

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ  
ВЫСШЕГО ОБРАЗОВАНИЯ  
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ  
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

**Факультет информатики, математики и компьютерных наук**

**Программа подготовки бакалавров по направлению  
01.03.02 Прикладная математика и информатика**

*Сидорова Анна Павловна*

**КУРСОВАЯ РАБОТА**

Применение нейронных сетей для анализа тональности твитов

Научный руководитель  
Старший преподаватель

Семенов Дмитрий Павлович

Нижний Новгород, 2023

# Содержание

<b>Введение</b>	<b>4</b>
<b>1 Постановка задачи</b>	<b>5</b>
<b>2 Обзор литературы</b>	<b>6</b>
<b>3 Теория нейронных сетей и трансформеров</b>	<b>7</b>
3.1 Нейронные сети . . . . .	7
3.1.1 Определение . . . . .	7
3.1.2 Основные компоненты . . . . .	7
3.1.3 Структура связи . . . . .	9
3.1.4 Обучение нейронной сети . . . . .	10
3.2 Трансформеры . . . . .	14
3.2.1 Историческая справка . . . . .	15
3.2.2 Рекуррентные нейронные сети . . . . .	17
3.2.3 Архитектура . . . . .	17
3.2.4 Positional Encoding . . . . .	18
3.2.5 Энкодер . . . . .	19
3.2.6 Кратко о декодере . . . . .	26
<b>4 BERT</b>	<b>28</b>
4.1 Входная последовательность . . . . .	28
4.2 Архитектура . . . . .	30
4.3 Pre-training . . . . .	31
4.4 Fine-tuning . . . . .	32
<b>5 Анализ тональности твитов</b>	<b>33</b>
5.1 История использования нейронных сетей в задаче классификации сентимента . . . . .	33
5.2 Задача . . . . .	34
5.3 Датасет . . . . .	34
5.3.1 RuSentiTweet . . . . .	34
5.3.2 Tweets Dataset . . . . .	36

5.3.3	Объединение . . . . .	38
5.4	Метрики . . . . .	39
5.5	Fine-Tuning . . . . .	40
5.5.1	RuSentiTweet - binary . . . . .	40
5.5.2	Tweets Dataset - binary . . . . .	43
5.5.3	Многоклассовая классификация . . . . .	46
5.6	Результаты . . . . .	61
	<b>Заключение</b>	<b>62</b>
	<b>Библиографический список</b>	<b>64</b>

# **Введение**

Классификация сентимента текста является важным инструментом анализа текстовой информации, который позволяет автоматически определять эмоциональную оценку определенного текста.

Это позволяет оптимизировать многие бизнес-процессы, такие как мониторинг общественного мнения относительно продукции или услуг, извлечение ценной информации из многих источников, анализ и оценку репутации компаний на рынке, а также предпринимать действия на основе этих данных для улучшения качества продукта или услуги.

Классификация сентимента текста может быть полезна, когда речь идет о социальных сетях, таких как Twitter, Facebook и LinkedIn. Обработка служебных сообщений и отзывов пользователей может использоваться для принятия бизнес-решений, связанных с продуктами и услугами.

Например, если общественное мнение относительно продукции положительно, то компании необходимо продолжить развивать и улучшать продукт, чтобы продолжать удерживать и привлекать новых клиентов. Если же общественное мнение отрицательно, то принимаются меры по преодолению проблем, улучшению качества продукта или предоставлению более качественного сервиса.

Таким образом, классификация сентимента текста является важным инструментом, который помогает компаниям глубже понимать потребности и чувства клиентов, а также принимать правильные бизнес-решения на основе этих данных.

В данной работе исследуются нейронные сети и практика на их применение в задаче анализа тональности твитов.

# 1 Постановка задачи

Для реализации данной работы были поставлены задачи:

1. Изучить и описать работу нейронных сетей.

А именно: дать определение, описать основные компоненты, объяснить структуру связи и метод обучения.

2. Изучить и описать архитектуру, которая используется в задаче.

Подробнее: объяснить архитектуру трансформер - то, на чем основана архитектура BERT. Дать понятие работы self-attention, Multi-Head attention, positional encoding. Объяснить архитектуру BERT, как происходит предобучение и дообучение.

3. Найти качественный сет данных.

Основная сложность состоит в том, что сами твиты должны быть на русском языке, когда основная масса данных на сегодняшний день на английском.

4. Сделать предобработку данных.

5. Найти подходящие метрики.

6. Дообучить BERT на задачу сентимент анализа и предоставить готовую модель.

## **2 Обзор литературы**

В любой работе различные источники играют важную роль. Данная работа опирается на статьи (1), (2), (4) и сайт (3) в первой главе, чтобы дать базовые определения нейронной сети, её компонент и методов обучения.

В свою очередь статьи (5), (6), (7) вводят в историю и устройство ныне популярной архитектуры - Transformer. Информация из данных источников также используется в первой главе.

Во второй главе идет описание архитектуры BERT, которое опирается на статью разработчиков (8).

# 3 Теория нейронных сетей и трансформеров

## 3.1 Нейронные сети

Перед началом построения и обучения модели следует изучить теорию. Базовое понятие, которое следует изучить в первую очередь - нейронные сети.

### 3.1.1 Определение

Обращаясь к (1) нейронные сети представляют собой устройства параллельных вычислений, состоящие из множества взаимодействующих простых процессоров. Более математически - сложная дифференцируемая функция, которая задается как  $f : X \rightarrow Y$ , где  $X$  - пространство признаков, а  $Y$  - ответов. Все параметры нейронной сети настраиваются синхронно и зависимо друг от друга.

Несмотря на термин “сложная дифференцируемая функция” сложную функцию всегда можно представить как суперпозицию простых, что и имеется ввиду в части “состоящие из множества взаимодействующих простых процессоров”. Обычно в пример взаимодействия приводятся два самых распространенных слоя (иными словами - функции, с помощью которых преобразуются входные данные):

1. Линейный слой - использование линейной функции над данными;
2. Функция активации - нелинейное преобразование над данными. Благодаря данными преобразованиям нейронные сети способны создавать более содержательные признаки.

### 3.1.2 Основные компоненты

Как и было описано ранее нейронная сеть это совокупность связанных каким-то способом элементов для обеспечения взаимодействия между собой. Этими компонентами называются нейроны или узлы.

Данные элементы являются простыми процессорами. Указанные процессоры обладают некой “вычислительной возможностью”. Ограничиваются эта возможность двумя правилами: комбинирование входных сигналов и активизации.

Применение этих правил образуют выходной сигнал из нейрона, который может пойти в другие нейроны по взвешенным связям, которые имеют свой весовой коэффициент или просто вес. За счет этого какая-то связь может иметь больший или меньший приоритет.

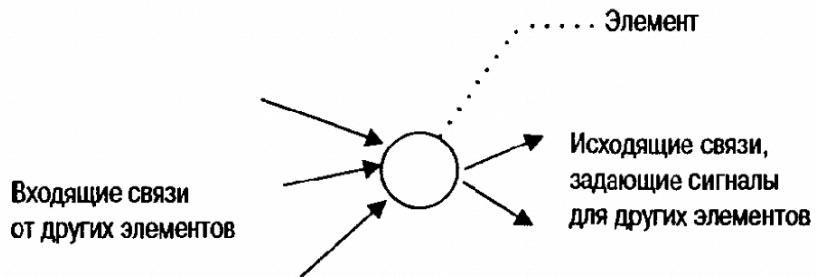


Рис. 1: Элемент сети

Несмотря на то, что по отдельности нейроны имеют ограничение по вычислительной возможности, объединённые в одну большую сеть элементы образуют нейронную сеть, способную решать сложные задачи.

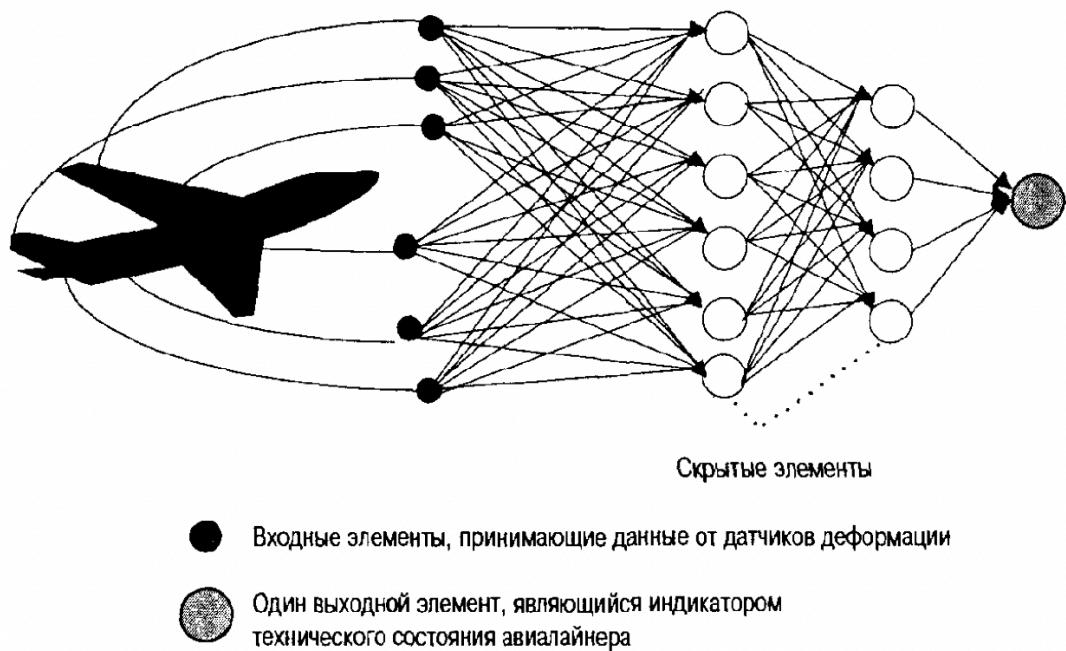


Рис. 2: Схема применения нейронной сети для контроля технического состояния авиалайнера.

### 3.1.3 Структура связи

Стоит также упомянуть о том, какие связи в нейронной сети существуют. В некоторых моделях все узлы соединены между собой, в других моделях связи организованы иерархически между смежными слоями. В некоторых моделях есть обратные связи, в том числе между или внутри слоев, и рефлексивность тоже может быть допущена. Способы здесь многообразны.

Связь имеет следующие параметры:

1. Элемент с исходящей связью;
2. Элемент со входящей связью;
3. Число, указывающее на вес, который данная связь имеет.

Вес - действительное число, модуль которого говорит о силе связи. Если вес отрицательный, то связь “подавляется”, если положительный, то “усиливается”.

Сама структура связи представляется в виде матрицы  $W$ , каждый элемент ( $w_{ij}$ ) которой представляет вес конкретной связи, идущей из нейрона  $i$  в нейрон  $j$ .  $W$  - память нейронной сети, с помощью которой она понимает как выполняется задача.

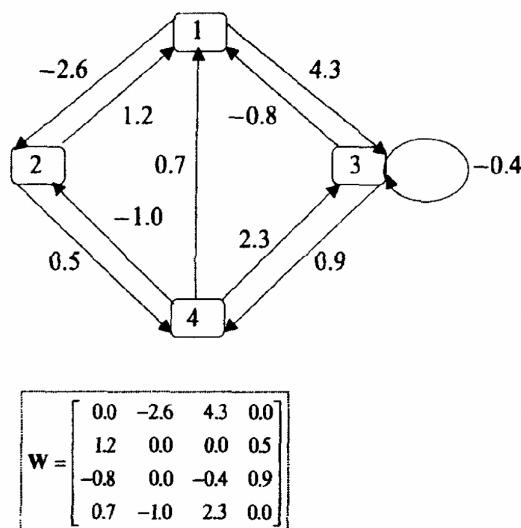


Рис. 3: Пример матрицы  $W$ .

### 3.1.4 Обучение нейронной сети

Данная секция обращается к статьям (2), (3), (4) за определениями математических терминов и описания обучения нейронных сетей.

#### Метод обратного распространения ошибки

Перед тем, как будет описан метод обучения нейронных сетей, хотелось бы затронуть несколько определений:

##### 1. Градиентный спуск:

Один из основных методов оптимизации. Метод нахождение локального максимума или минимума функции используя градиент (вектор, показывающий направление скорейшего роста функции или скалярной величины). Используется в “чистом” виде не часто.

Идея:

$$\vec{x}^{[j+1]} = \vec{x}^{[j]} - \lambda^{[j]} \nabla F(\vec{x}^{[j]}),$$

где  $\vec{x}^{[j]}$  - приближение,  $\lambda^{[j]}$  - скорость шага градиентного спуска,  $F$  - целевая функция,  $-\nabla F$  - антиградиент (направление скорейшего убывания).

##### 2. Стохастический градиентный спуск:

Метод, основанный на градиентном спуске, отличающийся только тем, что вместо того, чтобы вычислять градиент оптимизируемой функции как сумму градиентов от каждого элемента выборки, он вычисляется на каждом шаге как градиент от одного из случайно выбранных элементов.

##### 3. Функция потерь (loss function):

Данная функция помогает на основе неправильно принятых решений в данных описать потерю. То есть определяет насколько модель грамотно работает. Существует множество таких функций как и для бинарной, так и для многоклассовой классификации.

##### 4. Forward pass (forward propagation, прямой проход):

На моменте forward propagation начальные данные последовательно преобразуются путем последовательного применения слоёв к предыдущим представлениям, проходя весь путь от входных элементов к целевому значению (выходному элементу).

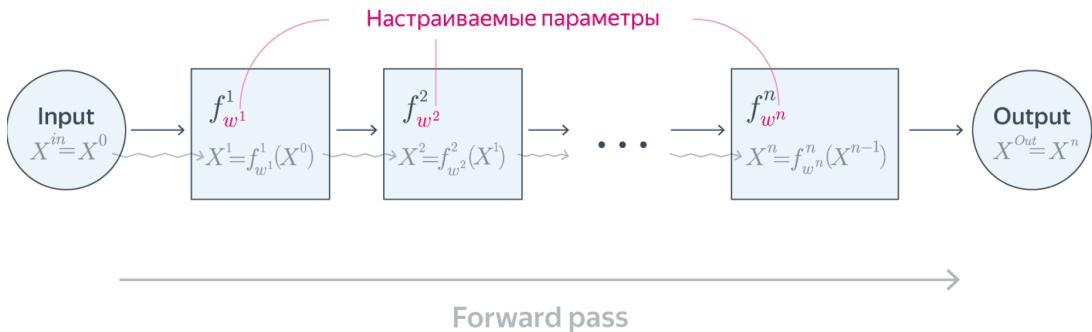


Рис. 4: Пример прямого прохода.

## 5. Backward propagation (обратное распространение):

Обратный режим автоматической дифференциации или обратного накопления. Иными словами, данные идут от выходного значения к входным элементам через все преобразования. Данными обычно являются сведениями об ошибке предсказания, а финальным представлением часто является функция потерь.

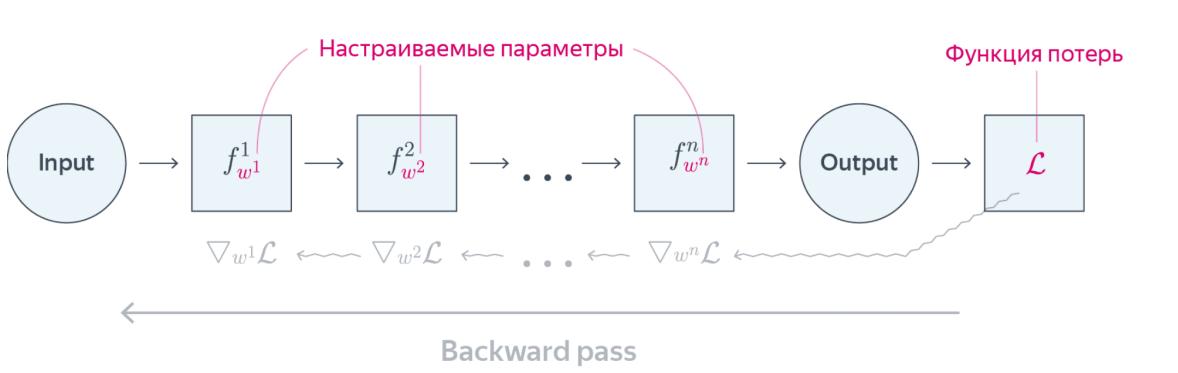


Рис. 5: Пример обратного распространения.

## Кратко

Когда мы используем нейронную сеть с прямым проходом для приема входных данных и получения целевого значения, информация передается по сети вперед. А алгоритм обратного распространения позволяет информации о стоимости затем передаваться в обратном направлении по сети для вычисления градиента.

По своей сути нейронные сети учатся с помощью разных вариаций градиентного спуска, поэтому нужно знать и уметь эффективно с ним работать.

Градиентный спуск используется на функции потерь. А как известно, это сложные функции и вся концепция этой идеи может уместиться в одной строчке:

$$f(x) = g_m(g_{m-1}(\dots(g_1(x))\dots)) \rightarrow \frac{\partial f}{\partial x} = \frac{\partial g_m}{\partial g_{m-1}} \cdot \frac{\partial g_{m-1}}{\partial g_{m-2}} \cdot \dots \cdot \frac{\partial g_2}{\partial g_1} \cdot \frac{\partial g_1}{\partial x}$$

Данная формула говорит о возможности последовательного вычисления градиента, производя умножение на частные производные предыдущего слоя.

Пример, основанный на одном наблюдении с функцией потерь:

$$-\log(1 + \exp[-y(\omega_0 + \omega_1x_1 + \omega_2x_2)])$$

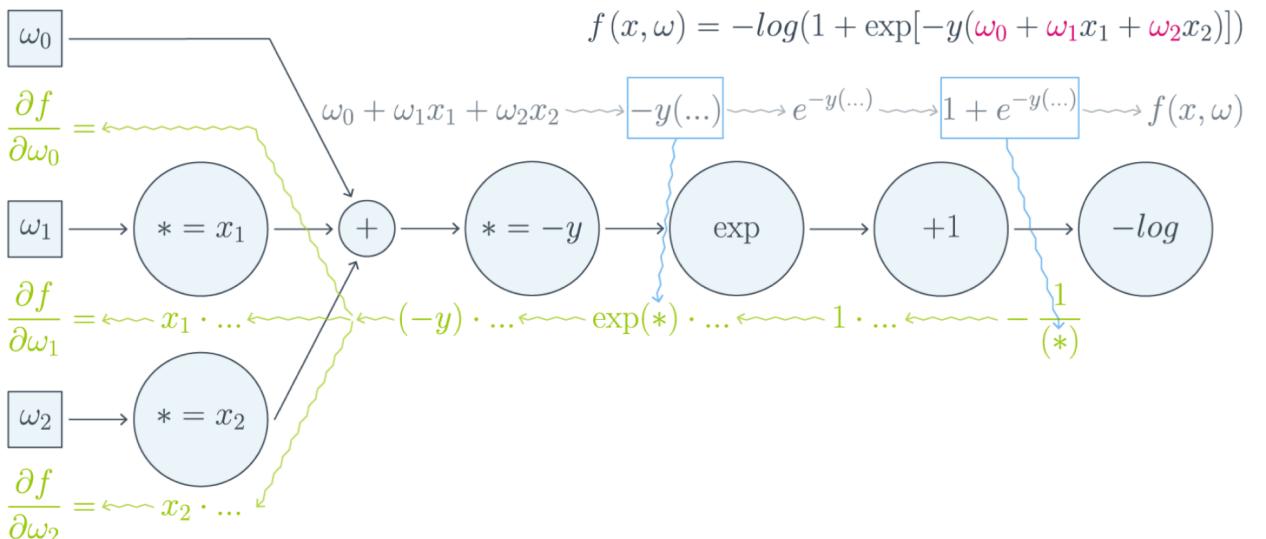


Рис. 6: Пример применения метода обратного распространения ошибки.

Собирая все воедино:

$$\begin{aligned}\frac{\partial f}{\partial \omega_0} &= 1 \cdot (-y) \cdot e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)} \cdot \frac{-1}{1 + e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)}} \\ \frac{\partial f}{\partial \omega_1} &= x_1 \cdot (-y) \cdot e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)} \cdot \frac{-1}{1 + e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)}} \\ \frac{\partial f}{\partial \omega_2} &= x_2 \cdot (-y) \cdot e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)} \cdot \frac{-1}{1 + e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)}}\end{aligned}$$

### Подробнее

Соответственно, алгоритм обратного распространения ошибки хорош тем, что выполняет численное вычисление градиента с помощью простой и недорогой процедуры.

Стоит отметить, что *backward propagation* часто считают целостным алгоритмом для обучения многослойных нейронных сетей, но на самом деле он относится только к методу вычисления градиента. Обучением нейронных сетей, как таковых, занимается, например, стохастический градиентный спуск.

Также стоит отметить, что данный метод может вычислять производные абсолютно любой функции (для некоторых функций правильным ответом будет сообщение о неопределенной производной) и подходит не только для многослойных нейронных сетей.

Интуитивно способ работы *backward propagation* был описан, однако математически всё ещё не ясен общий вид.

Пусть  $f(x) = g(h(x))$  - скалярная функция. Как сказано в (3) “сердцем механизма обратного распространения ошибки” является следующая формула:

$$\nabla_{x_0} f = [D_{x_0} h] * \nabla_{h(x_0)} g -$$

применение сопряжённого к  $D_{x_0} h$  линейного отображения к вектору  $\nabla_{h(x_0)} g$ , где  $D_{x_0} h$  - дифференциал функции  $h$ .

### Вывод формулы:

Зная формулу производной сложной функции:

$$[D_{x_0} u \circ v](\nabla x) = [D_{v(x_0)} u]([D_{x_0} v](\nabla x))$$

Найдём градиент для  $f(x)$ :

$$[D_{x_0} f](x - x_0) = \langle \nabla_{x_0} f, x - x_0 \rangle$$

Найдём градиент для  $g(h(x))$ :

$$[D_{h(x_0)}g]([D_{x_0}h](x - x_0)) = \langle \nabla_{h(x_0)}g, [D_{x_0}h](x - x_0) \rangle = \langle [D_{x_0}h] * \nabla_{h(x_0)}g, x - x_0 \rangle$$

Тогда, исходя из вышеперечисленного:

$$\nabla_{x_0}f = [D_{x_0}h] * \nabla_{h(x_0)}g$$

По данной формуле становится понятно то, что из некоторого промежуточного представления  $X^r$  при получении градиента функции потерь и при условии понимания того, как преобразуется градиент при проходе через слой  $f^r$  между  $X^{r-1}$  и  $X^r$ , что мы ту же способны найти градиент по переменным из  $X^{r-1}$ .

Подводя итог методу обратного распространения ошибки, ниже приведен общий алгоритм. Допустим идёт обучение нейронной сети с помощью стохастического градиентного спуска, имеется матрица текущих значений весов  $W_0^i$  и мини-батч  $X$ . Чтобы сделать шаг стохастического градиентного спуска:

1. Сделать прямой проход по нейронной сети. Вычислить и запомнить все промежуточные представления  $X = X^0, X^1, \dots, X^m = \hat{y}$ .
2. Вычислить все градиенты с помощью метода обратного распространения ошибки.
3. Совершить шаг с помощью полученных градиентов.

## 3.2 Трансформеры

Трансформеры напрямую связаны с темой данной работы, ведь архитектура BERT, заявленная в описании темы, расшифровывается как Bidirectional Encoder Representations from Transformers, что обозначает “Представление двунаправленного энкодера от трансформеров”. Поэтому очень важно описать работу энкодеров трансформеров для понимания работы архитектуры BERT.

### **3.2.1 Историческая справка**

#### **История до нейронных сетей в задаче машинного перевода.**

Двигателем развития человечества и порождаемых ею наук является война. Предысторией появления трансформеров и в целом развитие нейронных сетей является холодная война между СССР и США. Начало всего - задача машинного перевода текстов.

Изначально были методы машинного перевода, основанные на правилах. Данные системы основывались на изучении работы переводчиков, то есть брался словарь и имелся определенный набор лингвистических правил.

Затем пришли методы перевода, основанные на примерах. Здесь использовались готовые фразы и весь перевод заключался в нахождении отличий между предложениями. Позже были придуманы статистические методы перевода, основанные на вычисления закономерностей. Статистический метод в свое время использовал Google в своём переводчике.

#### **История после появления нейронных сетей в задаче машинного перевода.**

В 2014 году была выпущена статья (5), которая рассказывала об использовании нейронных сетей в машинном переводе. Здесь была представлена архитектура нейронной сети, которую назвали RNN Encoder-Decoder, состоящая из двух рекуррентных нейронных сетей (RNN), которые действуют как пара Encoder-Decoder.

Кодировщик преобразует исходную последовательность переменной длины в вектор фиксированной длины, а декодер преобразует векторное представление обратно в целевую последовательность переменной длины. Две сети обучаются совместно, чтобы максимизировать условную вероятность целевой последовательности при заданной исходной последовательности. Проще говоря, энкодер кодирует исходный текст в вектор, а декодер, зная только один язык, учится декодировать числа в связный текст на языке, отличным от энкодера.

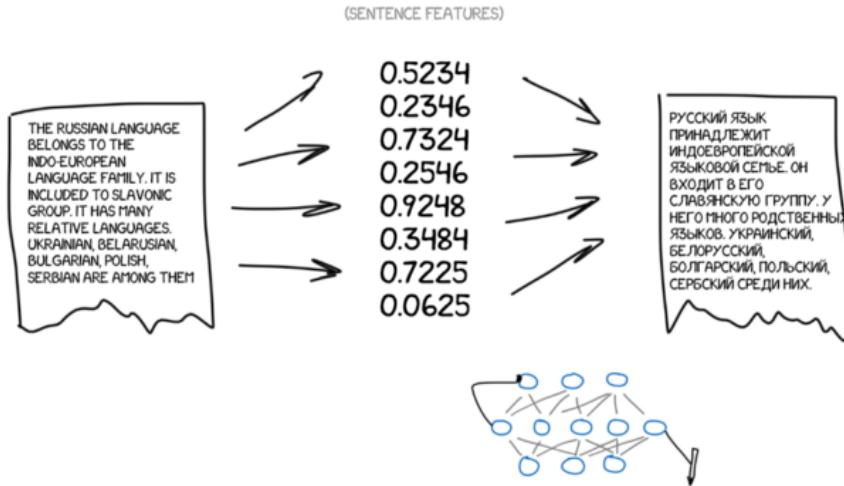


Рис. 7: Пример энкодер-декодера.

В 2016 году Google подхватил идею такой сети и изменил структуру своего перевода, представив в статье (6) свою архитектуру нейронной сети для машинного перевода. Данная модель состояла из глубокой сети LSTM с 8 блоками энкодеров и 8 блоками декодеров, использующими residual соединения как и attention соединения от сети декодера к энкодеру. Чтобы улучшить параллелизм и, следовательно, сократить время тренировки, механизм внимания соединяет нижний уровень декодера с верхним уровнем энкодера.

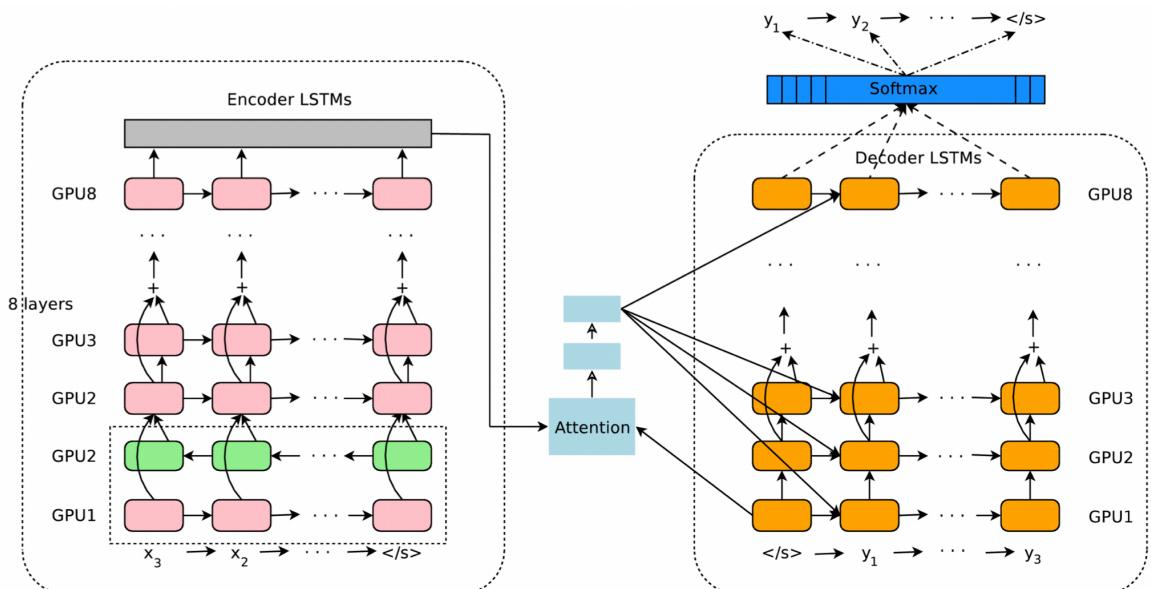


Рис. 8: Архитектура Google's Neural Machine Translation system.

### **3.2.2 Рекуррентные нейронные сети**

Как и было понятно из секции выше, популярным решением в 2015-ых годах являлись рекуррентные нейронные сети. Причины были следующими:

1. Данный вид нейронных сетей может использовать свои скрытые состояния как память для обработки последовательностей входов. То есть они могли улавливать контекст.
2. Были сравнительно точны и эффективны.

Стоит упоминания то, что в задачах NLP(Natural Language Processing) также активно использовались CNN. Однако данная работа не настолько глубоко затрагивает данную тему.

Однако у рекуррентных нейронных сетей были минусы:

1. Как и говорилось выше, данные нейронные сети хранят всю информацию в скрытых слоях. Информация с каждым шагом обновляется, поэтому, если необходимо вспомнить, что было ранее, придется пройти “ $n$ ” шагов назад. Сама операция “вспомнить” возможна только если иметь большие скрытые слои (то есть хранить всю информацию). Либо иметь альтернативу - терять информацию.
2. RNN обрабатывают предложение последовательно, поэтому увеличивается время обучения (то есть чтобы высчитать шаг  $n$ , надо знать  $n - 1$  шаг). Плюсом такой подход не подходит для GPU, созданный для параллельных вычислений.

И в 2017 году была обнародована статья (7) с нейронной сетью, которая исправляла вышеописанные минусы и не использовала ни рекуррентные, ни свёрточные сети. Данная архитектура была названа трансформеры.

### **3.2.3 Архитектура**

В официальной статье (7) была представлена следующая схема:

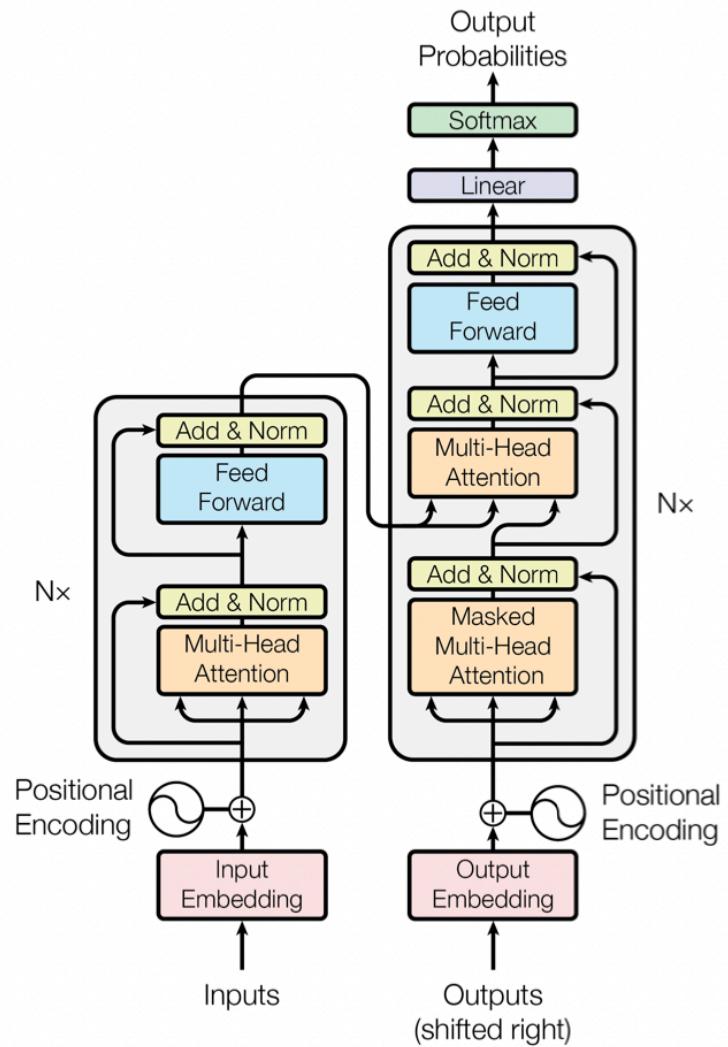


Рис. 9: Архитектура Transformer.

Подробно ниже будет описан энкодер данной модели и чуть затронуты механизмы декодера.

### 3.2.4 Positional Encoding

Positional Encoding - это метод кодирования подряд идущих токенов в модели Transformer, который обрабатывает информацию о порядке слов в предложении. Он добавляется к эмбеддингам для того, чтобы сеть могла учитывать порядок слов в предложении.

Positional Encoding использует формулу:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

где  $i$  - расположение по счету компоненты в векторе,  $pos$  - позиция каждого слова в предложении, а  $d$  - размерность вектора.

Positional Encoding создает вектор, который добавляется к эмбеддингу. Для каждой позиции вектор уникальный и показывает порядок слов в предложении. Таким образом, при обучении модели учитывается не только содержание каждого слова, но и его место в предложении.

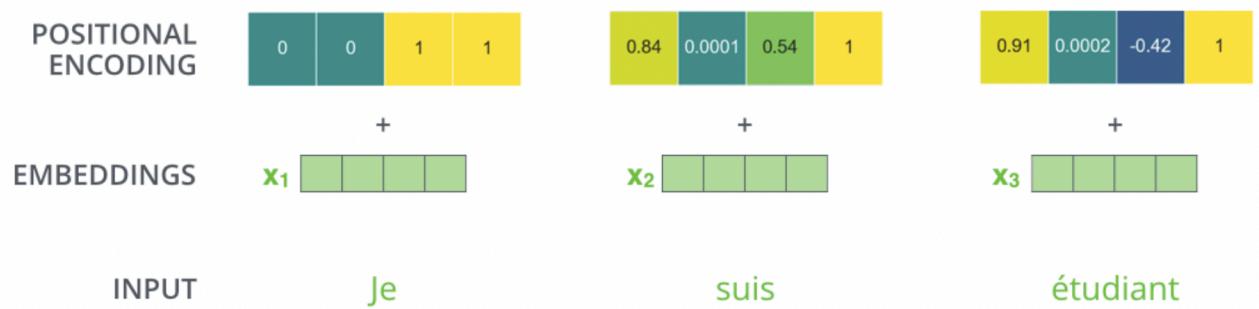


Рис. 10: Пример Positional Encoding при  $d = 4$ .

### 3.2.5 Энкодер

В данной модели энкодер выглядит следующим образом:

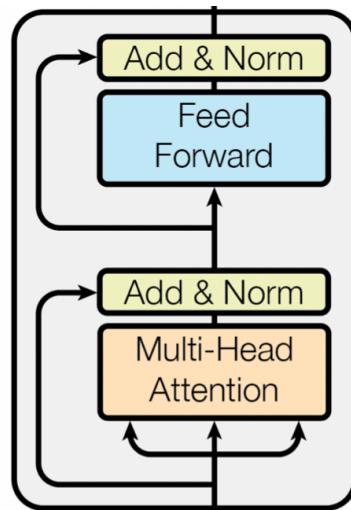


Рис. 11: Энкодер.

Энкодер трансформеров - это модуль глубокого обучения, предназначенный для обработки последовательностей данных.

Основная задача энкодера трансформеров заключается в преобразовании входных последовательностей токенов в векторные представления, или так называемые “эмбеддинги”.

Сам энкодер состоит из 6 идентичных слоев, каждый из которых состоит из двух подслоев:

1. механизм Multi-Head self-attention;
2. FFNN (fully connected feed-forward network).

Рассмотрим устройство покомпонентно.

## **Multi-Head Attention**

Multi-Head Attention позволяет энкодеру сфокусироваться на наиболее значимых токенах входной последовательности и давать им больший вес при генерации эмбеддингов. Это позволяет достичь лучших результатов при обработке текста.

За счет данного блока идет изучение и понимание контекста. Но перед тем, как пояснить его схему работы, стоит пояснить как действует обычный механизм внимания.

## **Self-attention**

Self-attention представляет собой процесс вычисления взвешенной суммы значений входной последовательности с использованием весов, которые определяются с помощью функции внимания. Веса вычисляются для каждого элемента входной последовательности с учетом его сходства с другими элементами. Эта взвешенная сумма затем используется для создания новой последовательности, которая представляет собой комбинацию исходных элементов с учетом их важности.

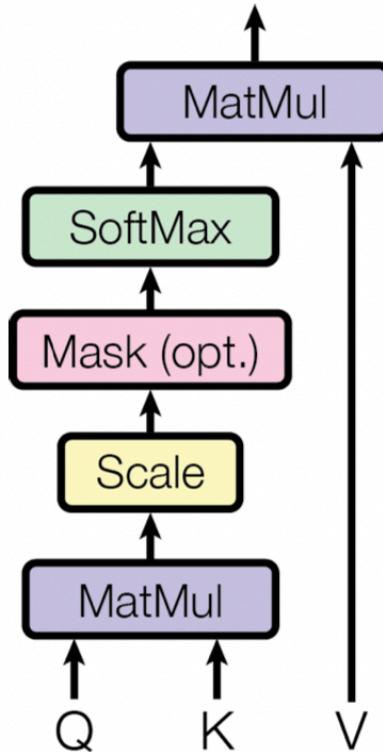


Рис. 12: Self-attention механизм.

На Рис. 12 изображен механизм внимания, где MatMul - матричное умножение, Scale - деление компонентов на величину  $\sqrt{d_k}$  - корень из размерности вектора Key, Mask - маскирование (приутствует только в декодере), Softmax - применение функции softmax.

Подробнее о процессе:

1. Первым шагом создаются три матрицы: Query, Key и Value, которые синхронизируются по размеру входной последовательности.

Значения Query, Key и Value получаются путем преобразования исходного текста на каждом шаге итерации. Текст проходит через три линейных слоя (для Key, Value и Query). Каждый из этих слоев имеет свой набор весов  $W_q, W_k, W_v$ , который обучается во время обучения модели, применяя обратное распространение ошибки для оптимизации весовых матриц (для архитектуры Transformer).

Key и Value представляют собой информацию о входных данных, а Query представляет запрос, который используется для получения значений из

Key и Value. Проще говоря Query говорит нам о значении слова, когда мы смотрим “из него”, а значение Key, когда мы смотрим “в него”. Value - сами значения.

Матрицы весов для Key и Value имеют одинаковые размерности и зависят от размера входной последовательности, так как они вычисляются для каждого слова. Матрица весов для Query имеет размерность, соответствующую размеру скрытого слоя модели.

**Пример:** Пусть мы имеем весовые матрицы  $W_q, W_k, W_v$  размерностью  $(n, m)$  и фразу “Машина думает”. Для каждого слова представлен эмбеддинг  $x_1, x_2$  соответственно. Каждый вектор  $x$  имеет размерность  $(1, n)$ . Матрица эмбеддингов  $X$  состоит из векторов  $x_1, x_2$ .

Тогда Query, Key и Value равны:

$$\begin{cases} Q = X \times W_q, \\ K = X \times W_k, \\ V = X \times W_v \end{cases} \quad (1)$$

2. Перемножение матрицы Query на матрицу  $Key^T$ , для вычисления попарного сходства между каждой парой элементов входной последовательности.

**Пример:** Из пункта выше получаем:

$$A = Q \times K^T$$

3. Масштабирование, путём деления матрицы из п. 2 на  $\sqrt{d_k}$ .

**Пример:**

$$A_{sq} = \frac{A}{\sqrt{d_k}}$$

Деление выполняется для того, чтобы дисперсия величины  $\xi = \langle q, k \rangle$ , где  $q$  - Query одного из слов в матрице, а  $k$  - Key того же слова, равнялась единице. Иначе данная скалярная величина будет обладать большой дисперсией, так как она зависит от размерности вектора.

$$\langle q, k \rangle = \sum_{i=1}^m q_i k_i = \sum_{i=1}^m \xi_i, \xi_i \sim N(0, 1)$$

$$\xi = \langle q, k \rangle = \sum_{i=1}^m \xi_i$$

$$\mathbb{E}\xi = \mathbb{E} \sum_{i=1}^m \xi_i = \sum_{i=1}^m \mathbb{E}\xi_i = m\mathbb{E}\xi_i = 0$$

$$\mathbb{D}\xi = \mathbb{D} \sum_{i=1}^m \xi_i = \sum_{i=1}^m \mathbb{D}\xi_i = m\mathbb{D}\xi_i = m$$

Воспользуемся свойством:  $\mathbb{D}\frac{\xi}{a} = \frac{\mathbb{D}\xi}{a^2}$ . Тогда:

$$\mathbb{D}\frac{\xi}{\sqrt{m}} = \frac{\mathbb{D}\xi}{m} = 1 \Rightarrow \frac{\xi}{\sqrt{m}} \sim N(0, 1)$$

В данном случае из примера  $m = d_k$ .

Переводя на русский, в статье (7) говорится:

“При больших значениях  $d_k$  произведения увеличиваются по величине, выталкивая функцию *softmax* в область, где она имеет чрезвычайно малые градиенты.”

Масштабирование преследует цель нейтрализации этого эффекта.

4. Нормирование полученных значений путем применения *softmax* функции, чтобы получить веса элементов, которые учитывают их важность для формирования выходной последовательности.

**Пример:**

$$\tilde{A} = \text{Softmax}(A_{sq})$$

5. Умножение полученных весов на матрицу Value, для вычисления взвешенной суммы значений элементов входной последовательности и создания новой последовательности.

**Пример:**

$$Z = \tilde{A} \times V$$

Всё это в общем можно описать одной формулой:

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

Этот процесс можно повторять множество раз, образуя блоки self-attention, которые могут быть использованы для решения задач обработки текстов и естественного языка в данной архитектуре.

Возвращаясь к теме раздела, опишем Multi-Head Attention.

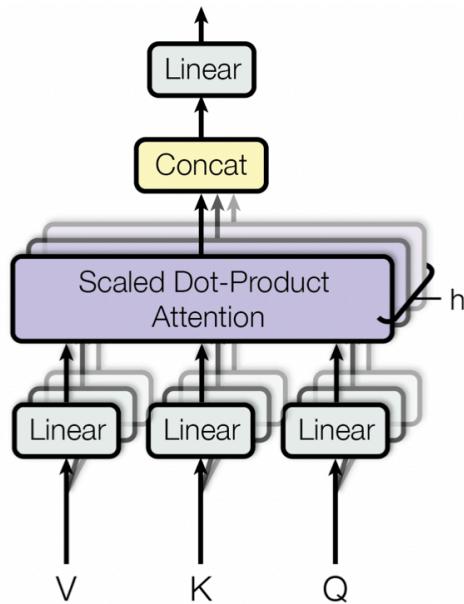


Рис. 13: Multi-Head Attention механизм.

Данный механизм имеет несколько “голов внимания”. Данные головы - self-attention, которые характеризуются своими весовыми матрицами  $W_i^Q, W_i^K, W_i^V$ , чтобы выявлять разные зависимости.

То есть в общих чертах, Multi-Head Attention производит несколько параллельных операций внимания на разных подпространствах признаков входных данных. Это позволяет сетям более эффективно использовать информацию, которая закодирована в этих подпространствах, для более точных предсказаний.

После всех операций блока self-attention (Scaled Dot-Product Attention) получается  $h$  векторов  $Z_i$ . После этого происходят следующие операции:

1. Конкатенация всех  $h$  векторов  $Z_i$  в вектор  $Z$ ;
2. Вектор  $Z$  умножается на матрицу  $W^O$ .

Матрица весов  $W^O$  обучается вместе с моделью;

3. Результирующий вектор  $Z$  после умножения на  $W^O$  проходит через FFNN.

## Add & Norm

На данном этапе:

1. Add - производится суммирование с вектором, полученным на предыдущем шаге, и с вектором, полученным 2 шага назад. Подробнее можно рассмотреть на Рис. 14.
2. Norm - каждый полученный вектор нормализуется.

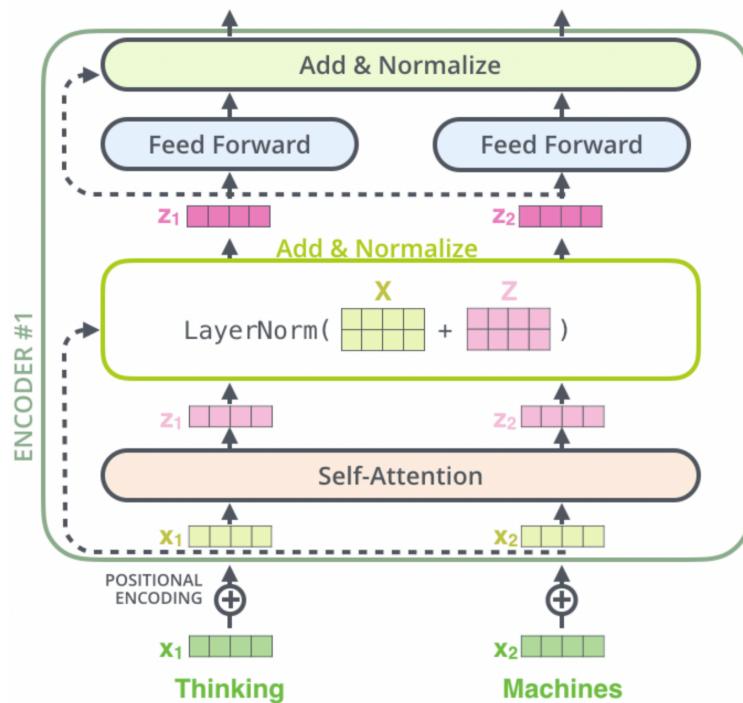


Рис. 14: Add & Norm.

## Feed Forward

В энкодере и декодере содержится FFN, которая применяется к каждой позиции отдельно и идентично. Он состоит из двух линейных преобразований с промежуточной активацией ReLU.

$$FFN(x) = \max(0, W_1 + b_1)W_2 + b_2$$

### 3.2.6 Кратко о декодере

#### Masked Multi-Head Attention

Декодер имеет практически те же компоненты, что и энкодер. Однако, здесь присутствует отличающийся один блок. Это Masked Multi-Head Attention.

Masked Multi-Head Attention - это слой в модели трансформера, который используется для вычисления важности порядка последовательности элементов в ключе.

Маскировка Multi-Head Attention заключается в том, что этот слой не обращается к информации от будущих входных слов. Это делается для того, чтобы модель могла генерировать последовательность поэлементно и учитывать только предыдущие слова. Для такого механизма используется маска, которая обозначает, какие элементы итерируемой последовательности включать в рассмотрение, а какие - нет.

Маскирование происходит путем установки значения  $-\infty$  перед операцией *Softmax* для тех значений, которые “видеть не стоит”.

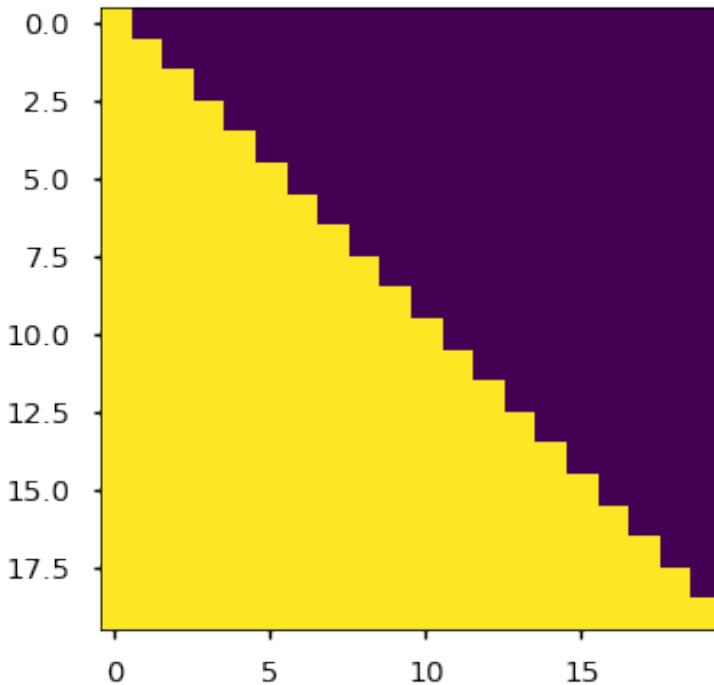


Рис. 15: Маска. Ось ординат - шаг, ось абсцисс - позиция в очереди

## **Соединение энкодера и декодера**

Как видно на Рис. 9 информация из энкодера и декодера “объединяется” в блоке Multi-Head Attention декодера. Выход из энкодера преобразуется в *Key* и *Value*, а выход из первого блока *Add&Norm* декодера преобразуется в *Query*. Результат преобразований идет дальше по декодеру.

## 4 BERT

BERT, или Bidirectional Encoder Representations from Transformers, представляет собой модель для предобработки естественного языка, которая использует трансформерную архитектуру. BERT разработан для выполнения двунаправленного обучения на больших корпусах текстовых данных и способен создавать контекстуальные представления слов с учетом их контекста.

Как и было сказано ранее, BERT основывается на предобучении на большом объеме текстовых данных, а затем саму модель можно дообучить на более узкоспециализированных задачах, таких как вопросно-ответные системы, распознавание именованных сущностей, классификация текста.

Данная модель имеет двунаправленное внимание и позволяет использовать информацию о всех токенах входной последовательности. Это особенно удобно для задач, где необходимо анализировать всю входную информацию целиком. Однако при этом BERT не обучается генерировать текст с нуля, а использует задачи, такие как *masked language modeling* и *next sentence prediction*, которые помогают предсказывать слова, основываясь на оставшихся и определять следуют ли текстовые фрагменты друг за другом.

Данная модель предлагает следующие нововведения:

1. Подавать дополнительный токен [CLS] на вход последовательности, который сосредотачивает на себе контекстную информацию всего предложения, которая в свою очередь может быть полезна при классификации.
2. Рандомное маскирование 15% от общего кол-ва токенов. Идея состоит в том, чтобы научить модель предсказывать не следующее слово, а пропущенное.

Стоит упомянуть, что предложения разделяются токеном [SEP].

BERT base будет использоваться в практике, поэтому будет затронута её архитектура, описанная в статье (8).

### 4.1 Входная последовательность

Входная последовательность в BERT образуется не так, как в Transformer.

Input	[CLS]	my	dog	is	cute	[SEP]	he	likes	play	# #ing	[SEP]
Token Embeddings	$E_{[CLS]}$	$E_{my}$	$E_{dog}$	$E_{is}$	$E_{cute}$	$E_{[SEP]}$	$E_{he}$	$E_{likes}$	$E_{play}$	$E_{\#\#ing}$	$E_{[SEP]}$
Segment Embeddings	$E_A$	$E_A$	$E_A$	$E_A$	$E_A$	$E_A$	$E_B$	$E_B$	$E_B$	$E_B$	$E_B$
Position Embeddings	$E_0$	$E_1$	$E_2$	$E_3$	$E_4$	$E_5$	$E_6$	$E_7$	$E_8$	$E_9$	$E_{10}$

Рис. 16: Входное представление BERT.

На вход модели подается либо одно, либо два предложения, разделенных [SEP].

Последовательность, которая пойдет в энкодер, преобразуется следующим образом:

### 1. Token Embeddings:

Токенизатор BERT - это процесс нарезки текста на токены - минимальные единицы, с которыми работает модель BERT. Он использует алгоритм WordPiece для разделения слов на наименьшие смысловые части и создания словаря токенов для эффективной обработки данных.

Процесс работы токенизера BERT включает следующие шаги:

- Обработка текста: удаление дополнительных пробелов, точек и других знаков пунктуации.
- Преобразование текста в токены: каждый текст преобразуется в набор токенов с помощью WordPiece токенизации.
- Добавление специальных токенов: добавление [CLS] и [SEP], которые указывают на начало и конец предложения, а также определяют, какие фрагменты текста находятся в одном предложении.
- Маскирование токенов: для задачи заполнения пробелов случайные токены заменяются на специальный токен [MASK] для обучения модели.

После токенизации каждое предложение конвертируется в числовой вектор.

2. Segment Embeddings: segment token в BERT - это специальный токен, который добавляется к входным данным в модель BERT для того, чтобы различать разные сегменты текста. Segment token применяется для обработки многосегментного текста, например в случае, когда два предложения передаются в модель как один объединенный текст. Первый сегмент отмечается одним типом segment token, а второй сегмент - другим. Обычно для идентификации сегментов используются целочисленные номера, которые могут быть присвоены каждому сегменту.

3. Position Embeddings

## 4.2 Архитектура

В системе BERT есть два этапа: pre-training и fine-tuning. Во время pre-training модель обучается на немаркированных данных в рамках различных задач предварительного обучения. Для fine-tuning BERT сначала инициализируется с pre-training параметрами, и все параметры настраиваются с использованием помеченных данных из последующих задач.

Отличительной особенностью BERT является его унифицированная архитектура для различных задач. Есть небольшая разница между pre-training архитектурой и конечной архитектурой последующего процесса.

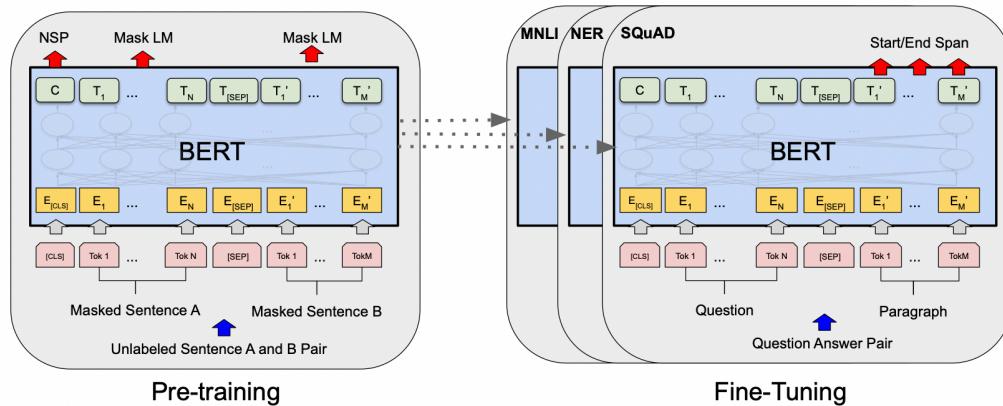


Рис. 17: Процедура предобучения и fine-tuning BERT.

BERT имеет оригинальный энкодер от Transformer. Следующие архитектуры имеют такие параметры со сравнением с Transformer:

Тип	Transformer	BERT base	BERT large
кол-во энкодеров	6	12	24
размер скрытого слоя	512	768	1024
кол-во голов self-attention	8	12	16
всего параметров	-	110M	340M

## 4.3 Pre-training

BERT предобучается на двух задачах:

### 1. Masked Language Model

- (a) На вход подаются 1-2 предложения.
- (b) Рандомно 15% токенов заменяются на маску.

Хотя это позволяет получить двунаправленную предварительно обученную модель, недостатком является то, что создается несоответствие между предварительным обучением и дообучением, поскольку токен [MASK] не появляется во время дообучения. Чтобы смягчить это, авторы статьи (8) не всегда заменяют “замаскированные” слова действительным токеном [MASK]. Генератор обучающих данных случайным образом выбирает 15% позиций токенов для прогнозирования. Если выбран  $i$ -й токен, то происходит замена  $i$ -го токена токеном [MASK] в 80% случаев, случайным токеном в 10% случаев и оставляют неизменным  $i$ -м токеном в 10% случаев.

- (c) Модель учится предсказывать, что находится под “маской”.
- (d) На выходе представлены все скрытые состояния токенов. Окончательные скрытые состояния, соответствующие токенам маски, передаются в выходной softmax по словарю.

### 2. Next Sentence Prediction

- (a) На вход подаются 2 предложения, разделенные [SEP]. Суть - предсказать последовательны ли они (то есть имеют ли общий смысл). Таким образом, предложения, которые подаются на вход, могут являться либо последовательными, либо могут не иметь общей связи.
- (b) Токен [CLS] несет в себе контекстную информацию, поэтому в конце берется скрытое состояние данного токена.
- (c) Данное состояние пропускается через линейный слой, который выдает вероятность связности двух предложений.

## 4.4 Fine-tuning

На данном этапе BERT дообучают на нужные задачи. Например, BERT умеет решать следующее:

1. Sentiment Analysis - помогает определить “настроение” у текста.
2. Question Answering - помогает ответить на поставленный вопрос.
3. Text Prediction - предсказывает текст при написании электронного письма.
4. Text Generation - может написать статью на любую тему, введя всего несколько предложений.
5. Summarization - может быстро резюмировать большой текст.
6. Polysemy resolution - может различать слова, имеющие несколько значений, на основе окружающего текста.

Для каждой задачи к BERTу просто подключаются входы и выходы, относящиеся к конкретной задаче. С этими данными выполняется fine-tuning всех параметров от начала до конца.

## **5 Анализ тональности твитов**

### **5.1 История использования нейронных сетей в задаче классификации сентимента**

История использования нейронных сетей в задаче классификации сентимента текста началась в конце 1990-х годов, когда исследователи стали использовать методы машинного обучения для анализа текстов. Одним из первых методов машинного обучения, который был применен для задачи классификации сентимента, был байесовский классификатор. Он был доступен и эффективен, но имел некоторые ограничения в точности предсказания, поэтому исследователи начали искать более мощные алгоритмы.

В начале 2000-х годов некоторые исследователи начали использовать нейронные сети для классификации сентимента текста. Одни из первых таких работ, где изучали анализ настроений, использовали сверточные нейронные сети для классификации сентимента твитов. В этой работе авторы использовали сверточные нейронные сети для анализа текстов, включая твиты, и добились довольно высокой точности предсказания.

В последующие годы исследователи продолжали исследовать применение нейронных сетей для классификации сентимента текста и разрабатывать новые архитектуры нейронных сетей. Например, в 2016 году была опубликована статья “Attention-Based Recurrent Neural Network Models for Joint Intent Detection and Slot Filling” (модели нейронных сетей с использованием внимания для совместной определения намерения и заполнения ячеек), в которой авторы использовали рекуррентные нейронные сети с механизмами внимания для классификации сентимента текста.

В настоящее время нейронные сети являются одним из наиболее эффективных методов для классификации сентимента и активно применяются в решении этой задачи в различных сферах, включая социальные сети, обзоры товаров и услуг, медицинскую диагностику и многое другое.

## 5.2 Задача

Была изучена теория по нейронным сетям, трансформерам и BERTу. Следующая задача - применить знания на практике.

Задача этого раздела состоит из 3х блоков:

1. Работа с датасетом.
2. Выбор метрик.
3. Обучение BERT.

## 5.3 Датасет

Как и было сказано ранее, поиск подходящего датасета - нелегкая работа. В первую очередь были проанализированы некоторые статьи, которые ссылались на два способа получения данных: Twitter API и готовый сет данных RuSentiTweet.

Однако Twitter API не допускал регистрацию новых пользователей из России, что ещё больше сложнило задачу. Данный вариант пришлось отсесть.

RuSentiTweet являлся решением проблемы отсутствия датасета.

### 5.3.1 RuSentiTweet

RuSentiTweet - набор данных анализа настроений состоящий из 13 392 твитов на русском языке, который был собран в рамках статьи “RuSentiTweet: A Sentiment Analysis Dataset of General Domain Tweets in Russian”. RuSentiTweet был вручную аннотирован с использованием руководящих принципов RuSentiment на 5 классов: Positive, Neutral, Negative, Speech Act и Skip.

Однако в разработку сначала было взято 2 класса - positive и negative (бинарная классификация), а затем три класса - neutral, positive, negative (много-классовая классификация).

Датасет состоял из двух файлов:

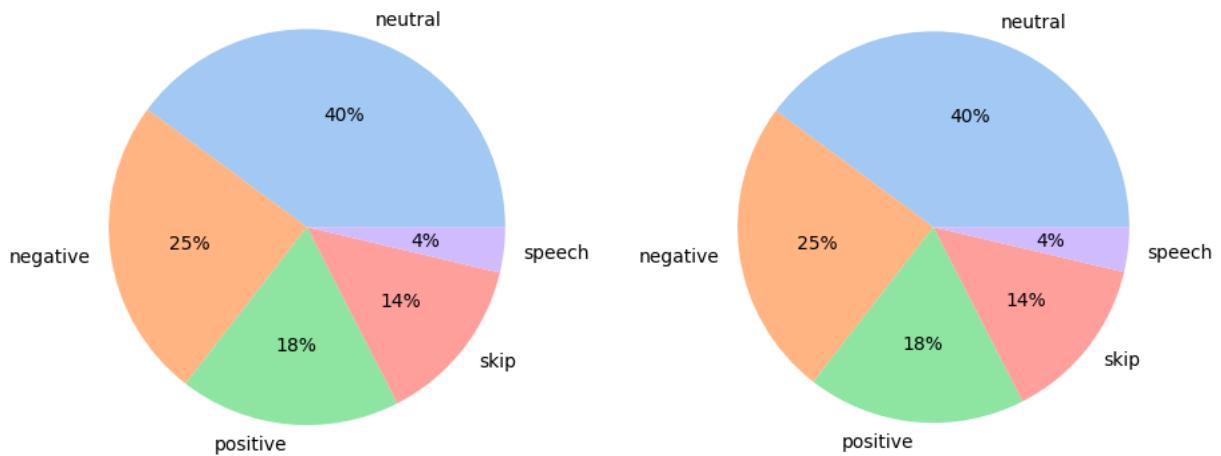


Рис. 18: Распределение классов в train. Рис. 19: Распределение классов в test.

При первой попытке построить модель, как и говорилось выше, были взяты два класса. После чего, некоторые данные вручную были перемаркированы, потому что имели двойное значение. Перемаркировка имела цель сбалансировать классы, потому что класс negative превалировал над классом positive. Итогом вышло следующее распределение:

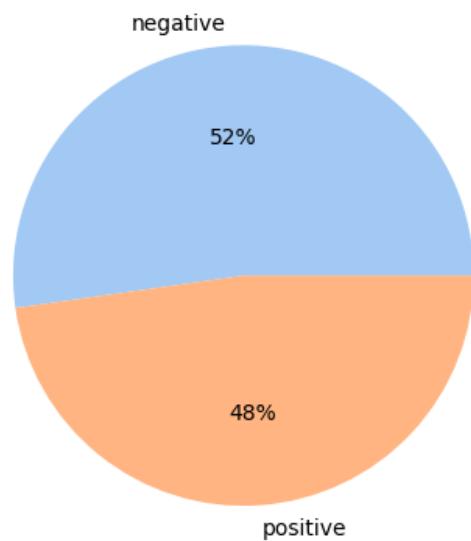


Рис. 20: RuSentiTweet - binary.

Классы получились почти сбалансированными. Из файла test были взяты только два класса соответственно. Train был поделен на обучающую и валидационную выборки в соотношении 7:3. Распределение классов в нем было сле-

дующее:

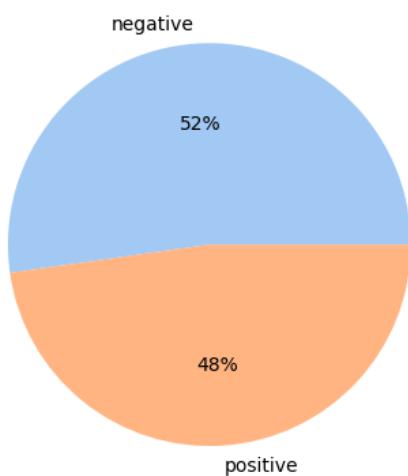


Рис. 21: Распределение классов в  $X_{train}$ .

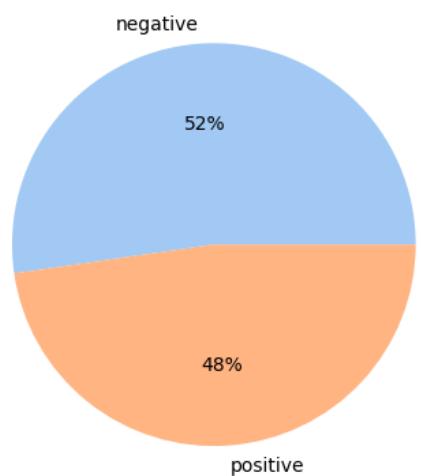


Рис. 22: Распределение классов в  $X_{dev}$ .

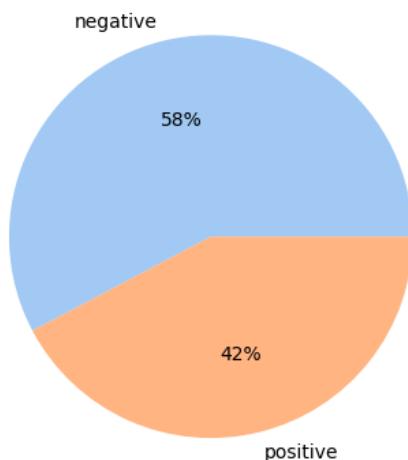


Рис. 23: Распределение классов в  $X_{test}$ .

### 5.3.2 Tweets Dataset

Позже был обнаружен ещё один датасет с твитами с яркой эмоциональной окраской. Если в RuSentiTweet твиты были более повседневными, то твиты из Tweets Dataset были более эмоционально интенсивными.

Данный датасет состоял из двух файлов - train и test. Однако, по причине того, что test имел всего 50 твитов, было решено объединить оба файла в один и затем распределить твиты нужным образом на train, val и test.

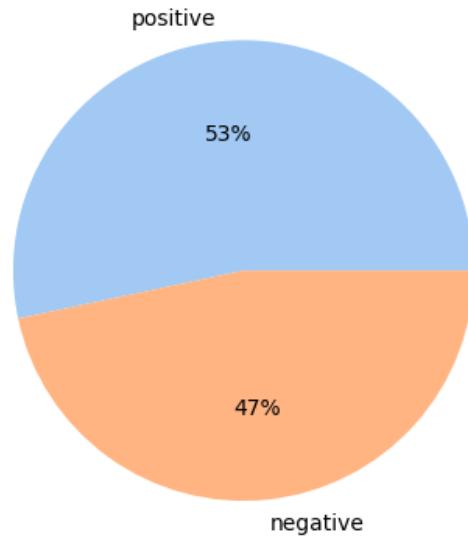


Рис. 24: Распределение в Tweets Dataset.

Было решено тоже взять данный датасет на обработку и посмотреть, какой из них даст лучшее решение.

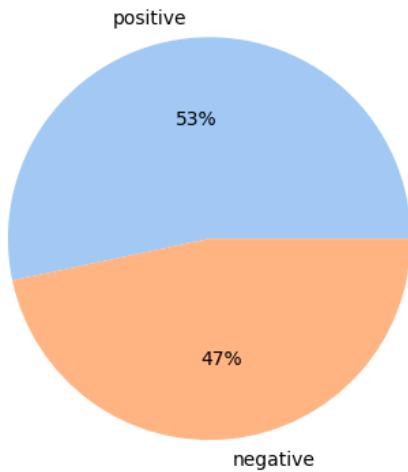


Рис. 25: Распределение классов в  $X_{train}$ .

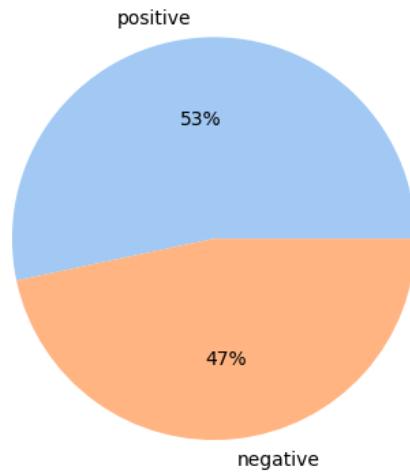


Рис. 26: Распределение классов в  $X_{dev}$ .

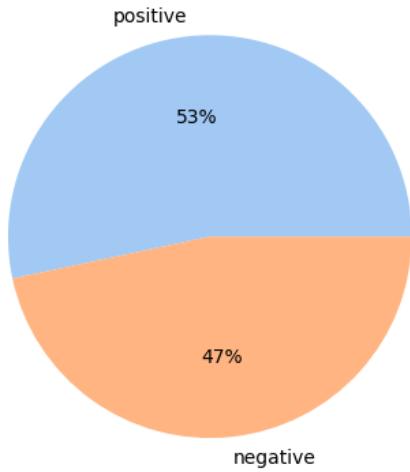


Рис. 27: Распределение классов в X\_test.

### 5.3.3 Объединение

Ранее говорилось о многоклассовой классификации. После обучения четырех моделей пришла мысль о том, что нельзя терять класс neutral. Ведь действительно, нельзя все делить на positive и negative, есть спорные моменты.

Поэтому было решено за основу взять три класса из RuSentiTweet - neutral, positive и negative. Класс neutral сильно перевешивал, как показано на Рис.18 и Рис.19, поэтому пришла идея дополнить меньшие по количеству твитов классы, взяв твиты из Tweets Dataset. Так как в Tweets Dataset эмоциональная окраска ярко выражена, то конфликтов на почве разметки не возникло.

Сэмплы из Tweets Dataset были добавлены рандомно в RuSentiTweet.

Распределение классов в объединенном датасете указано в Рис.28. Аналогичное распределение получается, когда данная выборка делится на train, val и test.

Теперь целью становится создание обобщающей модели<sup>1</sup>.

---

<sup>1</sup>Ссылка на датасеты и код работы: [https://github.com/Laitielly/term\\_paper3](https://github.com/Laitielly/term_paper3)

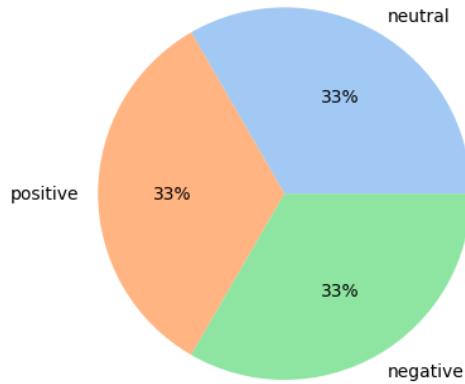


Рис. 28: Распределение классов в объединенном датасете.

## 5.4 Метрики

В качестве метрик были выбраны:

1. balanced accuracy & unweighted average recall & macro recall:

$$\frac{\sum_{i=1}^{\text{total classes}} \text{recall}_i}{\text{total classes}}$$

2. accuracy:

$$\frac{\text{correct predictions}}{\text{total cases}}$$

3. f1-macro:

$$\frac{\sum_{i=1}^{\text{total classes}} F1_i}{\text{total classes}}$$

Изначально датасеты были несбалансированны, поэтому была выбрана метрика balanced accuracy, так как она учитывает важность балансировки и не дает переобучиться модели.

F1-macro был выбран по той же логике.

Accuracy был выбран как часто используемая метрика.

## 5.5 Fine-Tuning

### 5.5.1 RuSentiTweet - binary

За основу был взят многоязычный BERT - предварительно обученная модель на задачу masked language modeling на 104 лучших языках с крупнейшей Википедией. На первых этапах предобработка не производилась, чтобы понять, как работает модель.

#### Обучение первое. Pooler\_output

Переписан классификатор, берется pooler\_output:

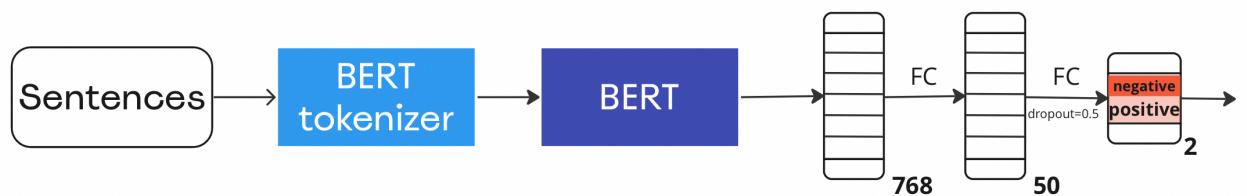


Рис. 29: Архитектура.

Результаты:

Оценка	train	val	test	test Tweets Dataset
balanced_accuracy_score	0.98	0.803	0.792	0.775
accuracy_score	0.979	0.802	0.79	0.775
f1-score	0.979	0.802	0.791	0.775

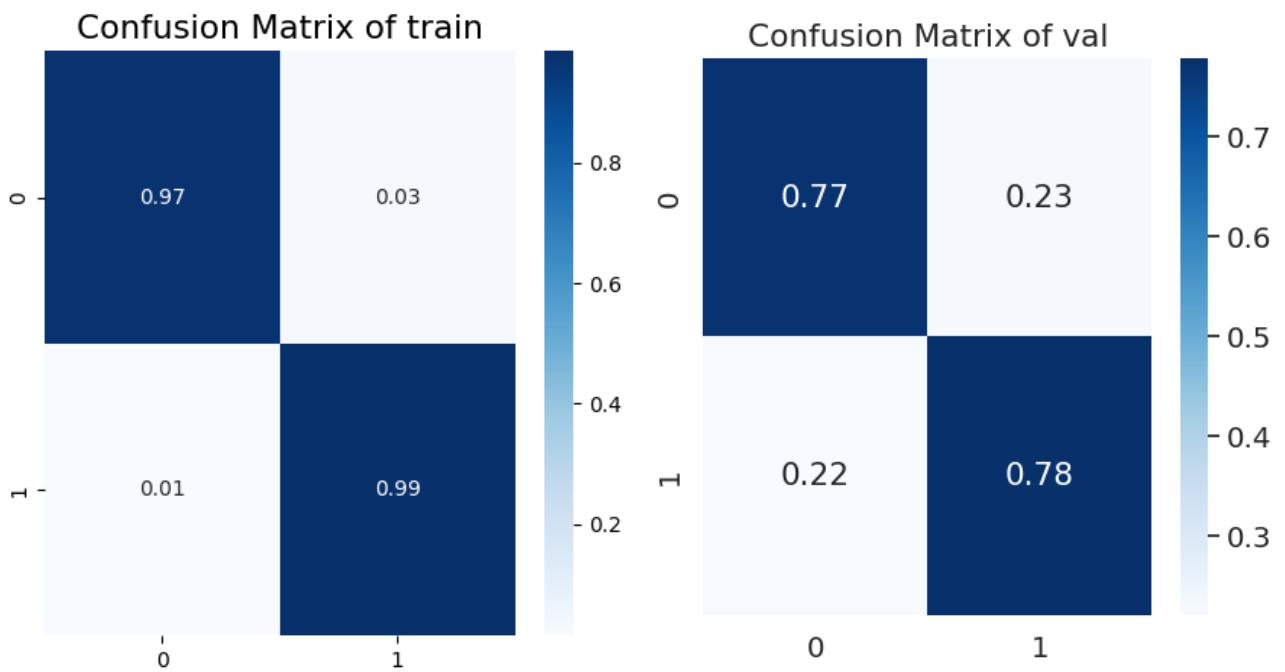


Рис. 30: Результаты в  $X_{train}$ .

Рис. 31: Результаты в  $X_{dev}$ .

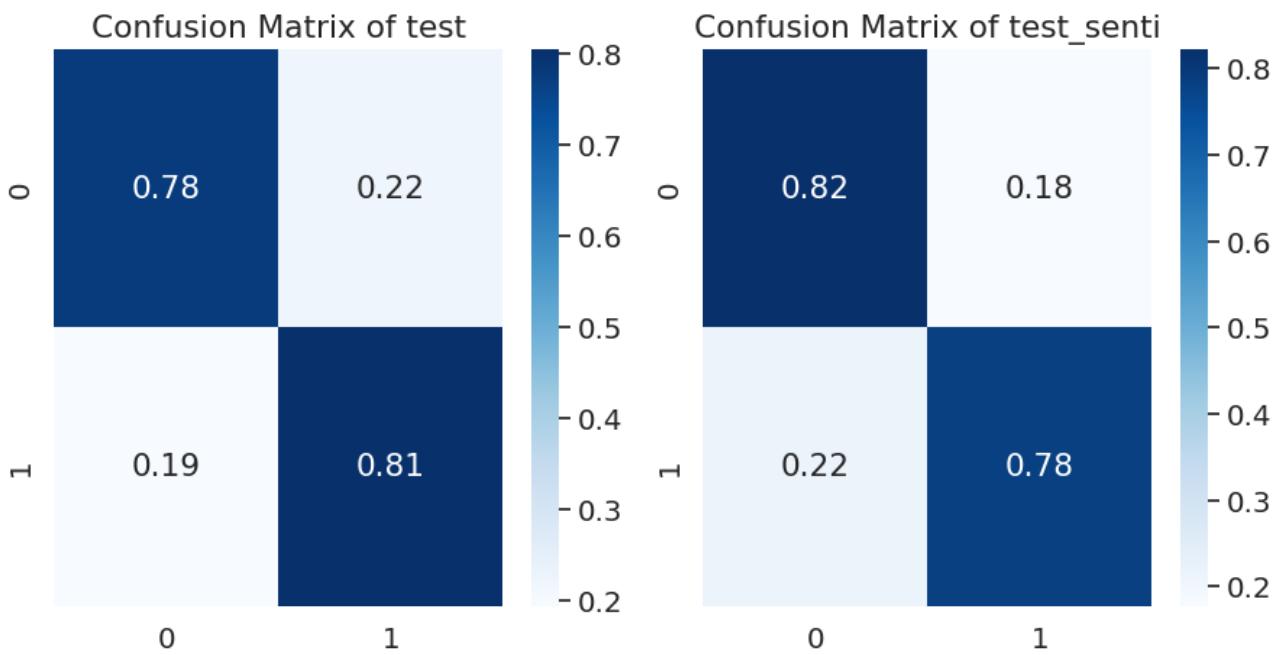


Рис. 32: Результаты в  $X_{test}$ .

Рис. 33: Результаты в  $X_{tweets}$ .

### Обучение второе. Pre-processing + Pooler\_output

Архитектура - аналогично предыдущему пункту, но делается pre-processing.

Для emoticons был создан словарь с возможными смайликами (из символов) и каждому ключу-смайлу был предоставлен “текстовый” перевод на русском.

Результаты:

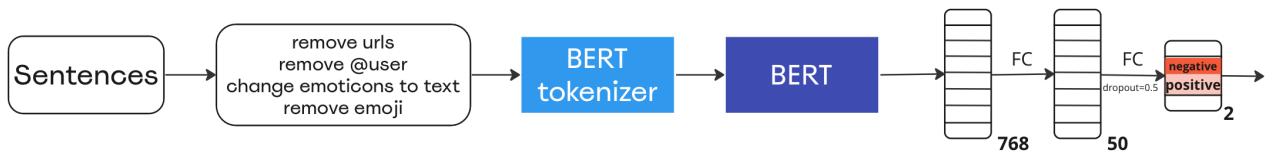


Рис. 34: Архитектура.

Оценка	train	val	test	test Tweets Dataset
balanced_accuracy_score	0.8425	0.74	0.751	0.832
accuracy_score	0.842	0.7399	0.748	0.832
f1-score	0.842	0.74	0.7495	0.832

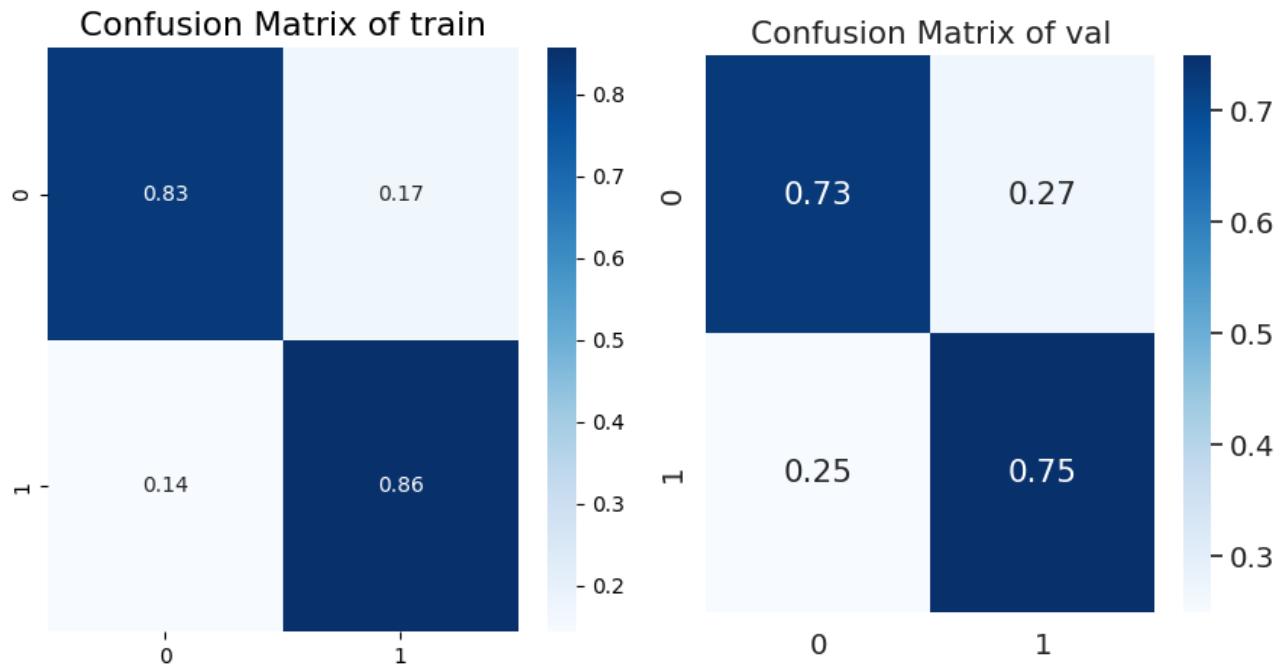


Рис. 35: Результаты в  $X_{train}$ .

Рис. 36: Результаты в  $X_{dev}$ .

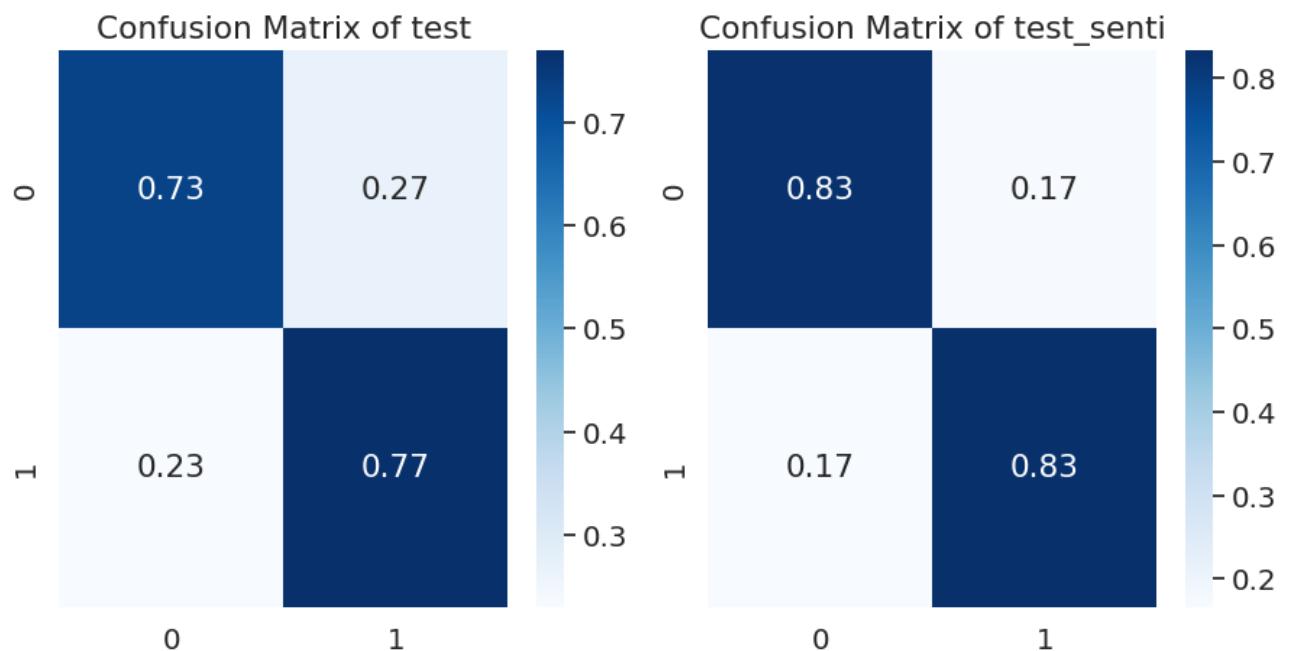


Рис. 37: Результаты в  $X_{test}$ .

Рис. 38: Результаты в  $X_{tweets}$ .

### 5.5.2 Tweets Dataset - binary

#### Обучение третье. Last\_hidden\_state\_cls

Архитектура - аналогично пункту первого обучения, но берется last\_hidden\_state\_cls.

Результаты:

Оценка	train	val	test	test RuSentiTweet
balanced_accuracy_score	0.9998	0.9994	0.9995	0.547
accuracy_score	0.9998	0.9994	0.9996	0.612
f1-score	0.9998	0.9994	0.9996	0.518

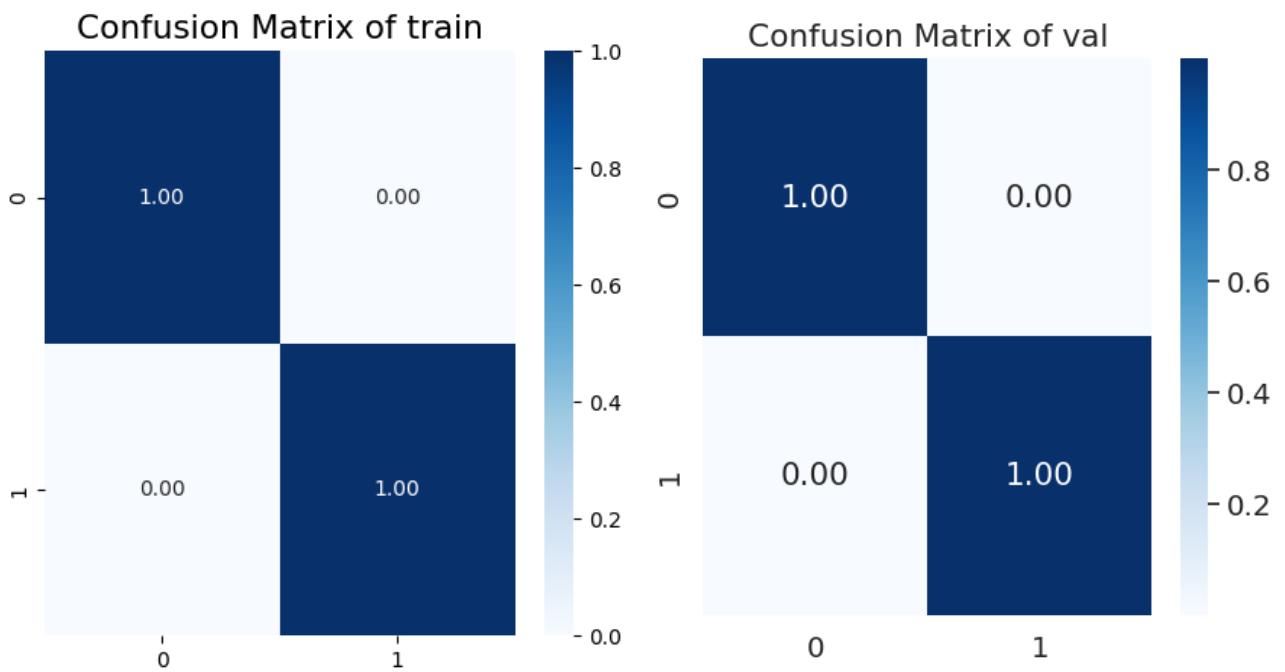


Рис. 39: Результаты в  $X_{train}$ .

Рис. 40: Результаты в  $X_{dev}$ .

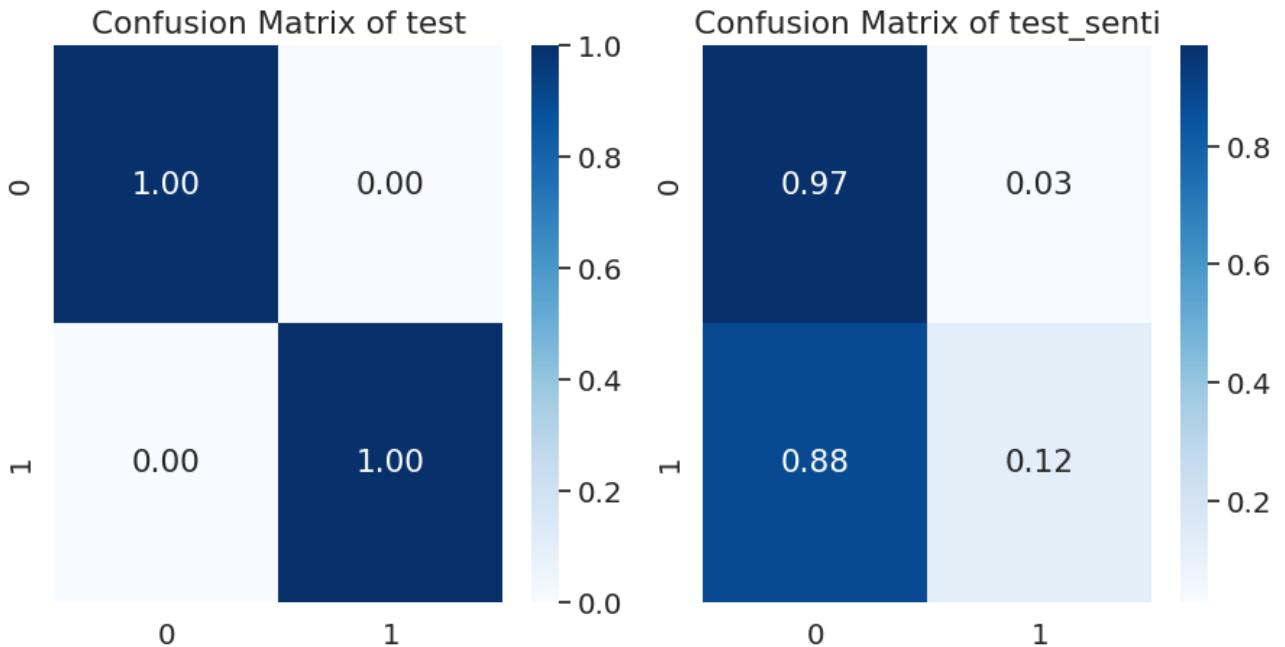


Рис. 41: Результаты в  $X_{test}$ .

Рис. 42: Результаты в  $X_{tweets}$ .

## Обучение четвертое. Pooler\_output

Архитектура - аналогично предыдущему пункту, но берется pooler\_output.

Результаты:

Оценка	train	val	test	test RuSentiTweet
balanced_accuracy_score	0.999	0.998	0.998	0.52
accuracy_score	0.999	0.998	0.998	0.503
f1-score	0.999	0.998	0.998	0.5

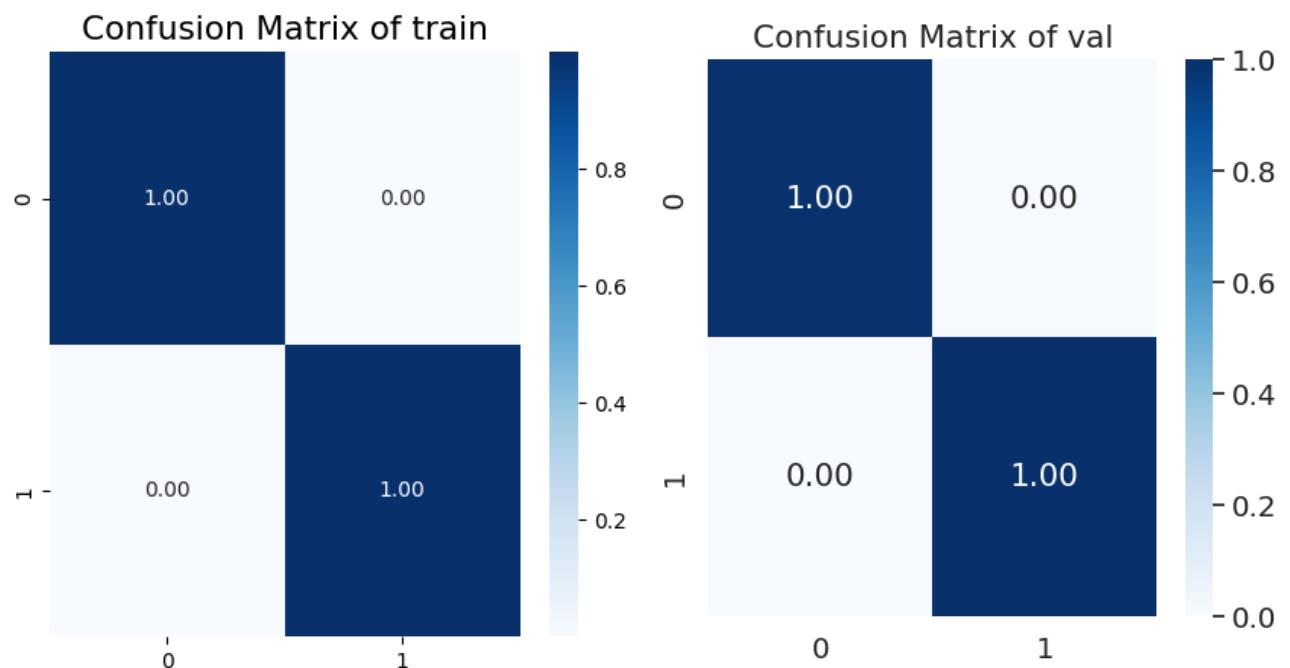


Рис. 43: Результаты в  $X_{train}$ .

Рис. 44: Результаты в  $X_{dev}$ .

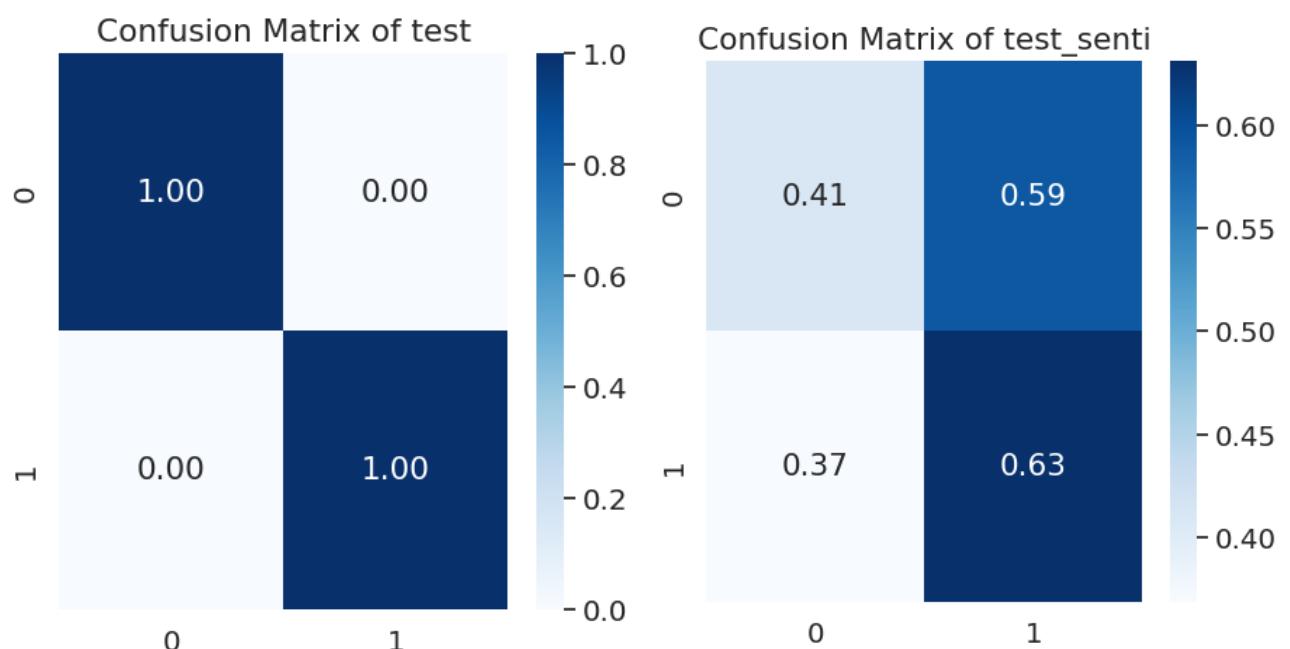


Рис. 45: Результаты в  $X_{test}$ .

Рис. 46: Результаты в  $X_{tweets}$ .

### 5.5.3 Многоклассовая классификация

#### Обучение пятое

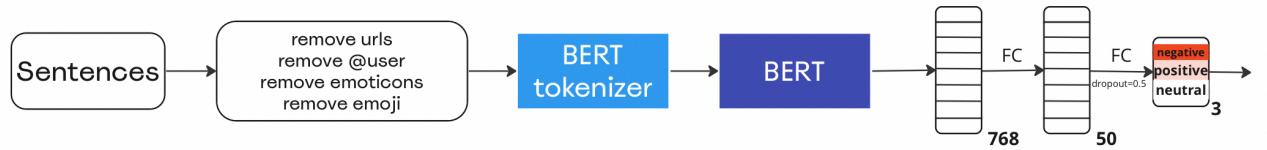


Рис. 47: Архитектура.

Результаты:

Оценка	train	val	test
balanced_accuracy_score	0.996	0.706	0.708
accuracy_score	0.996	0.706	0.708
f1-macro	0.996	0.707	0.708

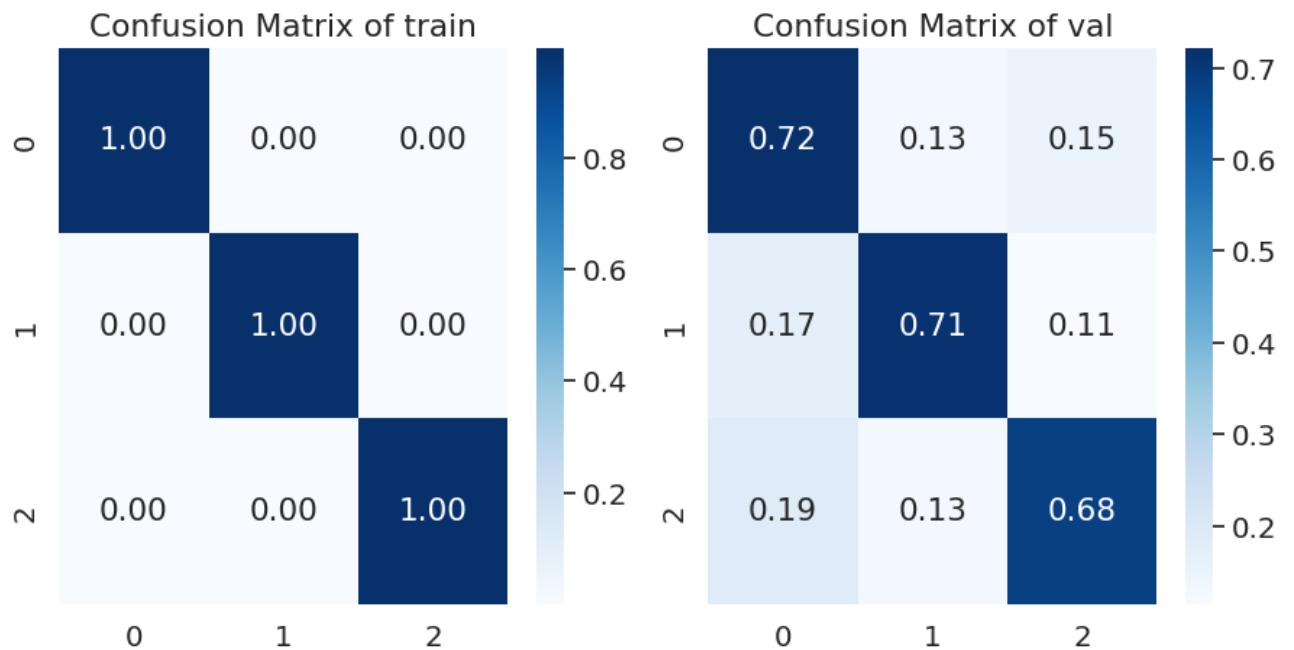


Рис. 48: Результаты в  $X_{train}$ .

Рис. 49: Результаты в  $X_{dev}$ .

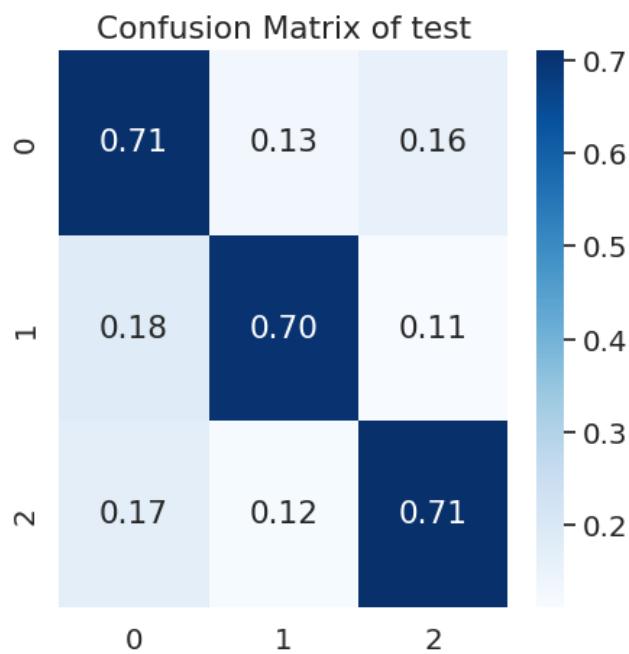


Рис. 50: Результаты в  $X_{test}$ .

## Обучение шестое

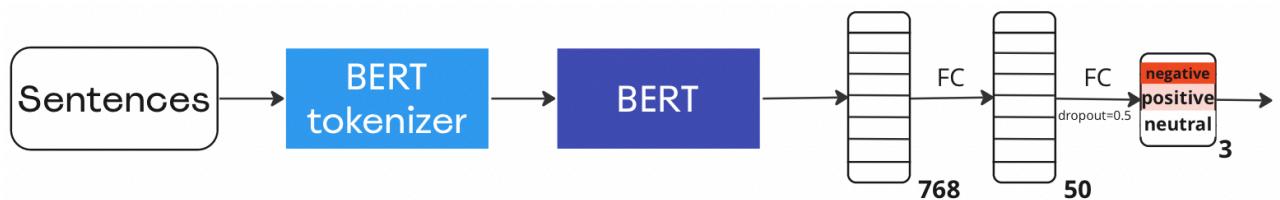


Рис. 51: Архитектура.

Результаты:

Оценка	train	val	test
balanced_accuracy_score	0.921	0.766	0.775
accuracy_score	0.921	0.766	0.775
f1-macro	0.921	0.766	0.775

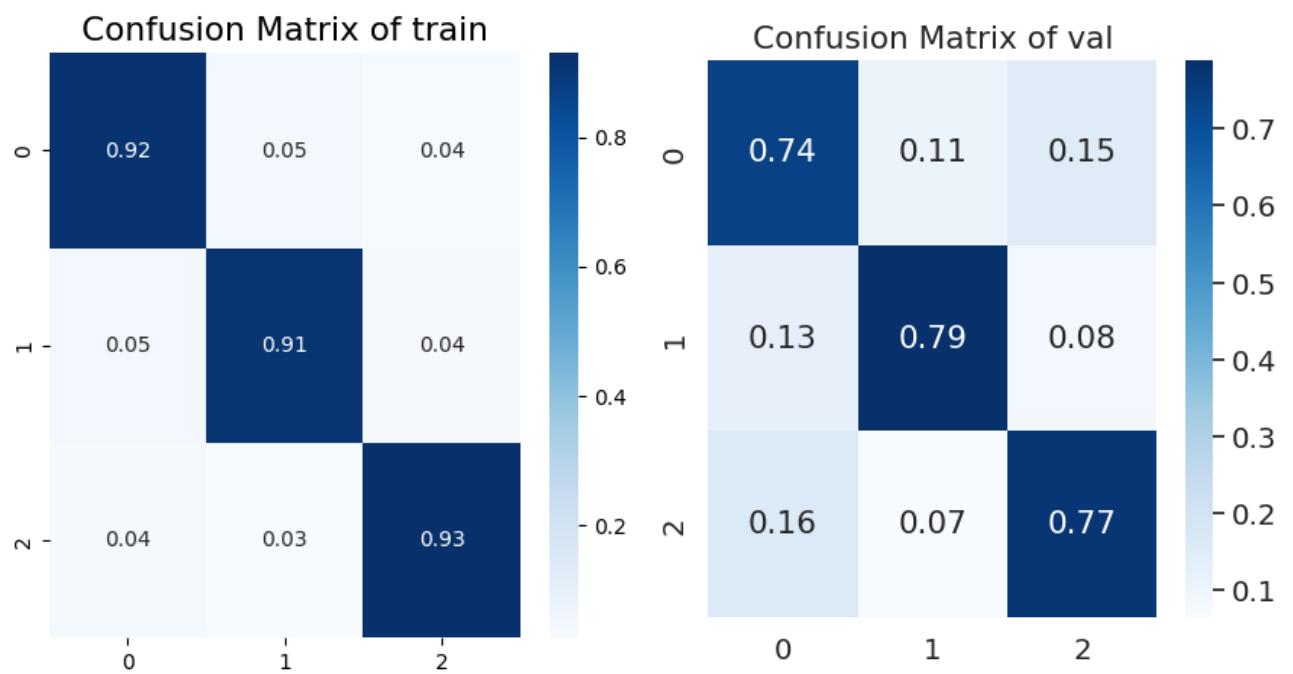


Рис. 52: Результаты в  $X_{train}$ .

Рис. 53: Результаты в  $X_{dev}$ .

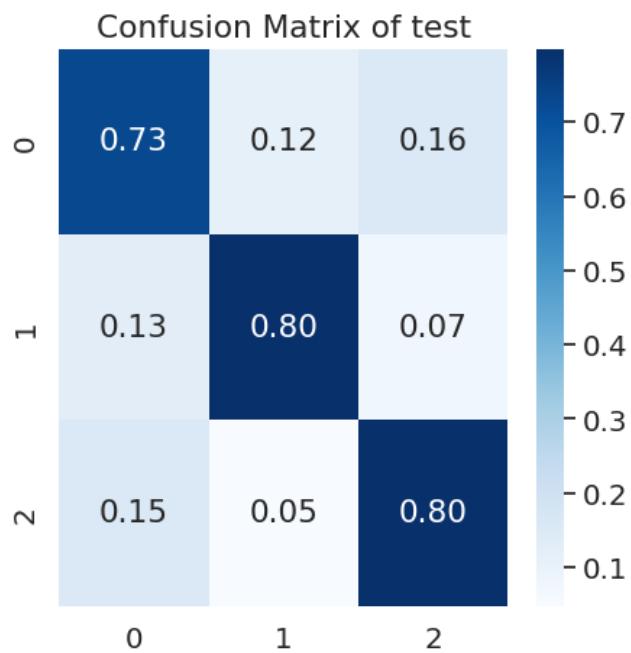


Рис. 54: Результаты в  $X_{test}$ .

## Обучение седьмое

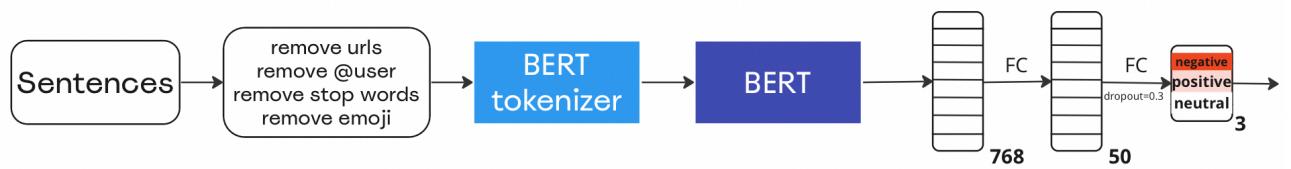


Рис. 55: Архитектура.

Результаты:

Оценка	train	val	test
balanced_accuracy_score	0.948	0.766	0.774
accuracy_score	0.948	0.766	0.774
f1-macro	0.948	0.766	0.774

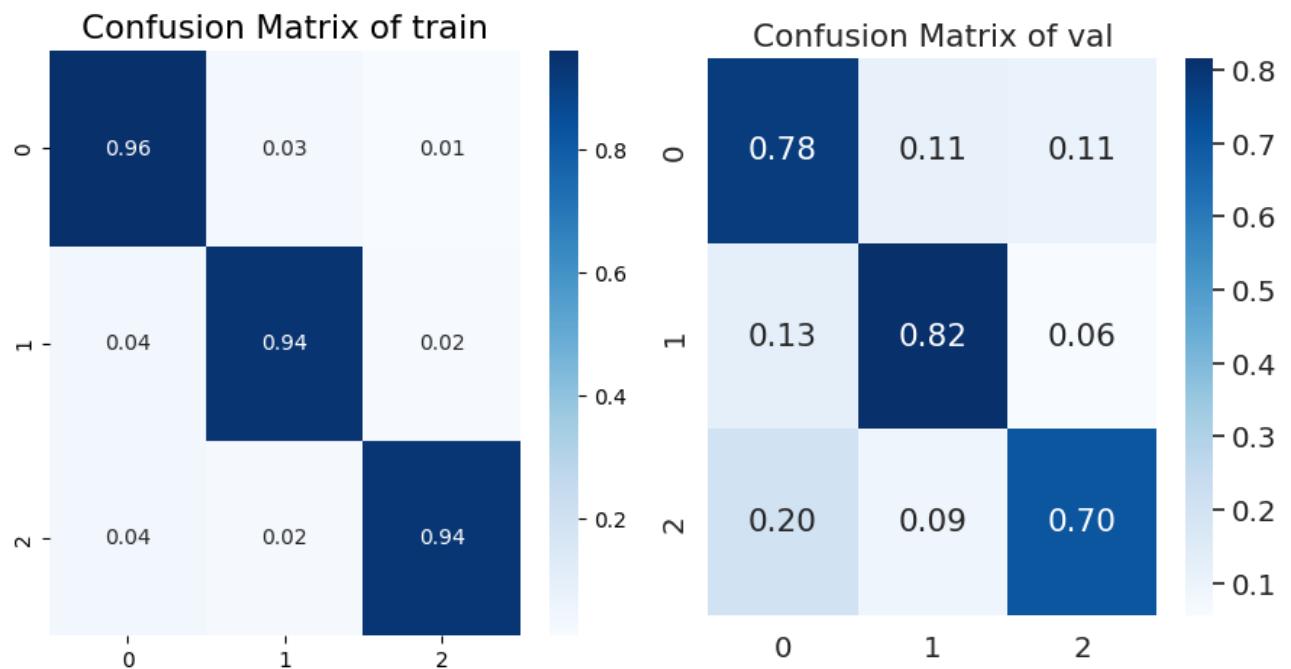


Рис. 56: Результаты в  $X_{train}$ .

Рис. 57: Результаты в  $X_{dev}$ .

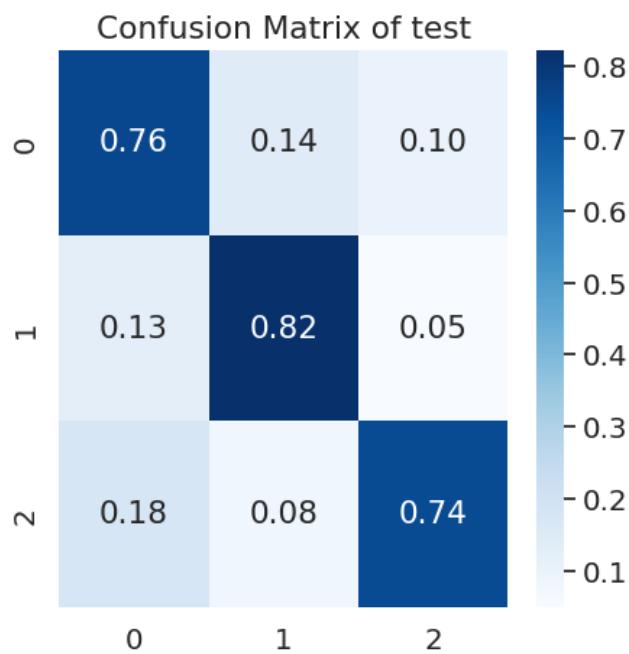


Рис. 58: Результаты в  $X_{test}$ .

## Обучение восьмое

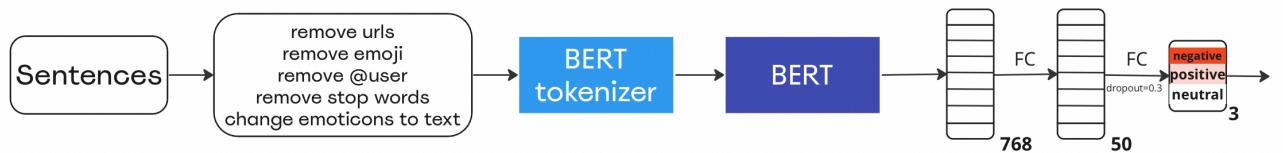


Рис. 59: Архитектура.

Результаты:

Оценка	train	val	test
balanced_accuracy_score	0.905	0.775	0.759
accuracy_score	0.905	0.775	0.759
f1-macro	0.906	0.776	0.76

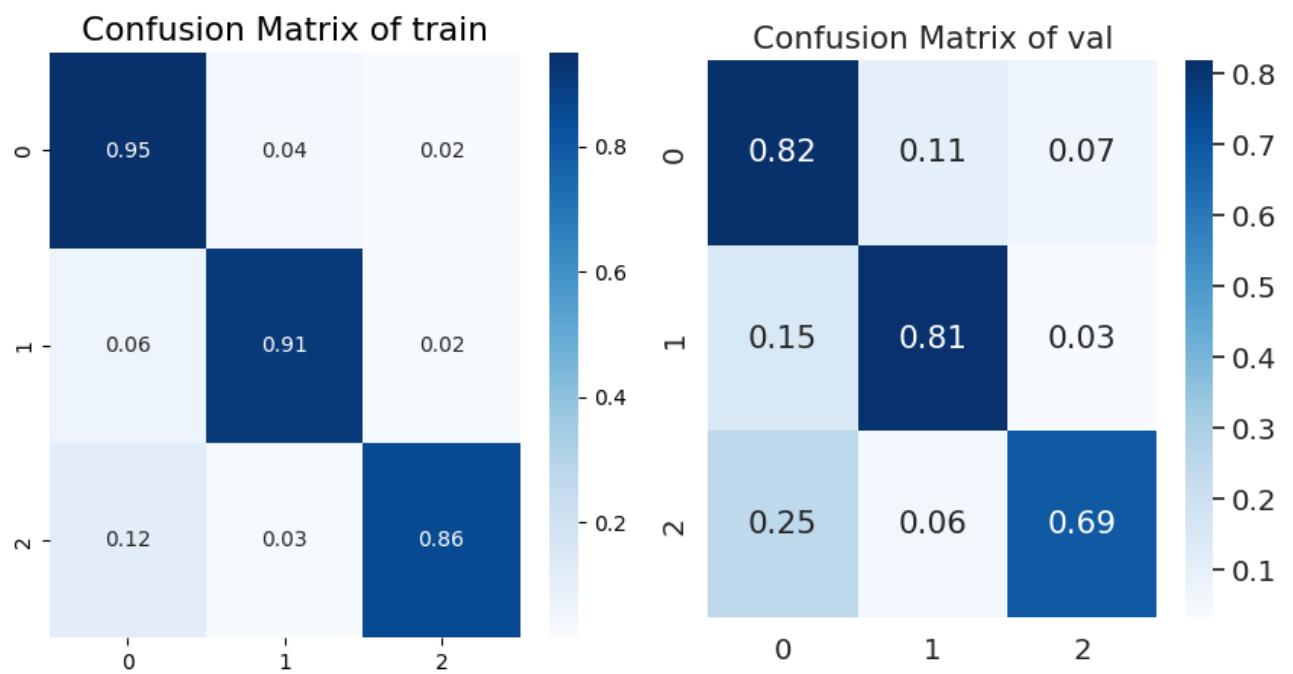


Рис. 60: Результаты в  $X_{train}$ .

Рис. 61: Результаты в  $X_{dev}$ .

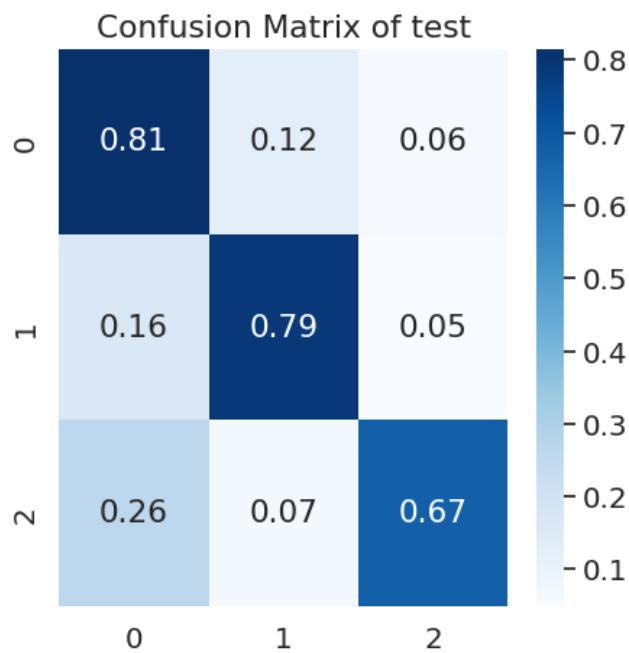


Рис. 62: Результаты в  $X_{test}$ .

## Обучение девятое

На данном этапе было принято решение попробовать немного другую модель - RoBERTa.

RoBERTa (Robustly Optimized BERT Approach) - это улучшенная версия модели BERT, разработанной Facebook AI Research. RoBERTa была представлена

в 2019 году и создавалась с целью улучшения работы BERT.

Основные отличия RoBERTы от BERTа:

1. Размер модели: RoBERTа имеет большую размерность чем BERT. 355 миллионов параметров на 110 миллионов соответственно.
2. Предобучение: При предварительном обучении RoBERTы использовалась дополнительная обработка текста, такая как удаление случайных предложений или изменение порядка слов в предложении. Это позволило улучшить качество модели.
3. Метод обучения: В отличие от BERT, где векторное представление последовательности слов строится путем объединения контекстов, при обучении RoBERTа используется метод маскирования. В результате чего модель более точно определяет значимость слов.
4. Декодирование: RoBERTа использует другой алгоритм декодирования, который позволяет усреднять несколько векторных представлений для одного и того же слова в разных контекстах, что повышает точность предсказания.
5. Дополнительные данные: Для обучения RoBERTы использовались большие наборы данных различных языков, что позволило сделать модель более устойчивой к языкам и улучшило ее качество.

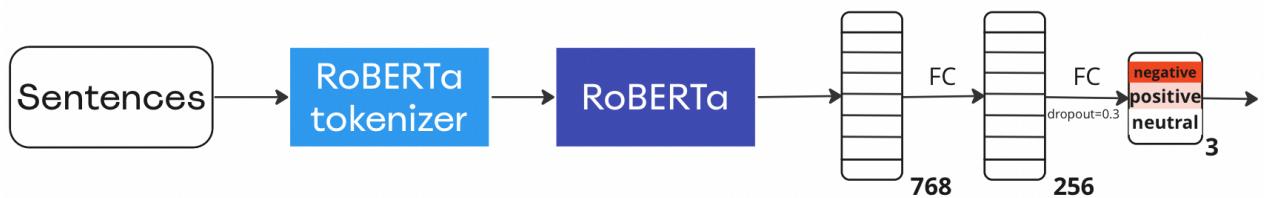


Рис. 63: Архитектура.

Результаты:

Оценка	train	val	test
balanced_accuracy_score	0.9053	0.725	0.73323
accuracy_score	0.9053	0.725	0.73323
f1-macro	0.906	0.727	0.735

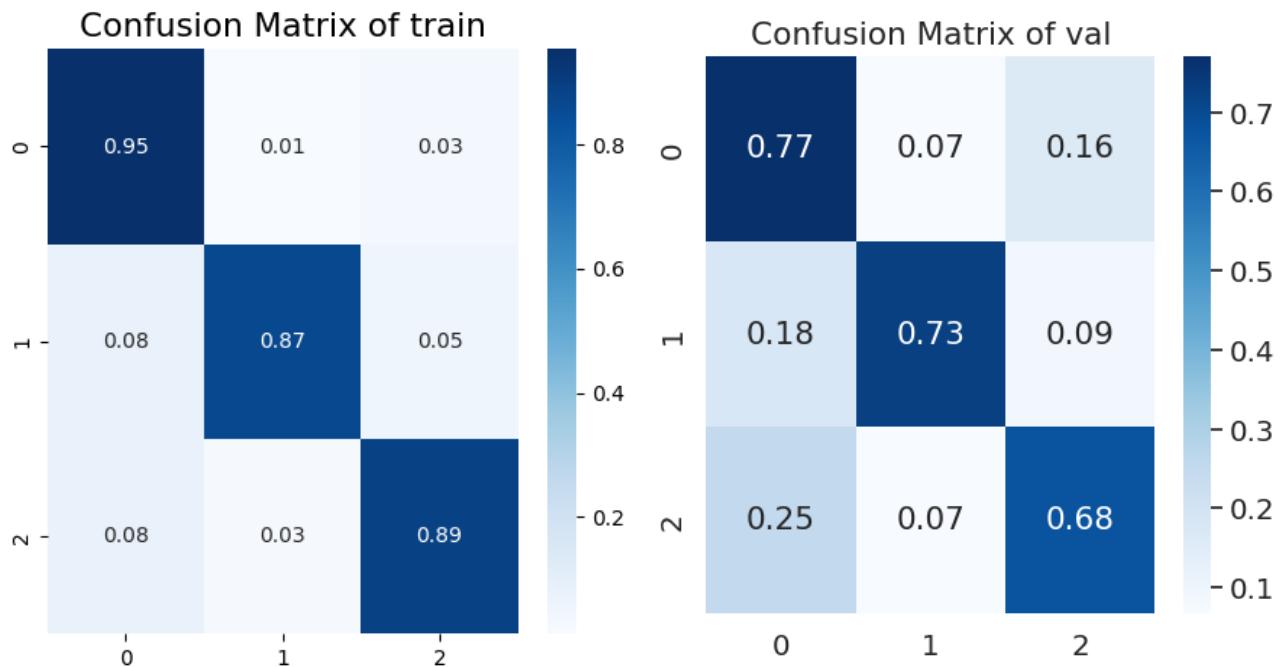


Рис. 64: Результаты в  $X_{train}$ .

Рис. 65: Результаты в  $X_{dev}$ .

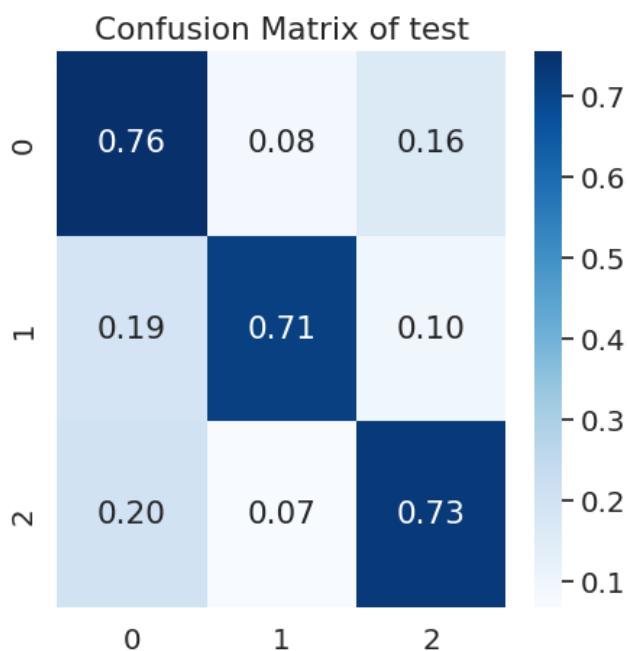


Рис. 66: Результаты в  $X_{test}$ .

## Обучение десятое

Здесь и далее датасет был перемешан между собой еще раз, чтобы сравнить результаты. Ведь могла попасть неудачная выборка.

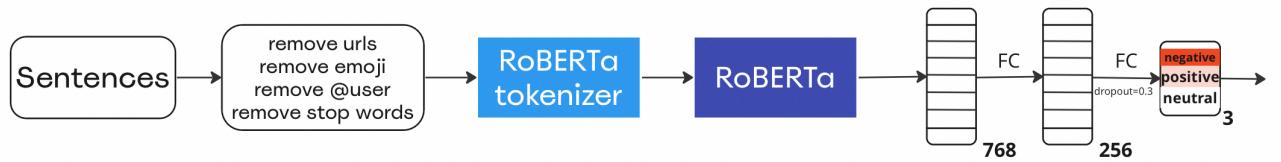


Рис. 67: Архитектура.

Результаты:

Оценка	train	val	test
balanced_accuracy_score	0.981	0.731	0.728
accuracy_score	0.981	0.731	0.723
f1-macro	0.981	0.731	0.723

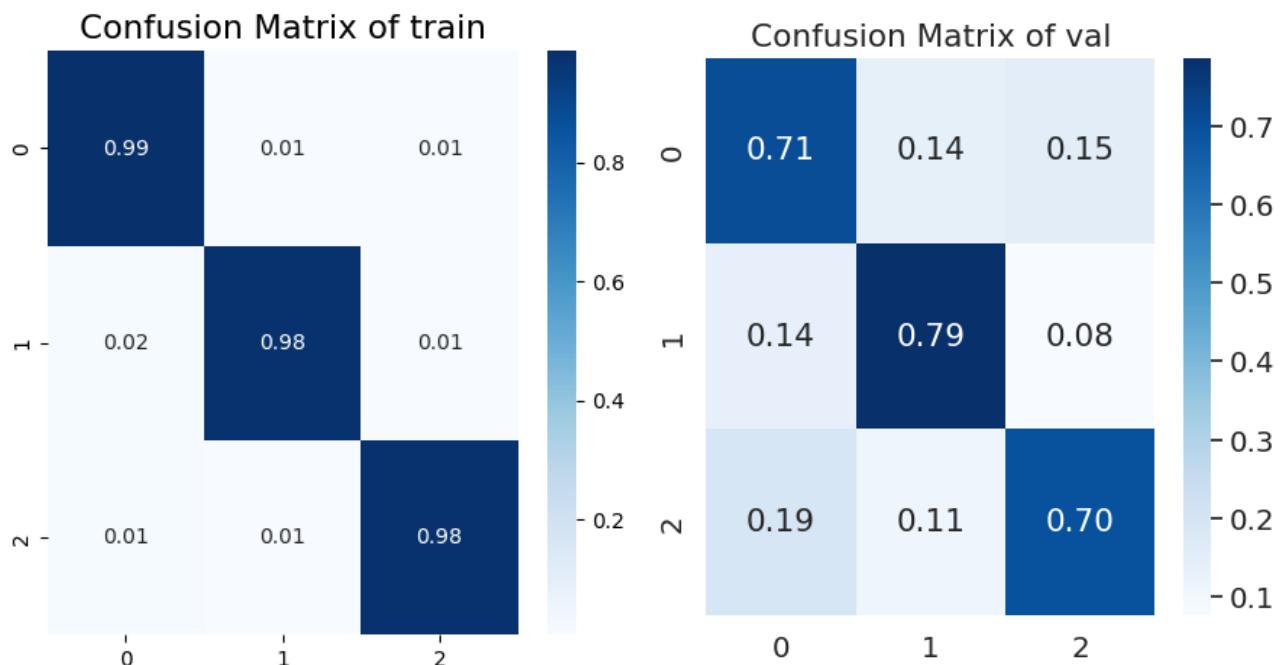


Рис. 68: Результаты в  $X_{train}$ .

Рис. 69: Результаты в  $X_{dev}$ .

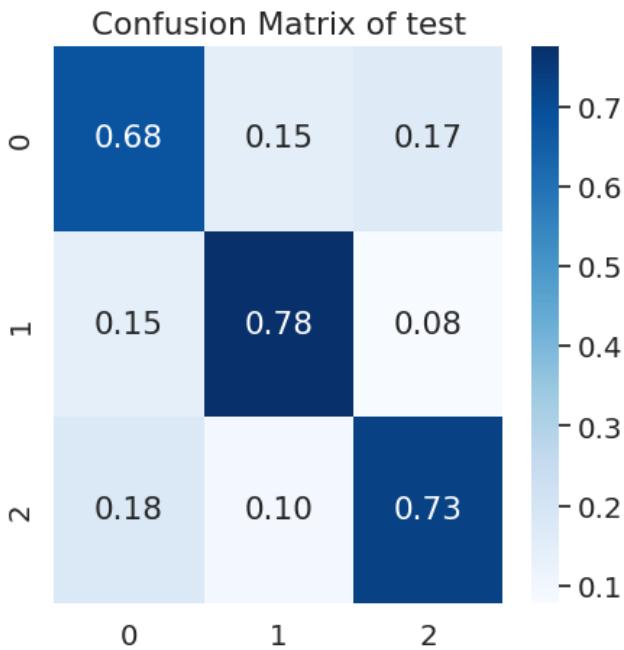


Рис. 70: Результаты в  $X_{test}$ .

## Обучение одиннадцатое

В данном обучении было принято решение взять еще одну архитектуру - RuBERT tiny.

Это представление небольшого русского кодировщика на базе BERT с высококачественным встраиванием предложений.

1. увеличенный словарный запас: 83828 лексем;
2. поддерживаемые последовательности большего размера: 2048 вместо 512;
3. значимые вложения сегментов (настроены в соответствии с задачей NLI)
4. модель ориентирована только на русский язык.

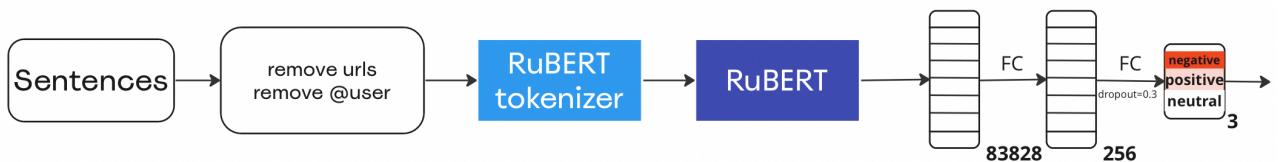


Рис. 71: Архитектура.

Результаты:

Оценка	train	val	test
balanced_accuracy_score	0.866	0.787	0.78
accuracy_score	0.866	0.787	0.78
f1-macro	0.867	0.788	0.782

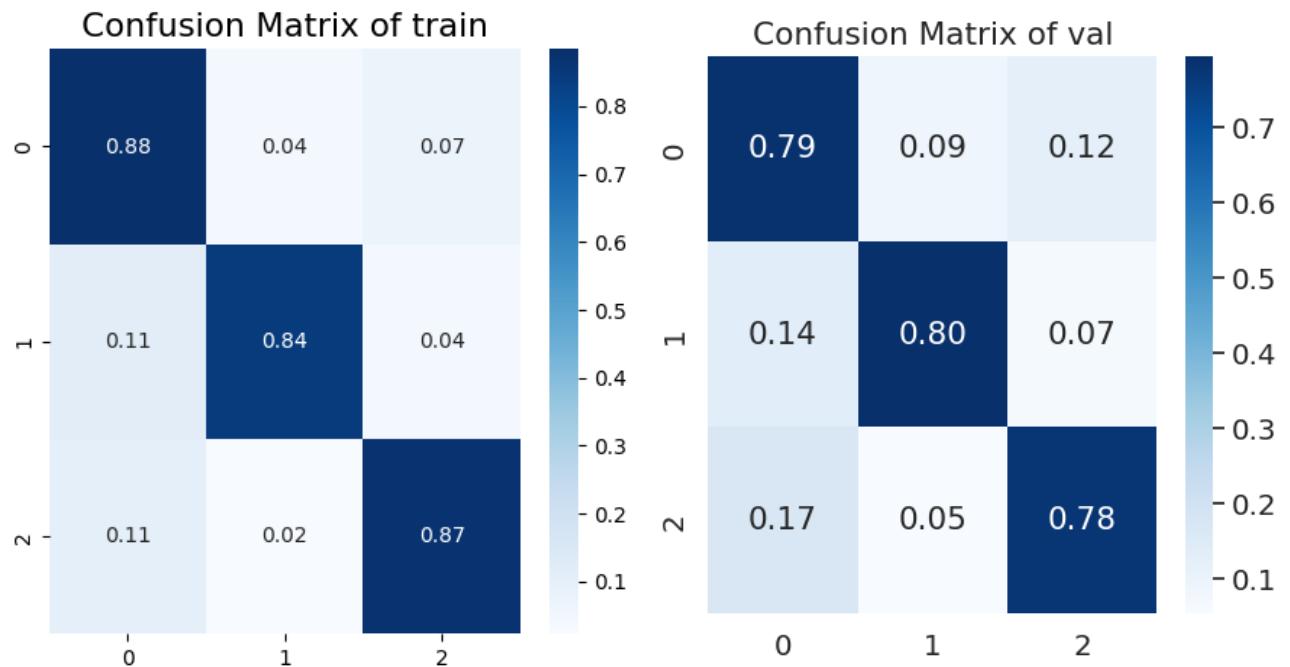


Рис. 72: Результаты в  $X_{train}$ .

Рис. 73: Результаты в  $X_{dev}$ .

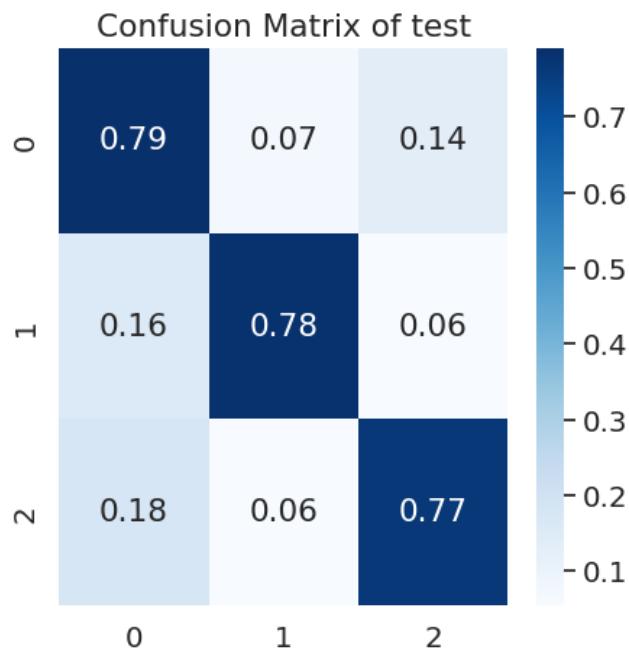


Рис. 74: Результаты в  $X_{test}$ .

## Обучение двенадцатое

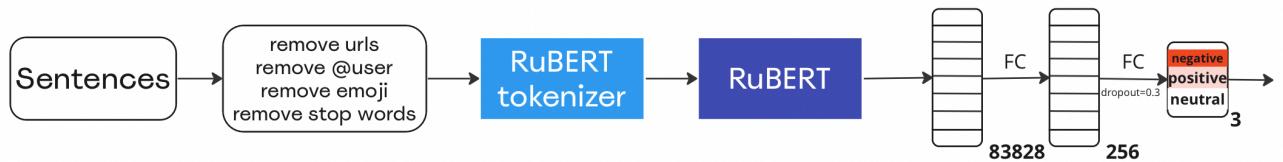


Рис. 75: Архитектура.

Результаты:

Оценка	train	val	test
balanced_accuracy_score	0.88	0.794	0.789
accuracy_score	0.88	0.794	0.789
f1-macro	0.88	0.794	0.789

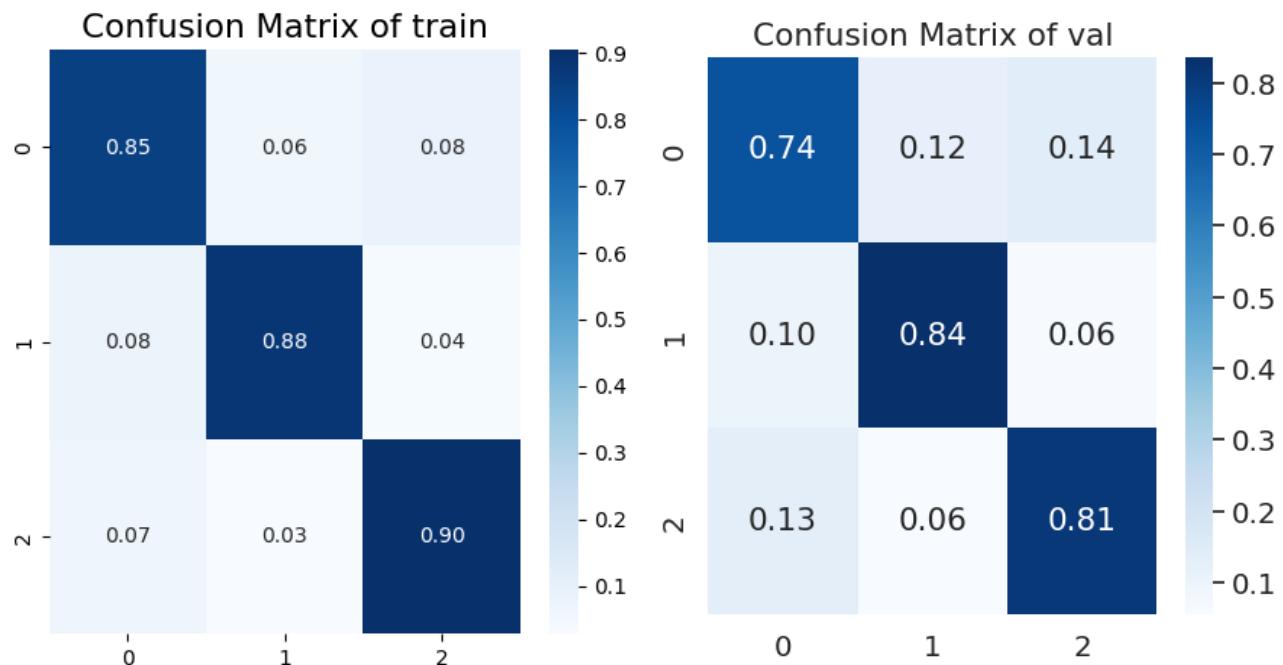


Рис. 76: Результаты в  $X_{train}$ .

Рис. 77: Результаты в  $X_{dev}$ .

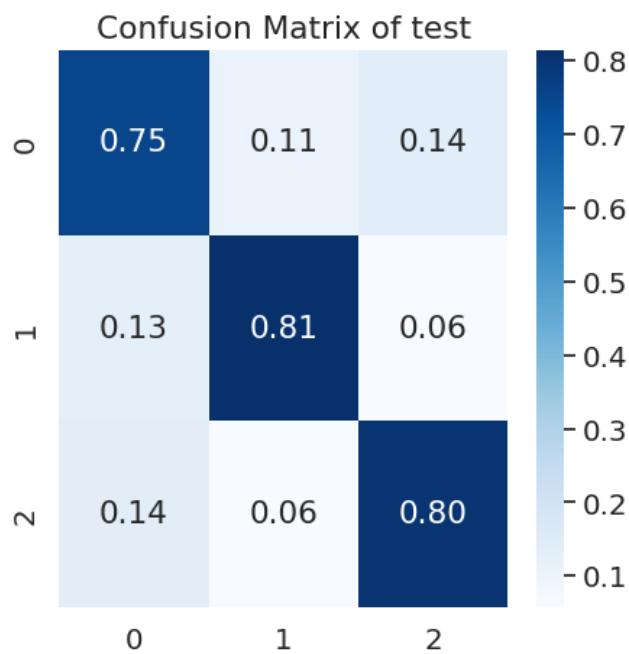


Рис. 78: Результаты в  $X_{test}$ .

## Обучение тринадцатое

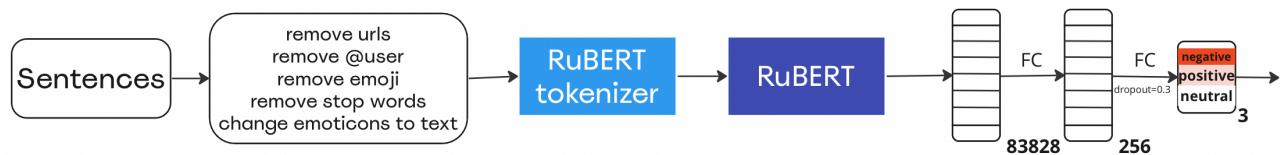


Рис. 79: Архитектура.

Результаты:

Оценка	train	val	test
balanced_accuracy_score	0.866	0.795	0.784
accuracy_score	0.866	0.795	0.784
f1-macro	0.866	0.796	0.785

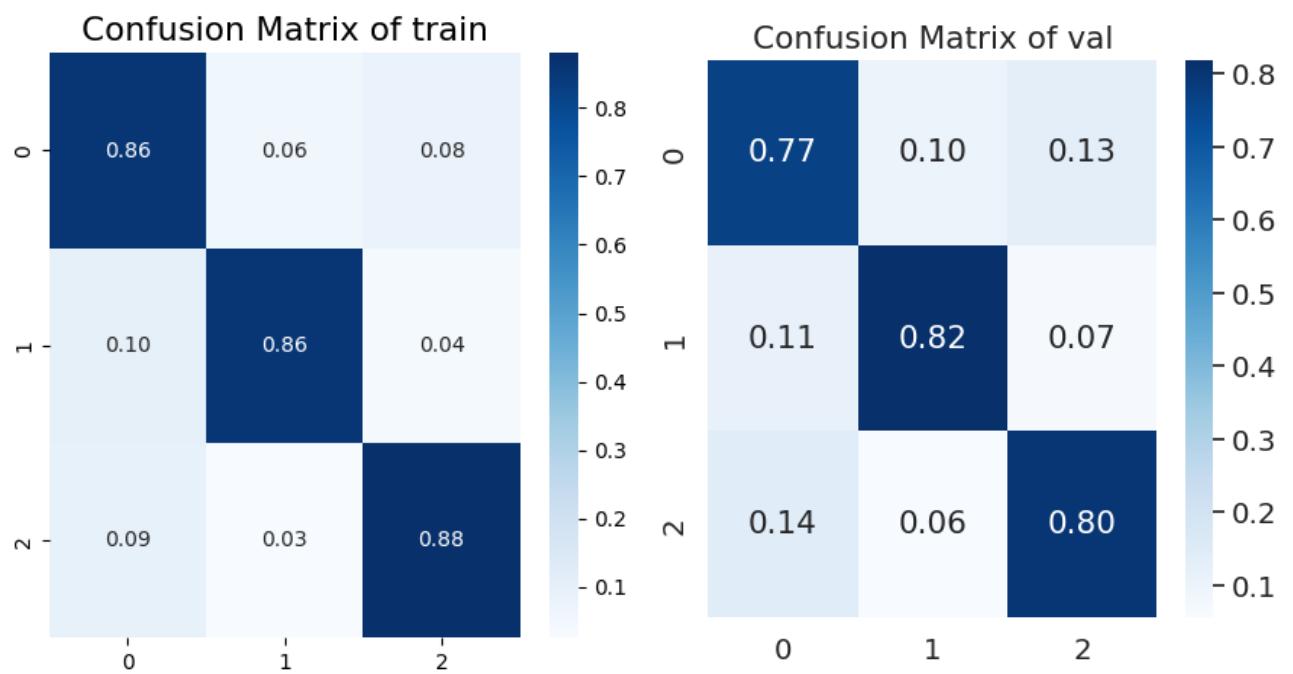


Рис. 80: Результаты в  $X_{train}$ .

Рис. 81: Результаты в  $X_{dev}$ .

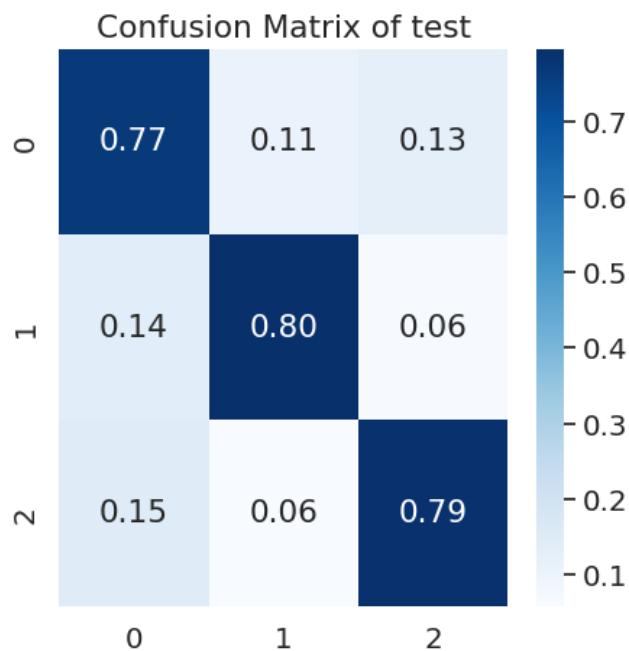


Рис. 82: Результаты в  $X_{test}$ .

## Обучение четырнадцатое

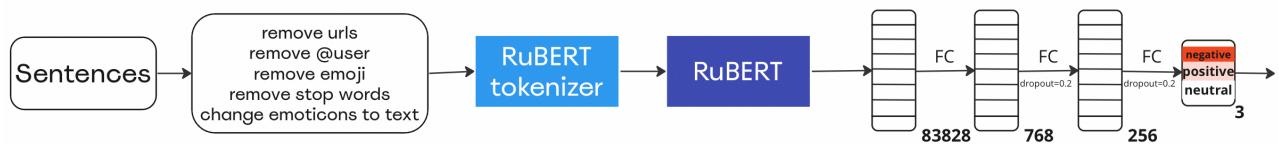


Рис. 83: Архитектура.

Результаты:

Оценка	train	val	test
balanced_accuracy_score	0.904	0.79	0.782
accuracy_score	0.904	0.79	0.782
f1-macro	0.904	0.79	0.78

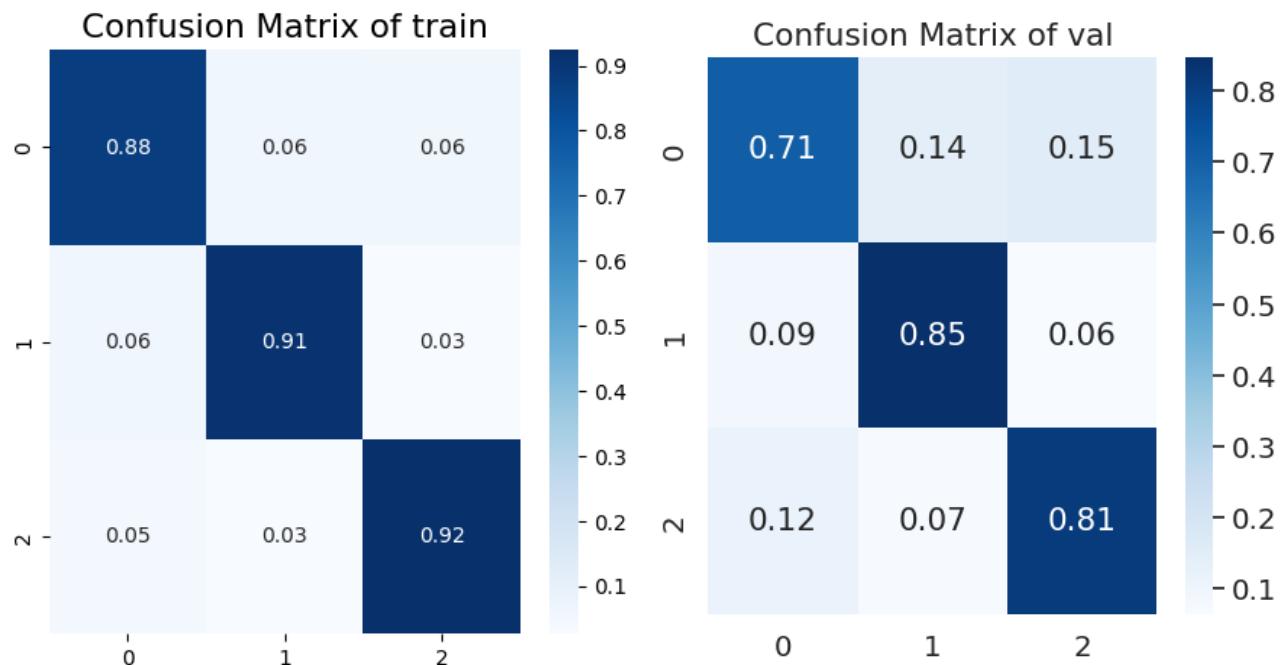


Рис. 84: Результаты в  $X_{train}$ .

Рис. 85: Результаты в  $X_{dev}$ .

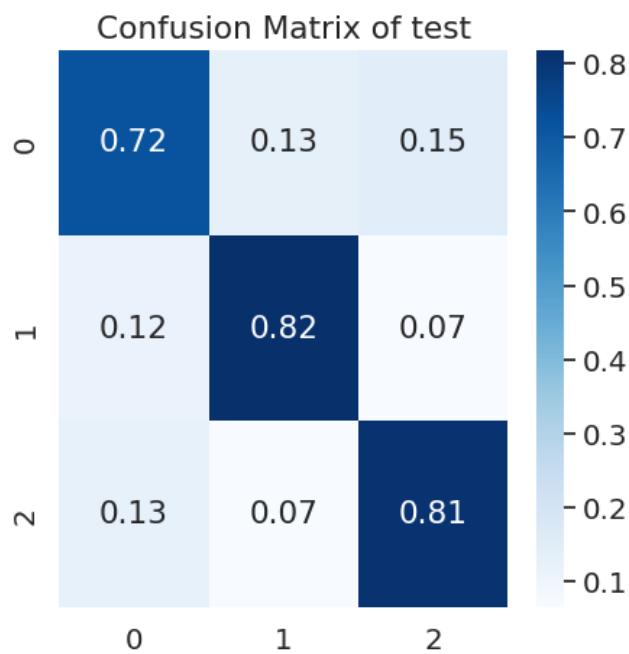


Рис. 86: Результаты в  $X_{test}$ .

## 5.6 Результаты

Ниже представлена таблица результатов:

balanced acc	train	val	test
exp_5	0.996	0.706	0.708
exp_6	0.921	0.766	0.775
exp_7	0.948	0.766	0.774
exp_8	0.905	0.775	0.759
exp_9	0.9053	0.725	0.733
exp_10	0.981	0.731	0.728
exp_11	0.866	0.787	0.78
exp_12	0.88	0.794	0.789
exp_13	0.866	0.795	0.784
exp_14	0.904	0.79	0.782

Рис. 87: результирующая таблица для трех классов.

По данной таблице можно видеть, что лучший результат по test имеет модель RuBERT в Обучении двенадцатом. Она и будет взята за итоговую.

# Заключение

Данная работа раскрывает тему нейронных сетей как в целом, так и в задаче анализа сентимента. Были выполнены следующие пункты:

1. Описана работа нейронных сетей;
2. Описана архитектура Transformer, BERT. Затронуты модели RoBERTa, RuBERT;
3. Найдены два качественных сета данных и составлен сбалансированный датасет для построения модели с хорошей обобщающей способностью;
4. Выполнена предобработка текста, а именно:
  - (a) Удаление ссылок;
  - (b) Удаление меток пользователя;
  - (c) Замена эмотикона на его текстовое значение;
  - (d) Удаление эмоджи;
  - (e) удаление некоторых стоп-слов.
5. Выбраны метрики:
  - (a) balanced accuracy;
  - (b) accuracy;
  - (c) f1-macro.
6. Всего проведено было 14 экспериментов. Дообучались разные модели: BERT, RoBERTa, RuBERT. Лучшим решением оказалась модель RuBERT с предобработкой текста в виде: удаление ссылок, удаление меток пользователя, удаление эмоджи и стоп-слов.

## **Библиографический список**

- [1] Robert Callan. “The Essence of Neural Networks.” (1999) - C.13-35 - ISBN 0-13-908732-X.
- [2] Поляк Б. Т. “Введение в оптимизацию.” (1983) — М.: Наука. Главная редакция физико-математической литературы, — 30 с.
- [3] Нейчев Р.Г., Федотов С.Н. “Метод обратного распространения ошибки”. URL: <https://academy.yandex.ru/handbook/ml/article/metod-obratnogo-rasprostraneniya-oshibki>. (Дата обращения: 25.05.2023).
- [4] Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron. “6.5 Back-Propagation and Other Differentiation Algorithms”. Deep Learning. MIT Press. (2016) - pp. 200–220. - ISBN 9780262035613.
- [5] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation”. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar. Association for Computational Linguistics. (2014) - pp. 1724–1734.
- [6] Y. Wu, M. Schuster, Z. Chen, Q. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”. arXiv preprint arXiv:1609.08144. (2016)
- [7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin. “Attention is all you need”. In Advances in Neural Information Processing Systems. (2017) - pp. 6000-6010.
- [8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” (2019) - In Proceedings of the 2019 Conference of the North American Chapter

of the Association for Computational Linguistics: Human Language Technologies,  
Volume 1 (Long and Short Papers) - pp. 4171–4186.