BERTDataset:
    def __init__(self, corpus_path, vocab, seq_len):
        """
        :param corpus_path:
        :param vocab:
        :param seq_len:
        """
        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:
        """
        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, iteration, parts):
        """
        :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 = CustomizedLinear(iteration,
                                         get_mask(d_model, d_k, parts,
                                                   iteration)) # nn.Linear(d_m
        self.k_linear = CustomizedLinear(iteration,
                                         get_mask(d_model, d_k, parts,
                                                   iteration)) # nn.Linear(d_m
        self.v_linear = CustomizedLinear(iteration,
                                         get_mask(d_model, d_v, parts,
                                                   iteration)) # nn.Linear(d_m

        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_head]
        """

```

```

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, iteration, parts):
        """
        :param n_head: Int number of heads
        :param d_model: Int
        """
        super(MultiHeadAttention, self).__init__()

        self.iteration = iteration

        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, iteration, parts) for _ in
             range(n_head)])
    self.Linear = CustomizedLinear(iteration,
                                   get_mask(n_head * d_v, d_model, parts,
                                            iteration)) # nn.Linear(n_head)

```

```
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)

    concat_for_indices = concat.clone()

    concat_for_indices[concat_for_indices != 0] = 1
    _, indices = torch.sort(concat_for_indices, descending=True)

    new_indices = indices
    batch_add_on = torch.arange(0, concat.size(0))[:, None, None].repeat(1

    c
```

```

sample_dim_add_on = torch.arange(0, concat.size(1))[None, :, None].repeat(concat.size(0), 1, concat.size(2)).long() * concat.size(2)
new_indices = new_indices + batch_add_on + sample_dim_add_on

sorted_concat = torch.index_select(concat.view(-1), dim=-1, index=new_indices.view(-1)).view(concat.size(0), concat.size(1), concat.size(2))

linear_output = self.Linear(sorted_concat)

# 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)]


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, iteration, parts):
        """
        :param size: Int input size
        :param inner_size: Int inner size of position wise
        """
        super(PositionWise, self).__init__()
        self.fc = nn.Sequential(

```

```

        CustomizedLinear(iteration,
                           get_mask(size, inner_size, parts, iteration)),
        GELU(),
        CustomizedLinear(iteration,
                           get_mask(inner_size, size, parts, iteration))
    )

self.layer_norm = nn.LayerNorm(size)
self.dropout = nn.Dropout(0.1)

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.dropout(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,
                 iteration, parts):
        """
        :param embeddings: An float embeddings tensor with shape [b_s, seq_len]
        :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, iteration, parts),
    nn.LayerNorm(d_model),
    PositionWise(d_model, d_model * 4, iteration, parts),
    nn.Dropout(0.1)) 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, dropout in self.trans:
    input = layer_norm(input + dropout(
        multi_head_block(q=input, k=input, v=input, mask=mask)))
    input = position_wise(input)

```

```

    return input

class Generator(nn.Module):

    def __init__(self, n_heads, n_layers, vocab_size, emb_size, iteration,
                 parts):
        """
        :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, iteration, parts)
        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 Discriminator(nn.Module):

    def __init__(self, n_heads, n_layers, vocab_size, emb_size, iteration,
                 parts):
        """
        :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, iteration, parts)
        self.is_replaced = NextSentencePrediction(self.d_model)

    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.is_replaced(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 IsReplaced(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 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

class CustomizedLinearFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weight, bias=None, mask=None):
        if mask is not None:
            weight = weight * (mask.type(torch.float).t())
        output = input.matmul(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        ctx.save_for_backward(input, weight, bias, mask.type(torch.float))

```

```

    return output

@staticmethod
def backward(ctx, grad_output):
    input, weight, bias, mask = ctx.saved_tensors
    grad_input = grad_weight = grad_bias = grad_mask = None

    if ctx.needs_input_grad[0]:
        grad_input = grad_output.matmul(weight)
    if ctx.needs_input_grad[1]:
        grad_weight = grad_output.permute(0, 2, 1).matmul(input)
        if mask is not None:
            grad_weight = grad_weight * mask.t()
    if ctx.needs_input_grad[2]:
        grad_bias = grad_output.sum(0).squeeze(0)

    return grad_input, grad_weight, grad_bias, grad_mask

class CustomizedLinear(nn.Module):
    def __init__(self, i, mask, bias=False):
        super(CustomizedLinear, self).__init__()
        self.mask = mask
        self.input_features = mask.shape[0]
        self.output_features = mask.shape[1]

        self.weight = nn.Parameter(
            torch.Tensor(self.output_features, self.input_features))

```

```

if bias:
    self.bias = nn.Parameter(torch.Tensor(self.output_features))
else:
    self.register_parameter('bias', None)
if i == 1:
    self.reset_parameters()

# mask weight
self.weight.data = self.weight.data

def reset_parameters(self):
    stdv = 1. / math.sqrt(self.weight.size(1))
    self.weight.data.uniform_(-stdv, stdv)
    if self.bias is not None:
        self.bias.data.uniform_(-stdv, stdv)

def forward(self, input):
    return CustomizedLinearFunction.apply(input, self.weight * (
        self.mask.type(torch.float).t()), self.bias, self.mask)

def get_mask(input_dim, output_dim, parts, iteration):
    mask = np.zeros((input_dim, output_dim))

    for k in range(iteration):
        for i in range(int(k * input_dim / parts),
                      int(k * input_dim / parts + input_dim / parts)):
            for l in range(int(k * output_dim / parts),
                          int(k * output_dim / parts + output_dim / parts)):
```

```

        mask[i, l] = 1

    for g in range(0, int(k * output_dim / parts + output_dim / parts)):
        for f in range(0, int(k * input_dim / parts)):
            mask[f, g] = 1

    return torch.tensor(mask).int()

def prepare_data(dataset_path, seq_len, test_size, batch_size):
    with open(dataset_path, "r") as f:
        vocab = Vocab(f)

    dataset = BERTDataset(dataset_path, vocab, seq_len)

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

    train_data_loader = DataLoader(train, batch_size, shuffle=True)
    test_data_loader = DataLoader(test, batch_size)

    return train_data_loader, test_data_loader, len(vocab)

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

# with open("/content/dataset_small.pickle", "wb") as f:

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

```

```

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

"""---Dataset params---"""
dataset_path = "/content/dataset_partition.txt"
seq_len = 64
batch_size = 256
test_size = 0.2

"""---Model params---"""
emb_size = 128
# emb_sizes =
n_heads = 3
n_layers = 4

"""---Training params---"""
epochs = 5 # 3
learning_rate = 1e-4
num_total_steps = 1000
num_warmup_steps = 100

"""---Prepare data---"""
train_data_loader, test_data_loader, len_vocab = prepare_data(dataset_path,
                                                               seq_len,
                                                               test_size,
                                                               batch_size)

```

```

"""---Optimizers and losses---"""
masked_criterion = nn.CrossEntropyLoss(ignore_index=0)
next_criterion = nn.CrossEntropyLoss()

# model = torch.load("/content/model_epoch_7.pth").to('cuda')

iterations = 4
parts = 4

start_training = time.time()
limit = False

for i in range(1, iterations + 1):
    print(
        "-----"
    )
    print("Iteration = " + str(i))
    print("-----")

    if i == 2:
        break

    if i == 1:
        model = Model(n_heads, n_layers, len_vocab, emb_size, i, parts).to(
            'cuda')
    else:
        model = Model(n_heads, n_layers, len_vocab, emb_size, i, parts).to(
            'cuda')
    model.load_state_dict(torch.load("colab.pth"))

```

```

# model = Model(n_heads, n_layers, len_vocab, emb_size, i).to('cuda')

optimizer = AdamW(model.parameters(), lr=learning_rate)
optim_scheduler = ScheduledOptim(optimizer, emb_size, n_warmup_steps=10000)

for epoch in range(epochs):
    if epoch == 0:
        testing_time = 0

    # Training stage
    model.train()

    avg_loss, avg_next_loss, avg_mask_loss, total_correct, total_element = 0, 0, 0, 0, 0

    with tqdm(total=len(train_data_loader), file=sys.stdout, position=0,
              leave=True,
              desc='Epoch {} Training stage'.format(epoch)) as pbar:
        for i, data in enumerate(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"])

            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()

# loss.backward()
# torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_
# optimizer.step()
# scheduler.step()
# optimizer.zero_grad()

optim_scheduler.zero_grad()
loss.backward()
optim_scheduler.step_and_update_lr()

pbar.update()

end_training = time.time()
training_time = end_training - start_training - testing_time
if (training_time / 60) > 100:
    pass
    # limit = True
    # break

start_testing = time.time()

```

```

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

print('Avg loss train' + " - " + "{0:.4f}".format(avg_loss / (i + 1)))
print('Acc test      ' + " - " + "{0:.4f}".format(
    total_correct * 100.0 / total_element))

tb.save_value('Avg loss train', 'avg_loss_train_epoch_modified', epoch,
              avg_loss / (i + 1))
tb.save_value('Avg next loss train',
              'avg_next_loss_train_epoch_modified', epoch,
              avg_next_loss / (i + 1))
tb.save_value('Avg mask loss train',
              'avg_mask_loss_train_epoch_modified', epoch,
              avg_mask_loss / (i + 1))
tb.save_value('Acc train', 'acc_train_epoch_modified', epoch,
              total_correct * 100.0 / total_element)

print()

# Testing stage
model.eval()

avg_loss, avg_next_loss, avg_mask_loss, total_correct, total_element = 

with tqdm(total=len(test_data_loader), file=sys.stdout, position=0,
          leave=True,
          desc='Epoch {} Testing stage '.format(epoch)) as pbar:

```

```

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

    with torch.no_grad():
        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()

    pbar.update()

# test_loss.append(avg_loss)
# test_acc.append(total_correct * 100.0 / total_element)

```

```

print('Avg loss test ' + " - " + "{0:.4f}".format(avg_loss / (i + 1)))
print('Acc test         ' + " - " + "{0:.4f}".format(
    total_correct * 100.0 / total_element))

print(epoch)

tb.save_value('Avg loss test', 'avg_loss_test_epoch_modified', epoch,
              avg_loss / (i + 1))
tb.save_value('Avg next loss test', 'avg_next_loss_test_epoch_modified',
              epoch, avg_next_loss / (i + 1))
tb.save_value('Avg mask loss test', 'avg_mask_loss_test_epoch_modified',
              epoch, avg_mask_loss / (i + 1))
tb.save_value('Acc test', 'acc_test_epoch_modified', epoch,
              total_correct * 100.0 / total_element)

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

if epoch + 1 != epochs: print("-----")

if limit == True:
    break

end_testing = time.time()
testing_time = end_testing - start_testing

torch.save(model.state_dict(), "colab.pth")

print(training_time)

```