

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
МОСКОВСКИЙ ИНСТИТУТ ЭЛЕКТРОНИКИ И
МАТЕМАТИКИ

Департамент Прикладной математики

Обработка текстов с помощью нейросетей

Исследовательская работа по дисциплине
Профориентационный семинар «Введение в специальность»

Выполняла работу:

Симонян Диана Гарегиновна

Научный руководитель:

Попов Виктор Юрьевич

Москва, 2023

Содержание

Аннотация	2
I Введение	3
II Теоретические основы	5
III Практическая часть	10
IV Вывод	22
Список источников	24
Приложение	25

Аннотация

Проект «Обработка текстов с помощью нейронных сетей» представляет из себя исследование возможностей использования нейронных сетей для анализа тональности текстовых данных. В рамках проекта была исследована теория по предметной области, изучены методы обработки текста.

В практической части проведена работа по обучению модели на большом объеме размеченных данных, а также оптимизации ее параметров для достижения максимальной точности в определении эмоций найденных в тексте. Полученные результаты демонстрируют высокую эффективность методологии и позволяют рассматривать применение данной технологии в широком спектре приложений, связанных со структурированием информации из неструктурированных источников.

Часть I

Введение

Обработка естественного языка (Natural Language Processing далее **NLP**) сочетает в себе компьютерную лингвистику, машинное обучение и модели глубокого обучения для обработки человеческого языка. [5]

Компьютерная лингвистика – это наука о понимании и построении моделей человеческого языка с помощью компьютеров и программных инструментов. Исследователи используют методы компьютерной лингвистики, такие как синтаксический и семантический анализ, для создания платформ, помогающих машинам понимать разговорный человеческий язык.

Машинное обучение – это технология, которая обучает компьютер с помощью выборочных данных для повышения его эффективности. Человеческий язык имеет несколько особенностей, таких как сарказм, метафоры, вариации в структуре предложений, а также исключения из грамматики и употребления, на изучение которых у людей уходят годы. Программисты используют методы машинного обучения, чтобы научить приложения NLP распознавать и точно понимать эти функции с самого начала.

Глубокое обучение – это особая область машинного обучения, которая учит компьютеры учиться и мыслить как люди. Это включает нейросеть, состоящую из узлов обработки данных, напоминающих операции человеческого мозга. С помощью глубокого обучения компьютеры распознают, классифицируют и сопоставляют сложные закономерности во входных данных.

В современном мире текстовые данные играют огромную роль. Они используются в различных областях, таких как социальные сети, новостные порталы, бизнес-анализ, перевод текстов с одного языка на другой и многое другое. Задача классификации текста позволяет автоматически определять к какому классу относится текст, что может быть полезно для автоматизации процессов обработки и анализа текстовых данных. В данном проекте мы будем использовать нейронные сети для классификации текста и рассмотрим примеры их применения.

В моей работе **объектом** исследования и дальнейшей разработки стала нейронная сеть, решающая задачу Анализа тональности текста (Sentiment analysis).

Предметом исследования стало создание модели нейросети, способная классифицировать тексты на основе их содержания, с высокой точностью и эффективностью.

Цель – изучить принципы работы нейросетей. Научиться решать задачи обработки естественного языка с помощью нейронных сетей.

Задачи:

- Изучение основных концепций и принципов работы нейронных сетей для классификации текста.
- Сбор и подготовка текстовых данных для обучения нейронной сети.
- Разработка и обучение нейронной сети для классификации текста.
- Оценка качества работы нейронной сети с помощью метрик точности.
- Анализ результатов работы нейронной сети и определение возможных путей улучшения её эффективности.
- Исследование возможностей использования обученной нейронной сети в различных областях.

Структура работы:

- 1) Теоретические основы (обзор подходов и инструментов)
- 2) Реализации (а именно: создание нейросети и обучение.)

Часть II

Теоретические основы

Нейронные сети - это компьютерные системы, которые моделируют работу нейронов в головном мозге и используются для решения различных задач, в том числе для обработки текста.

Backpropagation

Backpropagation - это алгоритм обучения нейронной сети, который используется для настройки весов между нейронами. Принцип работы backpropagation заключается в передаче ошибки от выходного слоя к входному слою, чтобы корректировать веса нейронов.

Алгоритм работы backpropagation:

1. Прямой проход: входные данные подаются на входной слой нейронной сети, затем вычисляются значения на каждом слое до выходного слоя.
2. Вычисление ошибки: сравниваются выходные значения нейронной сети с ожидаемыми значениями и вычисляется ошибка.
3. Обратный проход: ошибка передается от выходного слоя к входному слою, корректируя веса нейронов на каждом слое.
4. Обновление весов: после корректировки весов на каждом слое, они обновляются и процесс повторяется до достижения определенного уровня точности.

Оптимизаторы являются ключевым элементом для улучшения процесса обучения с помощью backpropagation. Их работа заключается в изменении скорости обучения и корректировки весов на каждом слое нейронной сети. Некоторые из наиболее популярных оптимизаторов: Adam, RMSprop и Adagrad.

Learning rate - скорость обучения, которая определяет, насколько быстро изменяются веса нейронной сети в процессе обучения. Она является одним из параметров, которые определяются оптимизатором и влияют на процесс обучения

Таблица 1: Оптимизаторы

Название оптимизатора	Формулы подсчёта	Описание параметров
SGD with momentum	$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta)$ $\theta_t = \theta_{t-1} - v_t$	γ - коэффициент затухания η - скорость обучения $\nabla_{\theta} J(\theta)$ - градиент функции потерь по параметрам θ v_t - скользящее среднее градиента на шаге t θ_t - значения параметров на шаге t
Adagrad	$g_t = \nabla_{\theta} J(\theta)$ $s_t = s_{t-1} + g_t^2$ $\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} g_t$	g_t - градиент функции потерь на шаге t s_t - сумма квадратов градиентов до шага t ϵ - малое число для стабильности вычислений
Adam	$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$ $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$ $\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$ $\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$	m_t и v_t - оценки первого и второго моментов градиента на шаге t β_1 и β_2 - коэффициенты затухания для моментов \hat{m}_t и \hat{v}_t - скорректированные оценки моментов ϵ - малое число для стабильности вычислений

Для обучения нейронной сети необходимо подготовить набор данных, который будет использоваться для тренировки. Это могут быть различные текстовые документы, размеченные по категориям. Для улучшения качества обучения можно использовать методы предварительной обработки данных, такие как лемматизация, стемминг и удаление стоп-слов.

Stemming vs Lemmatization

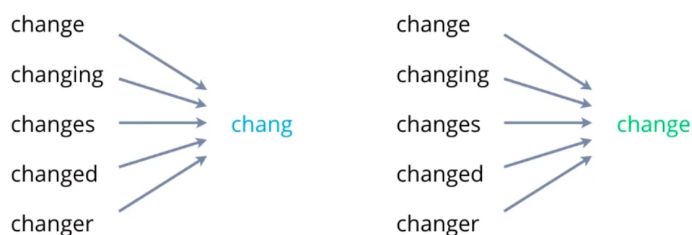


Рис. 1: Лемматизация и Стемминг

Предобработка текста

Предобработка текста является важным шагом перед обучением нейронных сетей для обработки естественного языка. Она включает в себя ряд операций, которые позволяют привести текст к формату, понятному для модели.

Одной из первых операций является токенизация, при которой текст разбивается на отдельные слова или токены. Это может быть достигнуто с помощью различных алгоритмов, таких как методы на основе правил или машинного обучения.

Далее, приводится к нормализованному виду, путем удаления знаков препинания, стоп-слов и других ненужных символов. Это помогает уменьшить размерность входных данных и улучшить качество модели.

Затем текст преобразуется в числовой формат, например, с помощью метода векторизации слов (word embedding), при котором каждое слово представляется в виде вектора чисел. Это позволяет модели работать с числовыми данными и учитывать связи между словами в предложении.

Перед обучением модели проводится операция паддинга (padding), при которой все входные данные приводятся к одной и той же длине.

Word embedding [6]

Word embedding - это метод представления слов в виде векторов чисел. Каждое слово в тексте представляется в виде вектора, где каждая компонента соответствует определенному признаку или характеристике слова.

Word embedding может быть создан с помощью различных алгоритмов, таких как Word2Vec, GloVe и FastText. Эти алгоритмы используют контекстные признаки слов, чтобы определить их семантические свойства и создать векторное представление.

С помощью word embedding модель может учитывать связи между словами в тексте и понимать смысл предложений. Например, слова «кошка» и «собака» будут иметь близкие векторные представления, потому что они относятся к животным, в то время как слово «автомобиль» будет иметь более далекое векторное представление. Word embedding также позволяет модели обраба-

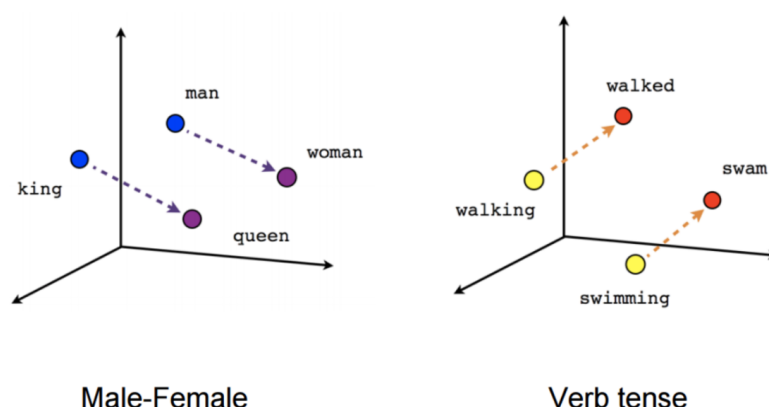


Рис. 2: Word embedding

тывать неизвестные слова, которые не были встречены в обучающих данных. Модель может использовать контекстные признаки, чтобы определить семантику неизвестного слова и создать соответствующее векторное представление.

Чтобы улучшить векторное представление слов, и предоставить возможность нейронной сети «выучить» язык, в 2018 году была представлена нейросеть BERT, которая является одной из самых популярных моделей в NLP.

Намного выгоднее заранее обучить подобное векторное представление слов на каком-нибудь огромном корпусе текстов, например на всей Wikipedia, а в конкретных нейронных сетях использовать уже готовые векторы слов, а не обучать их каждый раз заново.

BERT [3]

Нейронная сеть BERT использует архитектуру трансформеров* (см Приложение) , которая позволяет ей эффективно обрабатывать последовательности данных, такие как тексты. Она состоит из множества слоев, каждый из которых содержит несколько механизмов внимания (attention mechanisms), которые позволяют модели сосредоточиться на наиболее важных частях входных данных.

BERT является бидирекциональной моделью, что означает, что она способна учитывать контекст как слева, так и справа от текущего слова. Это позволяет ей лучше понимать связи между словами в предложении и улучшать качество обработки естественного языка.

Для обучения BERT используется метод маскирования (masking), при котором случайно выбранные слова в предложении заменяются на специальный токен [MASK]. Это позволяет модели учиться предсказывать пропущенные слова и лучше понимать контекст.

Кроме того, BERT использует метод предсказания следующего предложения (next sentence prediction), при котором модель обучается предсказывать, являются ли два предложения последовательными или нет. Это помогает модели лучше понимать связи между предложениями и улучшать качество ответов на вопросы.

BERT является мощной моделью нейронной сети для обработки естественного языка, которая позволяет достигать высокой точности в решении различных задач. Ее архитектура и принципы работы позволяют ей эффективно учитывать контекст и связи между словами в предложении.

Поэтому в практической части для представления слов в форме векторов будем использовать предобученную BERT. То есть реализация нейронной сети будет в формате дообучения BERT под нашу задачу.

Часть III

Практическая часть

Задача: Анализ тональности текста (Sentiment analysis) [датасета с Kaggle](#).

Для реализации нейронной сети я выбрала библиотеку PyTorch [4] на Python по нескольким причинам:

1. *Простота и удобство использования:* PyTorch имеет простой и интуитивно понятный интерфейс, что делает его легким в освоении и использовании.
2. *Гибкость:* PyTorch позволяет гибко определять структуру нейронных сетей и изменять ее в процессе обучения, что может быть очень полезно при экспериментировании с различными архитектурами.
3. *Высокая производительность:* PyTorch использует оптимизированные вычисления на графических процессорах (GPU), что позволяет значительно ускорить процесс обучения.

1. Введение

Цель - описать процесс классификации эмоций в текстах с помощью нейросети на основе датасета «Emotions Dataset for NLP». Для достижения этой цели были выполнены следующие шаги: подготовка данных, создание модели нейросети, обучение модели и оценка результатов.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer, BertModel
from sklearn.preprocessing import LabelEncoder
```

Рис. 3: Импортируемые библиотеки

Импортируем необходимые библиотеки: Pandas и Numpy для работы с датасетом, matplotlib.pyplot для отрисовки графиков, tqdm для оборачивания итерируемых объектов, torch для написания нейросети и LabelEncoder из sklearn.preprocessing для кодировки категориальных переменных.

2. Описание датасета

Датасет «Emotions Dataset for NLP» содержит 416809 текстовых сообщений, каждое из которых относится к одной из пяти эмоций: радость, грусть, злость, страх и нейтральность.

```
train_df = pd.read_csv('train.txt', names=['text', 'emotion'], sep=';')
val_df = pd.read_csv('val.txt', names=['text', 'emotion'], sep=';')
test_df = pd.read_csv('test.txt', names=['text', 'emotion'], sep=';')

train_df.head()
```

	text	emotion
0	i didnt feel humiliated	sadness
1	i can go from feeling so hopeless to so damned...	sadness
2	im grabbing a minute to post i feel greedy wrong	anger
3	i am ever feeling nostalgic about the fireplac...	love
4	i am feeling grouchy	anger

Рис. 4: Загрузка train, val и test

```
train_df.groupby(['emotion']).size().plot.bar(color=['#FF5733', '#4C8B17', '#1E90FF', '#8B008B', '#FFD700', '#FF1493']);
```

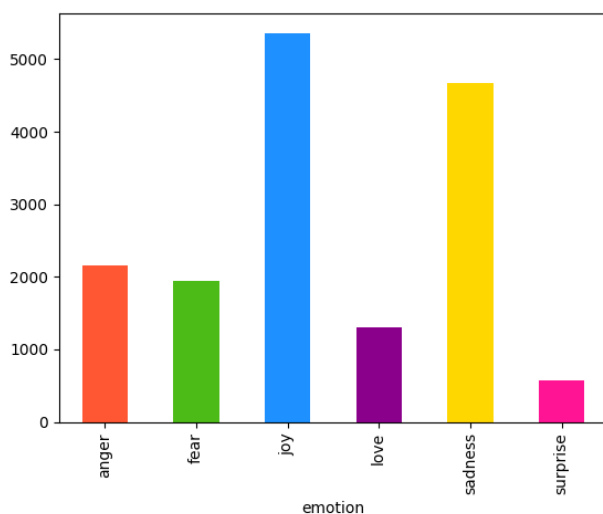


Рис. 5: Распределение классов

```
train_df['text'].apply(lambda x : len(x)).describe()

count    16000.000000
mean      96.845812
std       55.904953
min        7.000000
25%       53.000000
50%       86.000000
75%      129.000000
max       300.000000
Name: text, dtype: float64
```

Рис. 6: Статистика распределения длин слов с датасете

Так как свыше 75% длины слов меньше 256, то $MAX_LEN = 256$

3. Подготовка данных

Перед обучением модели необходимо было подготовить данные. Для этого были выполнены следующие шаги:

- Токенизация: каждое сообщение было разбито на отдельные слова.
- Перевод меток классов в числа: каждой эмоции было присвоено уникальное числовое значение.

```
from transformers import BertTokenizer, BertModel

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```




Downloading (...)solve/main/vocab.txt: 100%  232k/232k [00:00<00:00, 3.72MB/s]
 Downloading (...)okenizer_config.json: 100%  28.0/28.0 [00:00<00:00, 2.02kB/s]
 Downloading (...)ve/main/config.json: 100%  570/570 [00:00<00:00, 25.3kB/s]

Рис. 7: Загрузка токенайзера

```
le = LabelEncoder()
train_df['emotion'] = le.fit_transform(train_df['emotion'])
val_df['emotion'] = le.transform(val_df['emotion'])
val_df.head()
```

	text	emotion
0	im feeling quite sad and sorry for myself but ...	4
1	i feel like i am still looking at a blank canv...	4
2	i feel like a faithful servant	3
3	i am just feeling cranky and blue	0
4	i can have for a treat or if i am feeling festive	2

Рис. 8: Кодирование категориальных переменных

Создадим Dataset класс для нашего набора данных. [7]

```
class EmotionDataset(Dataset):
    def __init__(self, df, tokenizer, max_len):
        self.df = df
        self.tokenizer = tokenizer
        self.max_len = max_len

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

    def __getitem__(self, index):
        text = str(self.df.loc[index, 'text'])
        emotion = self.df.loc[index, 'emotion']

        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            truncation=True,
            max_length=self.max_len,
            return_token_type_ids=False,
            padding='max_length',
            return_attention_mask=True,
            return_tensors='pt',
        )

        return {
            'text': text,
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'emotion': torch.tensor(emotion, dtype=torch.long)
        }
```

Рис. 9: EmotionDataset

Стандартный класс расширяется методами `__init__`, `__len__`, `__getitem__`. В методе `__init__` инициализируем тексты, метки, максимальную длину текста в токенах, а так же токенайзер.

Метод `len` возвращает длину нашего датасета. Метод `getitem` возвращает словарь, который состоит из самого исходного текста, списка токенов, маски внимания, а также метки класса. Отдельно хочется остановиться на настройках токенизатора с помощью метода `encode_plus()`.

В этом методе мы указываем токенизатору, что исходный текст нужно обрамлять служебными токенами `add_special_tokens=True`, а также дополнять полученные векторы до максимальной длины `padding='max_len'`.

Далее указываем `shuffle=True` для перемешки значений тренировочной выборки. `TRAIN_BATCH_SIZE = 16` `VALID_BATCH_SIZE = 16`

Создаем DataLoader для итерации по батчам.

```
train_dataset = EmotionDataset(train_df, tokenizer, MAX_LEN )
train_loader = DataLoader(train_dataset, batch_size=TRAIN_BATCH_SIZE ,
                          shuffle=True)

val_dataset = EmotionDataset(val_df, tokenizer, MAX_LEN)
val_loader = DataLoader(val_dataset, batch_size=VALID_BATCH_SIZE )
```

Рис. 10: DataLoader

4. Создание модели нейросети

Архитектура BertClassifier:

- Слой входных эмбеддингов (input embeddings layer): преобразует входные данные в векторы фиксированной длины, используя предобученную модель BERT.
- Скрытый слой (Dropout): с коэффициентом 0.2 для предотвращения переобучения.
- Скрытый слой (Linear): содержит 256 нейронов и функцию активации ReLU.
- Скрытый слой (Dropout): с коэффициентом 0.2.
- Выходной слой (Linear): содержит 6 нейронов (по числу классов).

```
class BertClassifier(nn.Module):

    def __init__(self, bert_model_name, dropout=0.2):

        super(BertClassifier, self).__init__()

        self.bert = BertModel.from_pretrained(bert_model_name)
        self.dropout1 = nn.Dropout(dropout)
        self.fc1 = nn.Linear(768, 256)
        self.relu = nn.ReLU()
        self.dropout2 = nn.Dropout(dropout)
        self.fc2 = nn.Linear(256, 6)

    def forward(self, input_ids, attention_mask):

        outputs = self.bert(input_ids=input_ids,
                             attention_mask=attention_mask)
        pooled_output = outputs.pooler_output
        pooled_output = self.dropout1(pooled_output)
        output = self.fc1(pooled_output)
        output = self.relu(output)
        output = self.dropout2(output)
        logits = self.fc2(output)

        return logits
```

Рис. 11: BertClassifier

5. Обучение модели

```
[ ] def train(model, train_data_loader, val_data_loader, optimizer, criterion,
            device, EPOCHS):

    train_losses = []
    train_accuracies = []
    val_losses = []
    val_accuracies = []

    for epoch in range(EPOCHS):
        print(f'Epoch {epoch + 1} ')

        # переводим модель в режим обучения
        model.train()

        # инициализируем Loss и метрику Accuracy
        total_loss_train, total_accuracy_train = 0, 0

        # итерируемся по батчам
        for data in tqdm(train_data_loader):
            # переносим данные на device
            input_ids = data['input_ids'].to(device)
            attention_mask = data['attention_mask'].to(device)
            labels = data['emotion'].to(device)

            # обнуляем градиенты
            optimizer.zero_grad()

            # получаем предсказанные значения
            outputs = model(input_ids, attention_mask)

            # получаем loss
            loss = criterion(outputs, labels)
            total_loss_train += loss.item()

            # считаем accuracy
            logits = outputs.detach().cpu().numpy()
            label_ids = labels.to('cpu').numpy()

            total_accuracy_train += flat_accuracy(logits, label_ids)

            # производим backpropagation и делаем шаг оптимизации
            loss.backward()
            optimizer.step()

        # считаем средний loss и accuracy
        avg_loss_train = total_loss_train / len(train_data_loader)
        avg_acc_train = total_accuracy_train / len(train_data_loader)

        train_losses.append(avg_loss_train)
        train_accuracies.append(avg_acc_train)
```

Рис. 12: train() часть 1


```

# переводим модель в режим предсказания
model.eval()

# инициализируем Loss и метрику Accuracy
total_loss_val = 0
total_accuracy_val = 0

# итерируемся по батчам
for data in tqdm(val_data_loader):
    # переносим данные на device
    input_ids = data['input_ids'].to(device)
    attention_mask = data['attention_mask'].to(device)
    labels = data['emotion'].to(device)

    # отключить вычисление градиента
    with torch.no_grad():
        # получаем предсказание модели
        outputs = model(input_ids, attention_mask)

        # получаем loss
        loss = criterion(outputs, labels)
        total_loss_val += loss.item()

        # считаем кол-во правильных предсказаний
        logits = outputs.detach().cpu().numpy()
        label_ids = labels.to('cpu').numpy()

        total_accuracy_val += flat_accuracy(logits, label_ids)

# считаем средний loss и accuracy
avg_loss = total_loss_val / len(val_data_loader)
accuracy = total_accuracy_val / len(val_data_loader)

val_losses.append(avg_loss)
val_accuracies.append(accuracy)

print(f'Train Loss: {avg_loss_train:.4f} | Train Acc: {avg_acc_train:.4f}
      | Val Loss: {avg_loss:.4f} | Val Acc: {accuracy:.4f}')

# Сохраняем веса модели
torch.save(model.state_dict(), f'weights_epoch{epoch + 1}.pt')

return train_losses, train_accuracies, val_losses, val_accuracies

```

Рис. 13: train() часть 2

```

[ ] def flat_accuracy(preds, labels):
    pred_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
    return np.sum(pred_flat == labels_flat) / len(labels_flat)

```

Рис. 14: flat_accuracy

Подготовка к обучению:

Функция `train()` реализует процесс обучения модели. На вход она принимает модель, загрузчики данных для обучения и валидации, оптимизатор, функцию потерь, устройство, на котором будет проходить обучение, и количество эпох.

Внутри функции происходит итерация по эпохам. На каждой эпохе модель переводится в режим обучения и проходит по всем батчам из загрузчика данных для обучения. Для каждого батча вычисляется `loss` и `accrasy`, производится `backpropagation` и делается шаг оптимизации. После прохода по всем батчам считаются средний `loss` и `accrasy` для обучающей выборки.

Затем модель переводится в режим предсказания и проходит по всем батчам из загрузчика данных для валидации. Для каждого батча вычисляется `loss` и `accrasy`. После прохода по всем батчам считаются средний `loss` и `accrasy` для валидационной выборки.

На каждой эпохе также сохраняются веса модели в файл.

Функция возвращает списки `train_losses`, `train_accrasyes`, `val_losses`, `val_accrasyes`, содержащие значения `loss` и `accrasy` для каждой эпохи обучения на обучающей и валидационной выборках.

Обучение:

Создадим модель классификатора на основе предобученной BERT-модели ('bert-base-uncased'). Определим функцию потерь (кросс-энтропию) и устройство, на котором будет выполняться обучение (если доступна видеокарта - CUDA, иначе CPU).

Задаём оптимизатор Adam для обновления весов модели. Переместим модель на выбранное устройство.

Далее начинаем обучение, вызвав функцию `train()` с использованием переданных параметров.

В результате выполнения этой строки кода будут получены списки значений потерь (`losses`) и точности (`accrasyes`) как для тренировочных данных так и для валидационных после каждой эпохи обучения. Сохраним их в переменные для отрисовки процесса обучения.

```

model = BertClassifier('bert-base-uncased')

# используем Кросс энтропию для задачи многоклассовой классификации
criterion = nn.CrossEntropyLoss()

# определяем устройство
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# определяем оптимизатор
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)

# перемещаем model на device
model.to(device)

train_losses, train_accuracies, val_losses, val_accuracies = train(model, train_loader, val_loader, optimizer, criterion, device, EPOCHS)

```

Epoch 1
 100%|██████████| 1000/1000 [11:09<00:00, 1.49it/s]
 100%|██████████| 125/125 [00:30<00:00, 4.07it/s]
 Train Loss: 0.8383 | Train Acc: 0.7114 | Val Loss: 0.3159 | Val Acc: 0.9115
 Epoch 2
 100%|██████████| 1000/1000 [11:09<00:00, 1.49it/s]
 100%|██████████| 125/125 [00:30<00:00, 4.06it/s]
 Train Loss: 0.2255 | Train Acc: 0.9271 | Val Loss: 0.2035 | Val Acc: 0.9320
 Epoch 3
 100%|██████████| 1000/1000 [11:11<00:00, 1.49it/s]
 100%|██████████| 125/125 [00:30<00:00, 4.07it/s]
 Train Loss: 0.1462 | Train Acc: 0.9450 | Val Loss: 0.1625 | Val Acc: 0.9330
 Epoch 4
 100%|██████████| 1000/1000 [11:16<00:00, 1.48it/s]
 100%|██████████| 125/125 [00:32<00:00, 3.81it/s]
 Train Loss: 0.1151 | Train Acc: 0.9505 | Val Loss: 0.1453 | Val Acc: 0.9390

Рис. 15: Обучение

Зададим гиперпараметры: $EPOCHS = 4$, $LEARNING_RATE = 1e-05$
 Обученная модель достигает качества Ассигасы на тестовой выборке 95%
 и на валидационной 93.9%.

```

#Функция для отрисовки прогресса обучения

def plot_metrics(train_losses, val_losses, train_accuracies, val_accuracies):
    fig, ax = plt.subplots(1, 2, figsize=(12, 4))

    # Строем график loss на обучающем и тестовом датасете
    ax[0].plot(train_losses, label='Обучающий Loss')
    ax[0].plot(val_losses, label='Тестовый Loss')

    ax[0].set_xlabel('Эпоха')
    ax[0].set_ylabel('Loss')
    ax[0].set_title('Обучающий и Тестовый Loss')
    ax[0].legend()

    # Строем график ассигасы на обучающем и тестовом датасете
    ax[1].plot(train_accuracies, label='Обучающий Accuracy')
    ax[1].plot(val_accuracies, label='Тестовый Accuracy')
    ax[1].set_xlabel('Эпоха')
    ax[1].set_ylabel('Accuracy')
    ax[1].set_title('Обучающий и Тестовый Accuracy')

    plt.show()

```

Рис. 16: plot_metrics()

Посмотрим на процесс обучения:

```
plot_metrics(train_losses, val_losses, train_accuracies, val_accuracies)
```

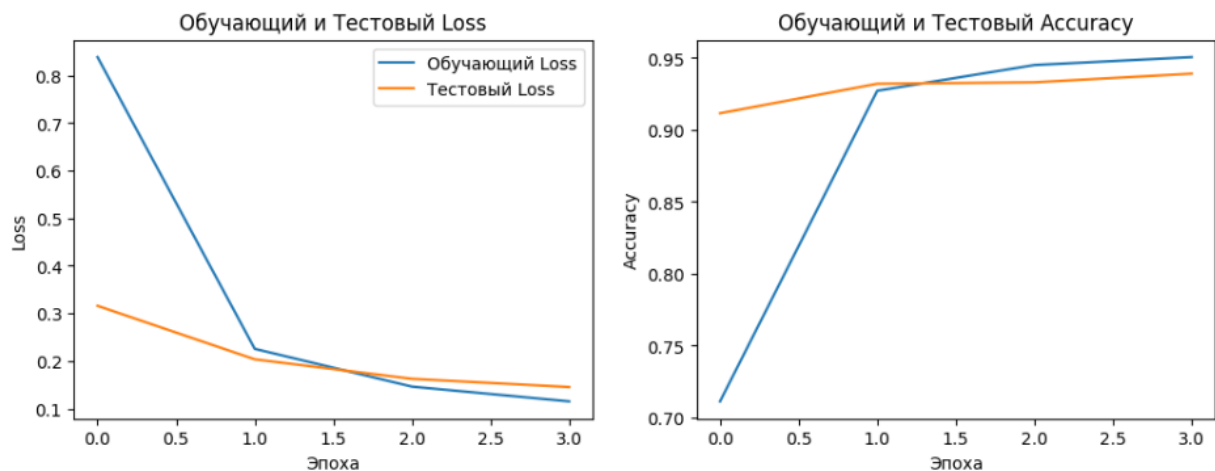


Рис. 17: Процесс обучения

Видим что loss на train и val падает, значит модель действительно обучается. Ассигасу растёт и разрыв между значениями ассигасу на train и val не такой уж и большой, значит удалось избежать переобучения.

6. Оценка результатов

Для оценки результатов, напишем функцию `predict()`, принимающую обученную модель (`model`) и тестовую выбоку.

Возвращаемое значение `predict()` - ассигасу на тестовой выборке.

```
# создаем функцию predict
def predict(model, test_data):

    test = EmotionDataset(test_data, tokenizer, MAX_LEN)
    test_dataloader = DataLoader(test, batch_size=VALID_BATCH_SIZE)

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    if use_cuda:
        model = model.cuda()

    # переводим модель в режим предсказания
    model.eval()

    # инициализируем метрику Accuracy
    total_accuracy_test = 0
```

Рис. 18: `predict()` часть 1

```

# итерируемся по батчам
for data in tqdm(test_dataloader):
    # переносим данные на device
    input_ids = data['input_ids'].to(device)
    attention_mask = data['attention_mask'].to(device)
    labels = data['emotion'].to(device)

    # отключить вычисление градиента
    with torch.no_grad():
        # получаем предсказание модели
        outputs = model(input_ids, attention_mask)

        # считаем кол-во правильных предсказаний
        logits = outputs.detach().cpu().numpy()
        label_ids = labels.to('cpu').numpy()

        total_accuracy_test += flat_accuracy(logits, label_ids)

# считаем средний loss и accuracy
accuracy = total_accuracy_test / len(test_dataloader)

print(f'Accuracy на тесте: {accuracy: .3f}')

```

Рис. 19: predict() часть 2

Проведем оценку результатов на тестовой выборке.

```

test_df['emotion'] = le.transform(test_df['emotion']) # закодируем целевые переменные

predict(model, test_df)

100%|██████████| 125/125 [00:31<00:00, 3.95it/s]Accuracy на тесте: 0.927

```

Рис. 20: оценка результатов

Ассигуру на тестовой выборке достиг значения 92.7% . Для улучшения результатов необходимо провести дополнительные исследования и адаптировать модель к конкретным задачам.

7. Возможное использование

Данную нейросеть можно использовать в следующих задачах:

1. Мониторинг социальных медиа: нейросеть может анализировать тональность сообщений в социальных сетях, так компании могут анализировать отношение пользователей к их продуктам или услугам и принимать меры для улучшения своей репутации.

2. Анализ мнений в опросах: нейросеть может использоваться для анализа мнений в опросах. Это может помочь политическим кампаниям исследовать общественное мнение и принимать меры для улучшения своей стратегии.

3. Анализ обращений клиентов: нейросеть может использоваться для анализа обращений клиентов в службу поддержки. Это может помочь компаниям быстро выявлять проблемы и решать их.

Часть IV

Вывод

В результате выполнения задачи классификации эмоций в текстах с помощью дообученной нейросети BERT на основе датасета «Emotions Dataset for NLP» была создана модель, которая показала достаточно высокую точность (accuracy) в 92.7%.

Это свидетельствует о том, что дообученные нейросети, такие как BERT, могут быть эффективным инструментом для классификации эмоций в текстах.

Проект «Обработка текстов с помощью нейронных сетей. Задача классификации текста» имеет большое значение для бизнеса и общества в целом, так как позволяет эффективно анализировать большие объемы текстовых данных и принимать соответствующие решения на основе полученной информации.

Создание инструмента, который ускорит и упростит процесс анализа текстовых данных, является важным шагом в развитии технологий и повышении эффективности работы компаний и организаций. Разработка точной модели нейросети для классификации текстов является ключевым фактором успеха проекта и может привести к значительному улучшению результатов анализа текстовых данных.

Список иллюстраций

1	Лемматизация и Стемминг	7
2	Word embedding	8
3	Импортируемые библиотеки	10
4	Загрузка train, val и test	11
5	Распределение классов	11
6	Статистика распределения длин слов с датасете	12
7	Загрузка токенайзера	12
8	Кодирование категориальных переменных	12
9	EmotionDataset	13
10	DataLoader	14
11	BertClassifier	14
12	train() часть 1	15
13	train() часть 2	16
14	flat_accuracy	16
15	Обучение	18
16	plot_metrics()	18
17	Процесс обучения	19
18	predict() часть 1	19
19	predict() часть 2	20
20	оценка результатов	20
21	Трансформер	25
22	Кодирующий слой	25
23	Декодированный слой	25

Список таблиц

1	Оптимизаторы	6
---	------------------------	---

Список источников

- [1] Клуб Теха Лекций от МФТИ
- [2] Трансформер (модель машинного обучения)
- [3] BERT — state-of-the-art языковая модель для 104 языков
- [4] PyTorch
- [5] Глубинное обучение для автоматической обработки текстов
- [6] Векторное представление слов
- [7] BERT для классификации русскоязычных текстов

Приложение

* **Transformers.** [2] Основным принципом работы Transformers является использование механизма внимания (attention mechanism), который позволяет модели фокусироваться на наиболее важных частях текста.

Архитектура Transformers состоит из нескольких слоев, каждый из которых содержит блоки кодировщика и декодировщика. Каждый блок кодировщика и декодировщика содержит многослойный перцептрон (multi-layer perceptron) и механизм внимания.

Процесс работы модели начинается с подачи текстовых данных на вход кодировщика. Затем модель использует механизм внимания для определения наиболее важных частей текста и создает векторное представление для каждого слова. Далее, эти векторные представления передаются на декодировщик, который генерирует ответ на основе полученных данных.

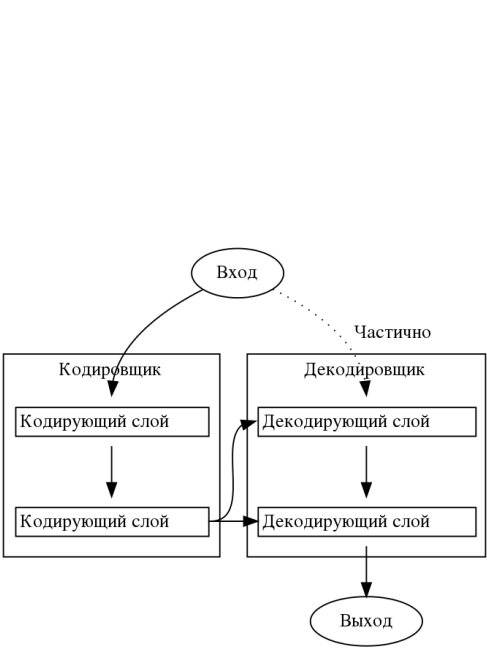


Рис. 21: Трансформер

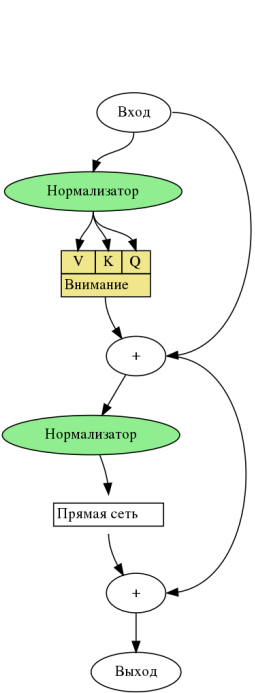


Рис. 22:
Кодирующий
слой

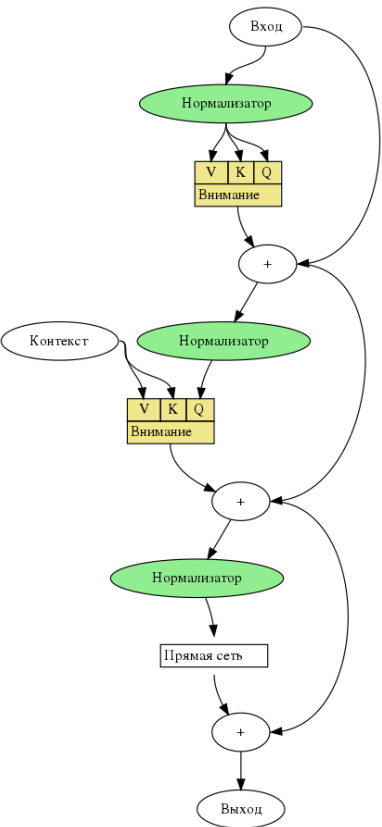


Рис. 23:
Декодировший слой