«Санкт-Петербургский государственный электротехнический университет
«ЛЭТИ» им. В.И.Ульянова (Ленина)»
(СПбГЭТУ "ЛЭТИ")

Направление	01.03.02 - Прикладная математика и информатика
Профиль	Направление без профиля
Факультет	КТИ
Кафедра	МО ЭВМ

К защите допустить
Зав. кафедрой Кринкин К.В.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

АВТОМАТИЧЕСКИЙ АНАЛИЗ ТОНАЛЬНОСТИ РЕЦЕНЗИЙ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ TENSORFLOW

Студент _____ Фанифатьева А. Д.
подпись

Руководитель *ассистент* _____ Шолохов А.В.
подпись

Консультанты *доцент* _____ Яценко И.В.
подпись

доцент _____ Фомин В.И
подпись

Санкт-Петербург
2017

ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Утверждаю
Зав. кафедрой МО ЭВМ
_____ Кринкин К.В.
« __ »_____ 2017 г.

Студент: Фанифатьева А. Д.

Тема работы: Автоматический анализ тональности рецензий с использованием библиотеки Tensorflow

Место выполнения ВКР: СПбГЭТУ "ЛЭТИ" , кафедра МО ЭВМ

Исходные данные (технические требования): Python 3.4.3, TensorFlow 1.1.0

Содержание ВКР:

1) Исследование методов классификации, основанных на нейронных сетях, и классических (линейных и нелинейных) методов;

2) Разработка свёрточной нейронной сети, рекуррентной нейронной сети и их обучение с использованием Tensorflow;

3) Оценка точности разработанных алгоритмов и их сравнение.

Перечень отчетных материалов: пояснительная записка, иллюстративный материал.

Дополнительный раздел: Информационный маркетинг.

Дата выдачи задания	Дата представления ВКР к защите
«24» апреля 2017г.	«21» июня 2017г.

Студент _____ Фанифатьева А. Д.
подпись

Руководитель *ассистент* _____ Шолохов А.В.
подпись

Консультанты *доцент* _____ Яценко И.В.
подпись

доцент _____ Фомин В.И
подпись

КАЛЕНДАРНЫЙ ПЛАН

Студент: Фанифатьева А. Д. Группа № 3382

Тема ВКР: Автоматический анализ тональности рецензий с использованием библиотеки Tensorflow.

Сроки выполнения ВКР : 24.04 - 04.06

№ п/п	Наименование работ	Срок выполнения
1	Изучение методов глубокого обучения	24.04 - 14.05
2	Реализация методов глубокого обучения и сравнение их эффективности	14.05 - 24.05
4	Написание пояснительной записке о выполненной работе	24.05 - 29.05
5	Анализ и правки пояснительной записки и презентаций	29.05 - 04.06

Студент _____ Фанифатьева А. Д.
подпись

Руководитель *ассистент* _____ Шолохов А.В.
подпись

« __ » _____ 2017

РЕФЕРАТ

Пояснительная записка 82 страниц, 21 рис., 8 таблиц, 27 источников, 3 приложения.

Ключевые слова: искусственные нейронные сети, глубокие нейронные сети, обучение с учителем, глубокое обучение, рекуррентная нейронная сеть, LSTM, свёрточная нейронная сеть, анализ тональности текста, мешок слов, word2vec.

Объектом исследования являются методы на основе нейронных сетей для анализа тональности корпуса текстов.

Для достижения поставленной в работе цели необходимо решить следующие задачи:

- Изучить теоретический материал про обучение глубинных нейронных сетей и их особенности применительно к обработке естественного языка;
- Изучить документацию библиотеки Tensorflow;
- Разработать модели свёрточной и рекуррентной нейронных сетей;
- Разработать реализацию линейных и нелинейных методов классификации на моделях мешка слов и Word2Vec;
- Сравнить точность и другие показатели качества реализованных нейросетевых моделей с классическими методами.

  Для визуализации обучения используется Tensorboard.

В работе показано преимущество классификаторов на основе глубоких нейронных сетей над классическими методами классификации, даже если для векторных представлений слов используется модель Word2Vec. Самую высокую предсказательную точность для данного корпуса текстов имеет модель рекуррентной нейронной сети с LSTM-блоками.

ABSTRACT

Thesis 82 pages, 21 figures, 27 sources, 3 appendixes.

This project is focused on the binary classification of movie reviews. The goal is developing deep and supervised machine learning algorithms to make sentiment analysis on short texts.

The following tasks are:

- Learn deep learning fundamentals and special features of training deep neural networks on NLP problems;
- Read Tensorflow documentation;
- Develop following architectures: convolutional neural network and recurrent neural network with LSTM units;
- Implement supervised learning methods using Bag of Words and Word2Vec models;
- Compare accuracy and other quality metrics of deep neural networks and supervised learning methods.

The following program is binary classifier which can use different models to make predictions for short texts corpora.

Содержание

СПИСОК СОКРАЩЕНИЙ

В настоящей пояснительной записке применяют следующие термины с соответствующими определениями:

- Deep learning — глубокое обучение;
- Supervised learning — обучение с учителем;
- SVM (support vector machine) — метод опорных векторов;
- Pooling — операция объединения в свёрточных нейронных сетях;
- Softmax — функция мягкого максимума;
- Dropout — метод регуляризации для предотвращения переобучения сети;
- Batches — группы примеров, используемые для обучения сети;
- CBOW (continuous bag of words) — непрерывный мешок со словами, один из алгоритмов обучения Word2Vec;
- CNN (convolutional neural network)— свёрточная нейронная сеть;
- RNN (recurrent neural network) — рекуррентная нейронная сеть;
- LSTM (long short term memory) — долгая краткосрочная память, разновидность архитектуры рекуррентных нейронных сетей.

ВВЕДЕНИЕ

Каждый день в интернете появляется огромное количество контента: пользователи высказывают своё мнение о фильмах и событиях, оставляют отзывы о различных продуктах и услугах. Для решения задач, связанных с анализом эмоциональной окраски текста, используются методы анализа тональности текста.

Задача автоматического анализа тональности текста является довольно популярной [1]. Зачастую пользователи при выборе чего-либо (например, в какой университет поступить) руководствуются мнениями других людей. Поэтому набор обработанных мнений представляет значительный интерес для социологов, маркетологов и владельцев бизнеса. Также система для анализа тональности текста может пригодиться на сайтах, где люди оставляют отзывы: например, пользователь может положительно оценить фильм, случайно поставив отрицательную оценку — система автоматического анализа тональности текста исправит ошибку. Поэтому анализ тональности текстов является важной и актуальной задачей.

Автоматический анализ тональности текста обычно применяется на корпусах текстов, которые содержат рецензии или отзывы. Также его можно применить для данных из блогов и социальных сетей, чтобы извлечь общественное мнение по тому или иному вопросу или товару.

Определение эмоциональной оценки авторов рецензий не является тривиальной задачей. Стандартные методы классификации текста изолируют слова по признакам и применяют различные методы выбора признаков, чтобы найти наиболее “важное слово” в тексте: например, предложения “Мне нравится бегать” и “Мне не нравится бегать” будут считаться одинаковыми. В ситуации, когда рецензия представлена только положительно окрашенным набором слов, а на самом деле является отрицательной, стандартные методы классификации текста перестают быть эффективными. Методы машинного обучения позволяют алгоритмам “понимать” структуру предложения и его семантическую структуру. Предложение будет представлено в виде вектора, в котором сохранена структура предложения и то, как слова связаны с друг другом.

Для проведения численных экспериментов были использованы рецензии сайта RottenTomatoes [2] — набор из 5331 позитивных и 5331 негативных рецензий.

Требовалось построить бинарный классификатор, определяющий, позитивной или негативной оказалась рецензия. В качестве методов рассматриваются свёрточная нейронная сеть и рекуррентная нейронная сеть с LSTM-блоками, наивный байесовский классификатор, метод опорных векторов и логистическая регрессия. Также были использованы два варианта векторных моделей представления текста: мешок слов (Bag of Words) и Word2Vec.

В результате обучения были получены модели нейронных сетей, позволяющие с высокой точностью определять тональность рецензий, а также проведено сравнение эффективности использования реализованных методов.

ПОСТАНОВКА ЗАДАЧИ

Целью дипломного проекта является разработка алгоритмов для анализа тональности текста на основе глубоких нейронных сетей (свёрточной и рекуррентной), а также сравнение их эффективности с другими классификаторами. В качестве средства разработки будет использована библиотека TensorFlow.

Для достижения цели в ходе разработки должны быть решены следующие задачи:

- Знакомство с методами машинного обучения для определения тональности текста;
- Обзор существующих нейросетевых моделей для представления текстовых данных;
- Программная реализация свёрточной нейронной сети и рекуррентной нейронной сети с добавлением LSTM-блоков с использованием Tensorflow;
- Проведение численных экспериментов, сравнение результатов.

ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

Задачей, поставленной в работе, является реализация различных алгоритмов глубинного обучения для анализа тональности текста и сравнение их эффективности с классическими (линейными и нелинейными) алгоритмами классификации текста.

В исследовании [3] представлены результаты эффективности применения различных архитектур свёрточных нейронных сетей, при использовании предварительно обученной модели векторного представления слов word2vec.

Результаты применения нейронных сетей с LSTM-блоками и их модификациями были представлены в исследовании [4], а также было показано, что данная архитектура является наиболее эффективной для построения бинарного и многоклассового классификаторов для анализа тональности текстов.

Также в работе [5] утверждается, что можно значительно улучшить эффективность классификатора, используя предварительно обученные модели векторных представлений слов (например, Word2Vec).

В данной работе для построения бинарного классификатора будут использоваться рецензии, состоящие из коротких предложений.

В настоящее время задача построения классификатора (бинарного или многоклассового) для определения тональности текста решается с помощью нейросетевых моделей, так как эффективность архитектур, использованных в упомянутых работах, значительно выше, чем у классических линейных алгоритмов.

Существует множество библиотек для реализаций алгоритмов машинного обучения. Существуют два типа фреймворков: символьные и императивные. В символьных фреймворках гораздо больше возможностей использовать память многократно, а оптимизация на основе графов зависимостей осуществляется автоматически. Самыми популярными символьными (symbolic) фреймворками в настоящее время являются TensorFlow и Theano.

В отличие от Theano, Tensorflow не ориентирован только на обуче-

ние нейронных сетей, поэтому можно использовать коллекции графов и очереди в качестве составных частей для высокоуровневых компонентов. Если необходимо обучать масштабные модели и использовать много внешней памяти, то Theano будет очень медленно работать из-за необходимости компиляции кода C/CUDA в бинарный код.

TensorFlow имеет прозрачную модульную архитектуру с множеством фронт-эндов. В архитектуре Theano разобраться довольно непросто: весь код - это Python, где код C/CUDA упакован как строка Python. В таком коде сложно ориентироваться, его непросто отлаживать и проводить рефакторинг. Более того, визуализация графов в TensorFlow реализована значительно эффективнее, чем в Theano [6].

Векторные представления слов для линейных алгоритмов будут представлены двумя моделями: Word2Vec и мешок слов (bag of words), а для классификаторов на основе нейронных сетей будет использована только модель Word2Vec. С помощью инструмента для построения векторных моделей Gensim будет обучена модель Word2Vec. С использованием TensorFlow будут реализованы свёрточная нейронная сеть и рекуррентная нейронная сеть с LSTM-блоками.

Аналитический обзор

Искусственные нейронные сети и их составляющие

Искусственные нейронные сети были построены по принципу биологических нейронных сетей, которые представляют собой сети нервных клеток, выполняющие определенные физиологические функции. Составным элементом нейронных сетей являются нейроны (представлены на рис. 3.11).

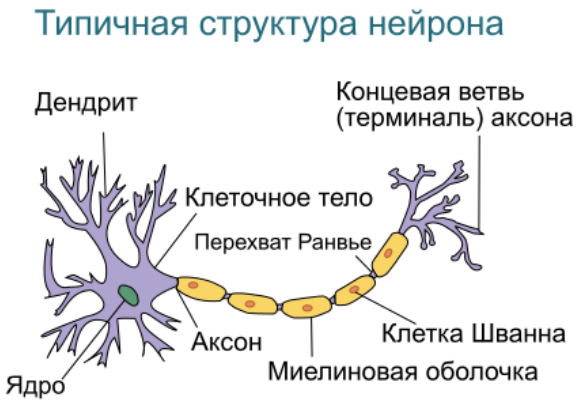

Рисунок 3.1 — Типичная структура нейрона

У нейрона есть несколько функций:

- Приёмная функция: синапсы получают информацию;
- Интегративная функция: на выходе нейрона сигнал, который несёт информацию обо всех суммированых в нейроне сигналах;
- Проводниковая функция: по аксону проходит информация к синапсу;
- Передающая функция: импульс, достигший окончания аксона, заставляет медиатор передавать возбуждение следующему нейрону.

Синапсами называют связи, по которым выходные сигналы одних нейронов поступают на входы других. Каждая связь характеризуется своим весом. Связи с положительным весом называются возбуждающими, а с отрицательным — тормозящими. Выход нейрона называется аксоном. В искусственной нейронной сети искусственный нейрон — это некоторая нелинейная функция, аргументом которой является линейная комбинация всех входных сигналов. Такая функция называется активационной. Затем результат активационной функции посылается на выход нейрона. Объединяя

такие нейроны с другими, получают искусственную нейронную сеть.

Активационная функция

Функция активации нейрона характеризует зависимость сигнала на выходе нейрона от суммы сигналов на его входах. Обычно функция вляется монотонно возрастающей и находится в области значений [-1,1] (гиперболический тангенс) и [0,1] (сигмоида). Для некоторых алгоритмов обучения необходимо, чтобы активационная функция была непрерывно дифференцируемой на всей числовой оси. Искусственный нейрон характеризуется своей активационной функцией (например, название "сигмоидальный нейрон").

Основными активационными функциями являются:

- Пороговая активационная функция (функция Хевисайда). Нельзя использовать для алгоритма обратного распространения ошибки;

$$f(x) = \begin{cases} 1 \text{ if } x \geq -w_0 x_0 \\ 0 \text{ else} \end{cases} \tag{3.1}$$

- Сигмоидальная активационная функция;

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

- Гиперболический тангенс.

$$tanh(Ax) = \frac{e^{Ax} - e^{-Ax}}{e^{Ax} + e^{-Ax}} \tag{3.3}$$

Перцептрон

Перцептрон — тип искусственного нейрона, разрабатываемый Фрэнком Розенблаттом в 1950-ых и 1960-ых годах. В современных работах чаще всего используют другую модель искуственного нейрона — сигмоидальный нейрон. Чтобы понять, как работает сигмоидальный нейрон, необходимо рассмотреть структуру и принцип работы перцептрона. Перцептрон принимает на вход значения $x_1, x_2...$ и выдаёт бинарный результат (см. рис. 3.2).

Розенблатт предложил использовать веса — числа, выражающие важность вклада каждого входа в конечный результат. Взвешенная сумма (или

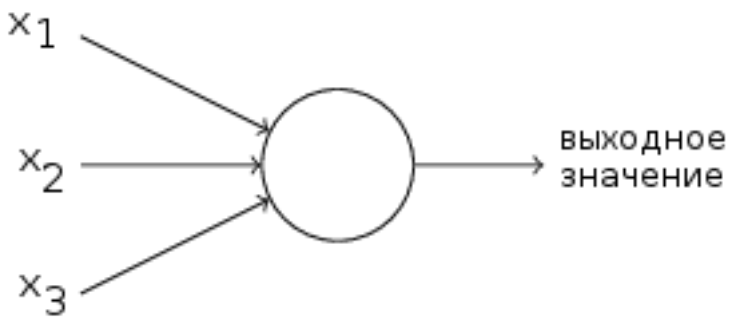

Рисунок 3.2 — Схема перцептрона

веса) сравнивается с пороговым значением (threshold), и по результатам определяется, будет ли выдан 0 или 1. Пороговое значение также является параметром нейрона.

$$\begin{cases} 0 \text{ if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 \text{ if } \sum_j w_j x_j > \text{ threshold} \end{cases} \tag{3.4}$$

Перцептроны могут быть классифицированы как искусственные нейронные сети

- с одним скрытым слоем;
- с пороговой активационной функцией;
- с прямым распространением сигнала.

Обучение перцептрона состоит в изменении матрицы весов. Существуют 4 исторически сложившихся видов перцептронов:

- Перцептрон с одним скрытым слоем;
- Однослойный перцептрон: входные элементы напрямую соединены с выходными с помощью системы весов. Является простейшей сетью прямого распространения (feedforward network);
- Многослойный перцептрон (по Розенблатту): присутствуют дополнительные скрытые слои;
- Многослойный перцептрон (по Румельхарту): присутствуют дополнительные скрытые слои, а обучение проводится по методу обратного распространения ошибки (backpropagation algorithm).

Если бы небольшое изменение весов (или смещения) вызывало небольшое же изменение на выходе сети, то желаемое поведение нейронной сети можно было бы получить с помощью простых модификаций смещений и

весов в процессе обучения. Однако обучение не так просто осуществить, если нейронная сеть состоит из перцептронов. Небольшое изменение весов или смещения одного из перцептронов сети может кардинально изменить выходное значение перцептрона, например, с 0 на 1. Поэтому самое незначительное изменение значений одного из элементов сети может создать значительные трудности в понимании изменения поведения сети. Поскольку задача обучения нейронной сети является задачей поиска минимума функции ошибки в пространстве состояний обучения, то для ее решения могут применяться стандарные методы теории оптимизации. Для однослойного перцептрона с n входами и m выходами речь идет о поиске минимума в nm-мерном пространстве.

Сигмоидальный нейрон

Сигмоидальные нейроны похожи на перцептроны, однако небольшие изменения в их весах и смещениях незначительно изменяют выход нейрона. Этот факт позволяет сети из сигмоидальных нейронов обучаться. На вход сигмоидального нейрона подаются любые значения между 0 и 1. На выходе также выдаётся значение между 0 и 1, так как в качестве активационной функции используется сигмоида, являющаяся нелинейной (см. рис. 3.3).

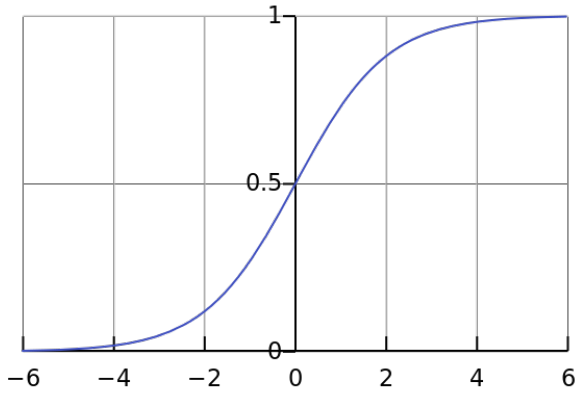

Рисунок 3.3 — График логистической кривой сигмоиды

$$f(x) = \frac{1}{1 + e^{-\beta x}} \tag{3.5}$$

Чем больше β (параметр наклона сигмоидальной функции активации),тем сильнее крутизна графика. При $\beta \to \infty$ сигмоида стремится к функции Хевисайда.

Важным свойствои сигмоидальной функции является её дифференцируемость. Применение непрерывной функции активации позволяет использовать при обучении градиентные методы.

Нейроны можно разделить на группы в зависимости от их положения в сети:

- входные нейроны принимают исходный вектор данных;
- в промежуточных нейронах происходят основные вычислительные операции — обучение;
- выходные нейроны — результат работы сети.

Архитектура нейронных сетей

Рассмотрим задачу обучения с учителем. Дано множество тренировочных примеров X с метками (labels) Y. Нейронные сети определяют нелинейную гипотезу $h_{W,b}(x)$ с параметрами W и b. Нейронная сеть составлена из множества простых нейронов так, что выход одного из нейронов будет входом другого (см. рис. 3.4). Крайний левый слой называется входным,

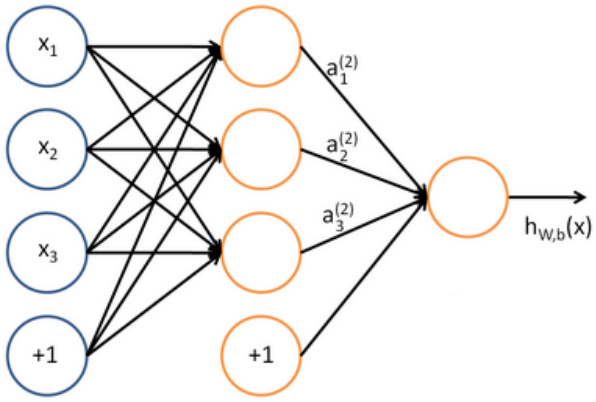

Рисунок 3.4 — Схема простейшей сети прямого распространения

а крайний правый слой — выходным. Слой посередине является скрытым и называется так из-за того что его значения не наблюдаются в тренировочных примерах. Таким образом в данной сети три элемента входа, три скрытых элемента и один выходной элемент. Пусть n_l — количество слоёв в сети (в данном случае 3). Параметры сети (W, b) = $(W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$. Результат применения функции активации (выхода) обозначается a_i для i-ого элемента. Получаем такую систему:

$$\left\{a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_{(1)}^1), a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b \right. \tag{3.6}$$

Обозначив функцию суммирования через z, получим в векторной форме:

$$\begin{cases} z^{(2)} = W^{(1)}x + b^{(1)} \\ a^{(2)} = f(z^{(2)}) \\ z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \\ h = f(z^{(3)}) \end{cases} \tag{3.7}$$

Общая формула будет выглядеть таким образом:

$$\begin{cases} z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)} \\ a^{(l+1)} = f(z^{(l+1)}) \end{cases} \tag{3.8}$$

Сетью прямого распространения называются нейронные сети, которые используют выход одного слоя в качестве входных данных для следующего слоя.

Обучение нейронных сетей

Общие понятия в обучении нейронных сетей

Эпоха – прямой и обратный проход по всем тренировочным примерам.

Размер серии (batch) – количество тренировочных примеров для одной итерации прямого и обратного проходов.

Количество итераций – количество проходов: каждый проход использует примеры (batch). Один проход = прямой проход + обратный проход.

То есть имея 1000 примеров, batch = 500, нам потребуется две итерации, чтобы завершить одну эпоху.

С математической точки зрения, обучение нейронных сетей – многопараметрическая задача нелинейной оптимизации.

Алгоритм обратного распространения ошибок

Алгоритм обратного распространения ошибки определяет стратегию подбора весов многослойной сети с применением градиентных методов оп-

тимизации. Поскольку целевая функция, обычно определяемая как квадратичная разность суммы между фактическими и ожидаемыми выходными значениями, является непрерывной, градиентные методы оптимизации являются эффективными при обучении сети. При обучении многослойной нейронной сети необходимо вычислить вектор градиента относительно параметров всех слоёв сети. Пусть имеется конечный набор тренировочных данных (m примеров). Для обучения нейронной сети применяют пакетный градиентный спуск (batch gradient descent). Квадратичная ошибка целевой функции (squared-error cost function) для одного примера будет вычислена по формуле:

$$J(W, b; x, y) = \frac{1}{2}||h_{W,b}(x) - y||^2 \tag{3.9}$$

Тогда целевая функция для m примеров будет выглядеть так:

$$\begin{aligned} J(W, b) &= [\frac{1}{m}\sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)})] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2 \\ &= [\frac{1}{m}\sum_{i=1}^{m}(\frac{1}{2}||h_{W,b}(x^{(i)}) - y^{(i)}||^2)] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2 \end{aligned} \tag{3.10}$$

Первый член выражения J(W,b) это сумма квадратов ошибок, второй — член регуляризации (L2 — уменьшение весов — weight decay), позволяющий уменьшить значения весов и предотвратить переобучение. Параметр регуляризации весов λ используют для проверки относительной значимости частей данного выражения. В задачах бинарной классификации y представлен 0 или 1 (так как сигмоидная функция выдает значение в пределах [0;1]; однако при использовании гиперболического тангенса лейблами классов были бы -1 и 1). Задача — минимизировать J(W,b). Для обучения нейронной сети необходимо инициализировать каждый параметр $W_{ij}^{(l)}$ и $b_i^{(l)}$ малыми случайными величинами, близкими к нулю (например случайное распределение $(0; \epsilon^2)$), где $\epsilon = 0.01$), а затем применить алгоритм оптимизации (упоминавшийся выше градиентный спуск).

Так как J(W, b) не является выпуклой функцией, то градиентный спуск восприимчив к локальным оптимумам. Каждая итерация градиентного спуска обновляет параметры таким образом:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$$
$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W,b), \tag{3.11}$$

где α — скорость обучения (learning rate).

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b; x^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)} \tag{3.12}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W,b; x^{(i)}, y^{(i)}) \tag{3.13}$$

Шаги алгоритма обратного распространения ошибки

1) Осуществляется прямой проход по сети, вычисляются активации слоёв L2, L3 и так далее до выходного слоя L_{n_l};

2) Для каждого выходного элемента i в выходном слое n_l (the output layer) рассчитывается ошибка

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \ \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)}) \tag{3.14}$$

3) Для $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$:

Для каждого элемента в слое l, рассчитывается

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)}) \tag{3.15}$$

4) Вычисляются частные производные

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b; x,y) = a_j^{(l)} \delta_i^{(l+1)} \tag{3.16}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b; x,y) = \delta_i^{(l+1)} \tag{3.17}$$

В матричной форме алгоритм будет записан таким образом (• обозначено произведение Адамара — покомпонентное произведение двух матриц):

1) Осуществляется прямой проход по сети, вычисляются активации слоёв L2, L3 и так далее до выходного слоя L_{n_l}.

2) Матрица ошибок для выходного слоя n_l

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)}) \tag{3.18}$$

3) Для слоя $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)}) \tag{3.19}$$

4) Вычисление частных производных

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T, \tag{3.20}$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)} \tag{3.21}$$

Псевдокод

- $\Delta W^{(l)} = 0$,$\Delta b^{(l)} = 0$ (матрица или вектор нулей) для всех l;
- For i = 1 to m
  - Использовать алгоритм обратного распространения ошибки, чтобы вычислить $\nabla_{W^{(l)}} J(W, b; x, y)$ and $\nabla_{b^{(l)}} J(W, b; x, y)$.
  - $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$.
  - $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$.
- Обновление параметров

$$W^{(l)} = W^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta W^{(l)}\right) + \lambda W^{(l)}\right] \tag{3.22}$$

$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)}\right] \tag{3.23}$$

Для обучения нейронной сети, нужно повторно выполнить шаги градиентного спуска, чтобы уменьшить значение целевой функции $J(W, b)$.

Недостатки градиентного спуска

Основная трудность обучения нейронных сетей состоит в методах выхода из локальных минимумов. Недостатками градиентного спуска при обучении сети являются:

- Паралич сети

  Значения весов сети в результате коррекции могут стать очень большими величинами. Поскольку ошибка, посылаемая обратно в процессе обучения, пропорциональна производной сжимающей функции, то процесс обучения может почти остановиться. Это можно предотвратить, уменьшая шаг η, одна процесс обучения будет происходить дольше.

- Размер шага

  Если значение шага не изменяется и оно довольно мало, то метод сходиться слишком медленно. Если же шаг слишком велик, то может возникнуть паралич сети. Необходимо изменять значение шага: увеличивать до тех пор, пока не прекратится улучшение оценки в направление антиградиента и уменьшать, если оценка не улучшается.

Сравнение стохастического и пакетного градиентных спусков

Если для пакетного градиентного спуска функция потерь вычисляется для всех образцов вместе взятых после окончания эпохи, а потом изменяются весовые коэффициенты нейронов, то для стохастического метода весовые коэффициенты изменяются после вычисления выхода сети на одном из обучабщих примеров. Недостатком пакетного градиентного спуска является его “застревание” в локальных минимумах. Несмотря на то, что стохастический метод работает медленнее пакетного, он способен выходить из локальных минимумов, что приводит к лучшим результатам обучения сети (стохастический метод использует недовычисленный градиент).

Мониторинг состояния сети

Функция перекрёстной энтропии в качестве целевой функции

Функция перекрёстной энтропии используется в качестве функции потерь: y_i' — предсказанные значения, y_i — верные значения.

$$L(x, y) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log a(x^{(i)}) + (1 - y^{(i)}) \log\Big(1 - a(x^{(i)})\Big) \tag{3.24}$$

Техники регуляризации

- L1-регуляризация: происходит изменение нерегуляризованной целевой функции путём добавления суммы абсолютных значений весов:

$$J = J_0 + \frac{\lambda}{n}\sum_{W} |W| \tag{3.25}$$

  При использовании L1-регуляризации происходит стремление одного или более весовых значений к 0.0, поэтому соответствующая функция (feature) больше не требуется. Этот эффект называется селекцией функций (feature selection);
- L2-регуляризация (также известная как weight decay) В отличии от L1-регуляризации, в L2 веса уменьшаются на величину, пропорциональную весам:

$$J = J_0 + \frac{\lambda}{2n}\sum_{W} W^2 \tag{3.26}$$

- Dropout не влияет на значение целевой функции: изменяется структура сети. Каждый нейрон удаляется из сети с некоторой вероятностью p. По полученной прореженной сети делается обратное распространение ошибки, для оставшихся весов делается градиентный шаг. После этого удалённые нейроны восстанавливают в сети. При обучении нейросети выход каждого нейрона домножается на (1-p). Так будет получено матожидание ответа сети по её $2\hat{\text{N}}$ (где N — количество нейронов в сети) архитектурам. Обученная таким образом сеть является результатом усреднения $2\hat{\text{N}}$ сетей. Отдельная нейронная сеть, обученная при помощи раннего останова, имеет слишком большую

ошибку, однако усреднение нескольких нейронных сетей приводит к существенному снижению ошибки;
- Искусственное расширение данных для обучения.

Гиперпараметры сети

- Темп обучения: сначала необходимо оценить пороговое значение для η, в котором значение целевой функции мгновенно начинает снижаться без колебаний. Снача значение оценки устанавливается $\eta = 0.01$. Если значение целевой функций снижается во время первых эпох, то нужно увеличивать темп обучения, пока будет не найдено значение колебания целевой функции. Если же при начальном темпе обучения значения целевой функции колеблются, то необходимо его уменьшать. Темп обучения регулирует размер шага в градиентном спуске и наблюдает за значениями целевой функции, определяя, был ли размер шага градиентного спуска слишком большим;
- Использовать раннюю остановку (early stopping) для определения размера эпох обучения: ранняя остановка значит, что в конце каждой эпохи нужно вычислить точность классификации на данных проверки (validation set). Когда улучшение точности прекратится, остановить процесс обучения. Такая остановка также предотвращает переобучение;
- График обучения: идея — сохранять темп обучения неизменным до момента, когда точность данных проверки не начнёт ухудшаться. Тогда необходимо уменьшить темп обучения (уменьшив, например, в 10 раз).
- Параметр регуляризации λ: после определения η, можно начать с λ=1.0 и затем увеливать или уменьшать значение (в 10 раз);
- Размер пакетов: если размер пакетов слишком мал, невозможно полностью использовать преимущества хороших матричных библиотек, оптимизированных для быстрого оборудования. Если же размер пакетов слишком велик, то веса сети будут обновлять очень нечасто. Необходимо выбрать компромиссное значение, которое максимизирует скорость обучения.

Глубокие нейронные сети

Обзор

Глубокими нейронными сетями называются такие сети, в которых есть несколько скрытых слоев. Поскольку каждый скрытый слой вычисляет нелинейное преобразование предыдущего слоя, глубокая сеть может иметь значительно большую репрезентативную мощность (то есть может представлять значительно более сложные функции), чем малослойная. При обучении глубокой сети важно использовать нелинейную функцию активации в каждом скрытом слое. Это связано с тем, что множество слоев линейных функций сами вычисляли бы только линейную функцию ввода и, следовательно, не были бы более выразительными, чем использование только одного скрытого слоя.

Главным достоинством глубинных сетей является сжатое представление достаточного большого множества функций. Можно показать, что существуют функции, которые k-слойная сеть может представлять сжато, а (k-1)-слойная сеть не может этого сделать, если только она не имеет экспоненциально большое количество элементов в скрытых слоях.

Доступность данных

С помощью метода, описанного выше, можно полагаться только на маркированные данные для обучения. Однако помеченных данных часто бывают недостаточно, и, следовательно, для многих задач трудно получить достаточное количество примеров для соответствия параметрам сложной модели. Например, учитывая высокую степень выразительности глубинных сетей, обучение при небольшом количестве данных приведет к переобучению.

Локальный оптимум

Обучение малослойной сети (с 1 скрытым слоем) с использованием контролируемого обучения обычно приводит к сближению параметров с подходящими значениями. Но при обучении глубокой сети, это работает намного реже. В частности, обучение нейронной сети с использованием обу-

чения с учителем включает в себя решение проблемы с невыпуклой оптимизацией (например, минимизация ошибки обучения $\sum_i ||h_W(x^{(i)}) - y^{(i)}||^2$ в зависимости от сетевых параметров W). В глубокой сети появляется большое количество локальных оптимумов, поэтому обучение с градиентным спуском перестаёт работать.

Градиентная диффузия

При использовании метода обратного распространения ошибки для вычисления производных, градиенты, которые распространяются от выходного слоя до более ранних слоев сети, быстро уменьшаются по мере увеличения глубины сети. В результате производная от общей стоимости по отношению к весам в более ранних слоях очень мала. Таким образом, при использовании градиентного спуска веса ранних слоев медленно меняются и более ранние слои не могут многому научиться. Эту проблему часто называют “диффузией градиентов” (diffusion of gradients).

Проблемы обучения глубоких сетей и их решения

Исчезающий градиент

Проблема исчезающего градиента — это трудность, возникающая при обучении искусственных нейронных сетей с использованием методов обучения на основе градиента и обратного распространения ошибки. В таких методах каждый вес нейронной сети обновляется пропорционально градиенту функции ошибки относительно текущего веса на каждой итерации обучения. Стандартные функции активации, такие как гиперболический тангенс, имеют градиенты в диапазоне (-1, 1), а метод обратного распространения ошибки вычисляет их по цепному правилу. После умножения этих чисел для вычисления градиентов “фронтальных” слоев в n-слойной сети, что означает, что градиент (сигнал ошибки) экспоненциально уменьшается вместе с n, а передние слои обучаются очень медленно.

Когда используются функции активации, производные которых могут принимать большие значения, есть риск столкнуться с exploding gradient problem. Возможными решениями являются:

- Многоуровневая иерархия: слой сети предварительно обучается, ис-

пользуя методы обучения без учителя, а затем его значение регулируется с помощью метода обратного распространения ошибки. Таким образом каждый слой сети изучает сжатое представление наблюдений, которое подается на следующий слой [7];

- Долгая краткосрочная память: разновидность архитектуры рекуррентных нейронных сетей. Когда величины ошибки распространяются в обратном направлении от выходного слоя, ошибка не выпускается из памяти LSTM-блока. Она непрерывно передаётся обратно каждому из вентилей, пока они не будут обучены отбрасывать подобные значения [8];
- Остаточные сети (Residual networks): один из наиболее эффективных методов решения проблемы исчезающего градиента является использование остаточных нейронных сетей (ResNets). Более глубокая сеть будет иметь более высокую ошибку обучения, чем малослойная сеть. Команда Microsoft Research обнаружила, что разделение глубокой сети на части (скажем, каждый часть представляет собой три слоя сети) и передача входных данных в каждый фрагмент до следующего фрагмента (наряду с остаточным выходом Из куска минус входные данные вновь введённого фрагмента) помогли устранить большую часть этой проблемы с исчезновением градиента. Никаких дополнительных параметров или изменений в алгоритме обучения не требуется. ResNets показали более низкую ошибку обучения (и тестовую ошибку), чем их более малослойные аналоги, путем повторного ввода выходов из более мелких слоев в сети для компенсации исчезающих данных.

Сигмоидальные активационные функции

Использование сигмоидальных активационных функций может вызвать проблемы в обучении глубинных сетей, а именно значения активаций в конечном слое будут близки к нулю на ранних этапах обучения, замедляя этот процесс. Были предложены альтернативные активационные функции [9], которые не так страдают от ограничения.

Выбор подходящих весов

Выбор подходящих весов и momentum schedule в импульсном стохастическом градиентном спуске (momentum-based stochastic gradient descent) существенно влияют на способность обучать глубокие сети [10].

Свёрточные нейронные сети

Обзор

Свёртка является операцией, которая применяется к двум последовательностям f и g и порождает третью последовательность.

$$(f * g)(c) = \sum_{a} f(a)g(c-a), \text{где } a = b + c \tag{3.27}$$

Формула для двумерной свёртки:

$$(f * g)(c_1, c_2) = \sum_{a_1, a_2} f(a_1, a_2)g(c_1 - a_1, c_2 - a_2) \tag{3.28}$$

Рассмотрим одномерный свёрточный слой с входами x_n и выходами y_n (см. рис. 3.5). Тогда функция для выходов будет представлена следующим образом:

$$y_n = A(x_n, x_{n+1}, \dots) \tag{3.29}$$

В свёрточном слое находится множества копий одного и того же нейрона,

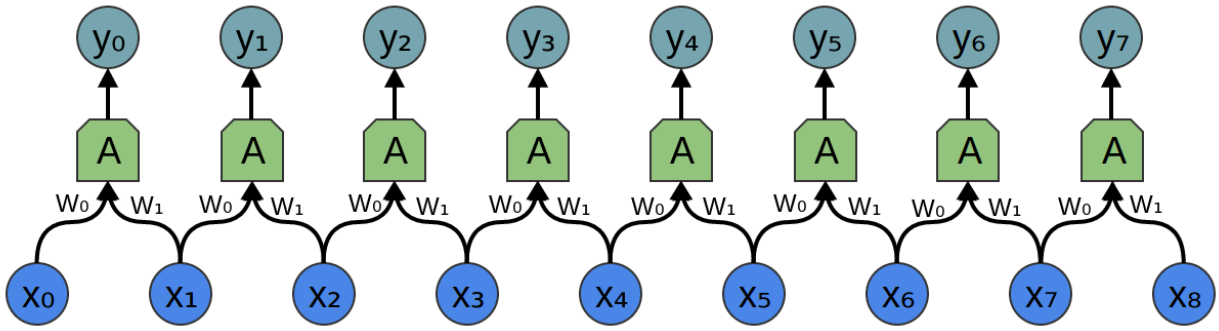

Рисунок 3.5 — Пример одномерного свёрточного слоя

поэтому многие веса появляются в нескольких позициях.

$$y_0 = \sigma(W_0 x_0 + W_1 x_1 - b) \tag{3.30}$$

$$y_1 = \sigma(W_0 x_1 + W_1 x_2 - b) \tag{3.31}$$

Стандартная матрица весов соединяет каждый вход с каждым нейроном с разными весами. Матрица для свёрточного слоя отличается тем, что различные веса могут появляться на нескольких позициях, а поскольку нейроны не соединены со всеми возможными входами, матрица содержит множество нулевых элементов:

$$M = \begin{bmatrix} w_0 & w_1 & 0 & \dots \\ 0 & w_0 & w_1 & \dots \\ 0 & 0 & w_0 & \dots \end{bmatrix} \tag{3.32}$$

То есть умножение на матрицу выше — то же самое, что и свёртка с $[\dots 0, w_1, w_0, 0 \dots]$. Ядро свёртки, скользящее по разным частям изображения, соответствует наличию нейронов в этих частях.

Свёртку можно пояснить на примере обработки изображений. Если представить, что изображения – двумерные функции, то различные преобразования изображений не что иное, как свёртка функции изображения с локальной функцией, которая называется ядром свёртки.

Каждый новый пиксель изображения представляет собой взвешенную сумму пикселей, которые ядро прошло к этому моменту времени. Двумерный свёрточный слой представлен на рис. 3.6.

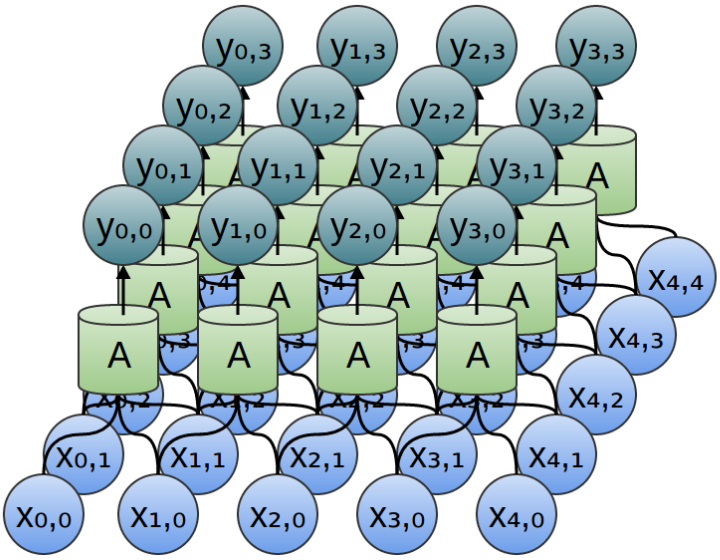

Рисунок 3.6 — Двумерный свёрточный слой

Свёрточная нейронная сеть — архитектура нейронных сетей, изначально созданная и использованная для эффективного распознавания изоб-

ражений: чередуются свёрточные слои (convolutions) с нелинейными активационными функциями (ReLU или гиперболический тангенс tanh) и слои объединения (pooling layers).

В отличии от сети прямого распространения, где каждый входной нейрон соединяется с выходным нейроном в следующем слое, в свёрточных сетях для получения выходных значений используются свёртки над каждым входным слоем. В операции свёртки используется матрица весов небольшого размера, которая сдвигается по всему обрабатываемому слою, формируя после каждого сдвига сигнал активации для нейрона следующего слоя с аналогичной позицией. Эта матрица называется ядром свёртки; она используется для различных нейронов выходного слоя.

При выполнении операции свёртки каждый фрагмент(например, изображения) поэлементно умножается на матрицу свёртки, а результат суммируется и записывается в аналогичную позицию выходного изображения. Матрицу свёртки представляет собой графическое кодирование какого-либо признака. Получившийся в результате операции свёртки следующий слой показывает наличие данного признака. В свёрточной нейронной сети существуют много наборов весов, которые кодируют элементы изображений. Ядра свёртки формируются в процессе обучения сети. При проходе каждым набором весов формируется карта признаков. Поскольку появляется много независимых карт признаков на одном слое, то сеть становится многоканальной.

В каждом слое свёртки для каждого канала свой фильтр, ядро свёртки которого обрабатывает предыдущий слой по фрагментам. Результат применения различных фильтров объединяется. Так получаются слои объединения. Операция субдискретизации выполняет уменьшение размерности сформированных картпризнаков. В данной архитектуре сети считается, что информация о факте наличия искомого признака важнее точного знания его координат, поэтому из нескольких соседних нейронов карты признаков выбирается максимальный и принимается за один нейрон уплотнённой карты признаков меньшей размерности. За счёт данной операции, помимо ускорения дальнейших вычислений, сеть становится инвариантной к масштабу входного изображения.

После начального слоя сигнал проходит серию свёрточных слоёв, в которых чередуется операции свёртки и объединения(pooling). Чередование слоёв позволяет составлять карты признаков: на каждом следующем слое карта уменьшается в размере, а количество каналов увеличивается. Практически это означает способность распознавания сложных иерархий признаков.

После прохождения нескольких слоев карта признаков вырождается в вектор или скаляр, но таких карт признаков становится сотни. На выходе свёрточных слоёв сети дополнительно устанавливают несколько слоев полносвязной нейронной сети (например, перцептрон), на вход которому подаются конечные карты признаков.

Гиперпараметры сети

Гиперапараметрами свёрточной нейронной сети являются:

- Узкая и широкая свёртки (wide and narrow convolutions): дополнение нулями (zero padding) позволяет сделать свёртку широкой в случае, когда, например, первый элемент матрицы не имеет соседних элементов слева и сверху. Без использования дополнения нулями получаем узкую свёртку;
- Размер шага (stride): Размер шага определяет величину сдвига фильтра на каждом шаге. Чем больше шан, тем меньше фильтр применяется и тем меньше размер выходной матрицы. Обычно использует шаг равный единице, однако больший шаг может позволить построить модель, поведение которой будет напоминать рекурсивную нейронную сеть (т.е. свёрточная сеть с большим шагом будет выглядеть как дерево);
- Слои объединения: Слои объединения помогают сократить размерность выходной информации, при этом сохраняя самую заметную информацию. Например, если фильтр определяет, содержит ли предложение отрицание ("not good"). Если где-то в предложении есть эта фраза, то результат применения фильтра к этому региону даст большое значение, но малое для других регионов. После применения операции максимума для региона, остается только информация, появля-

лось ли заявленное отрицание в предложение, однако информация о том, где оно появлялось, исчезает. То есть информация о местоположении пропадает, а локальная информация остаётся (очевидно, что “not good” сильно отличается от “good not”);

- Каналы (channels): каналы — это разные “взгляды” на входные данные. Например, в распознавании изображений, у нас обычно три канала — RGB. В обработке естественного языка такими каналами могут являться различные векторные представления слов (word2vec или GloVe), предложение на разных языках или перефразированные предложения.

Типовая структура

Структура сети является однонаправленной, а для обучения обычно используется метод обратного распространения ошибки. Сеть состоит из большого количества слоёв (см. рис. 3.7).

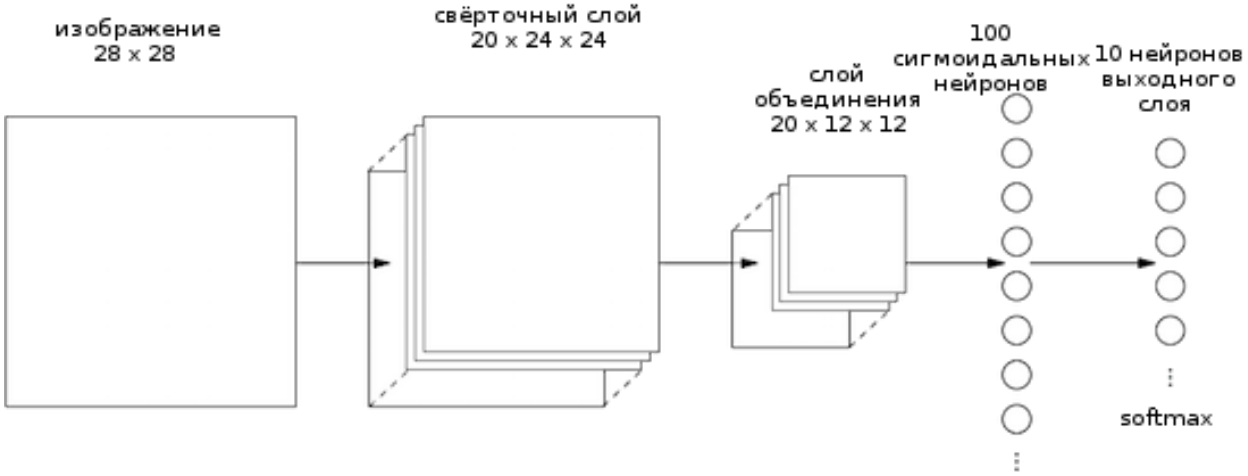

Рисунок 3.7 — Пример архитектуры свёрточной нейронной сети для распознавания объекта на изображении размерности 28x28 px

Слой свёртки (convolutional layer)

Основной компонент свёрточной нейронной сети. Слой свёртки включает в себя свой фильтр для каждого канала. Ядро свёртки фильтра обрабатывает предыдущий слой по фрагментам. Весовые коэффициенты ядра свёртки определяются в процессе обучения сети.

Усеченное линейное преобразование(rectified linear unit)

Также переводимый как блок линейной ректификации. Слой ReLU представляет является функцией активации после свёрточного слоя. Функция $f(x) = max(0, x)$ выбирается вместо сигмоиды или гиперболического тангенса, поскольку показывает хорошие результаты при обучении нейронных сетей и "отсекает" ненужные детали в канале. Ректификатор, приближение ректификтора, также называемый softplus, и производная softplus — логистическая функция.

$$f(x) = max(0, x) \tag{3.33}$$

$$f(x) = \sum_{i=1}^{inf} \sigma(x - i + 0.5) \approx \log(1 + e^x) \tag{3.34}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.35}$$

$$y_{i,j,d} = max\{0, x^l_{i,j,d}\}, \tag{3.36}$$

где $0 \leq i < H^l = H^{l+1}, 0 \leq j < W^l = W^{l+1}, 0 \leq d < D^l = D^{l+1}$.

Пулинг или слой объединения

Слой пулинга — это нелинейное уплотнение карты признаков. Использование пулинга позволяет существенно уменьшить пространственный объём изображения. Операция объединения интерпретируется следующим образом: если на предыдущей операции свёртки уже были выявлены некоторые признаки, то для дальнейшей обработки изображение уплотняется до менее подробного. Фильтрация ненужных деталей помогает сети не переобучаться.

Обычно слой пулинга вставляется после слоя свёртки перед слоем следующей свёртки: для реализации обычно используется функция максимума.

Можно также применить L2-pooling: вместо выбора максимальной активации из карты признаков, нужно взять квадратный корень от суммы квадратов активаций этой карты признаков.

Полносвязная нейронная сеть

После нескольких свёрток и уплотнения изображения с помощью пулинга из конкретной сетки пикселей с высоким разрешением получаются более абстрактные карты признаков: на каждом следующем слое увеличивается число каналов, при этом размерность изображения в каждом канале уменьшается. Таким образом остаётся большой набор каналов, в которых хранится небольшое число данных. Эти данные интерпретируются как абстрактные признаки, выявленные из исходного изображения. Затем их объединяют и передают на обычную полносвязную нейронную сеть, которая также может быть многослойной. Поскольку полносвязные слои уже не соответствуют пространственной структуре пикселей, они обладают относительно небольшой размерностью.

Использование свёрточных нейронных сетей в анализе тональности текста

На вход нейронной сети будет подаваться матрица, количество строк которой зависит от размерности словаря, а ширина фильтров равна количеству столбцов этой матрицы (то есть используемой размерности для кодирования каждого слова). Высота (или размер фрагмента входных данных) может меняться, но обычно она составляет около 2—5 слов.

Первые слои представляют слова в виде низкоразмерных векторов. Следующий слой выполняет свёртки над векторными представлениями слов, используя фильтры разных размеров (то есть они захватывают 3-5 слов одновременно). Затем производится пулинг (max-pool) над результатом свёртки. К полученному длинному вектору признаков добавляем регуляризацию (dropout в этом случае). Наконец, происходит классификация результата с помощью слоя softmax [3] (см. рис. 3.8).

В качестве входов задаются не только стандартные X и Y, но и вероятность того, что нейрон окажется в слое дропаута (дропаут задаётся только во время тренировки сети). Первый слой — слой представления слов в виде векторов word2vec — является таблицей преобразования (соответствия). Результат применения этого слоя — трёхмерный тензор размерности [None,sequence_length, embedding_size].

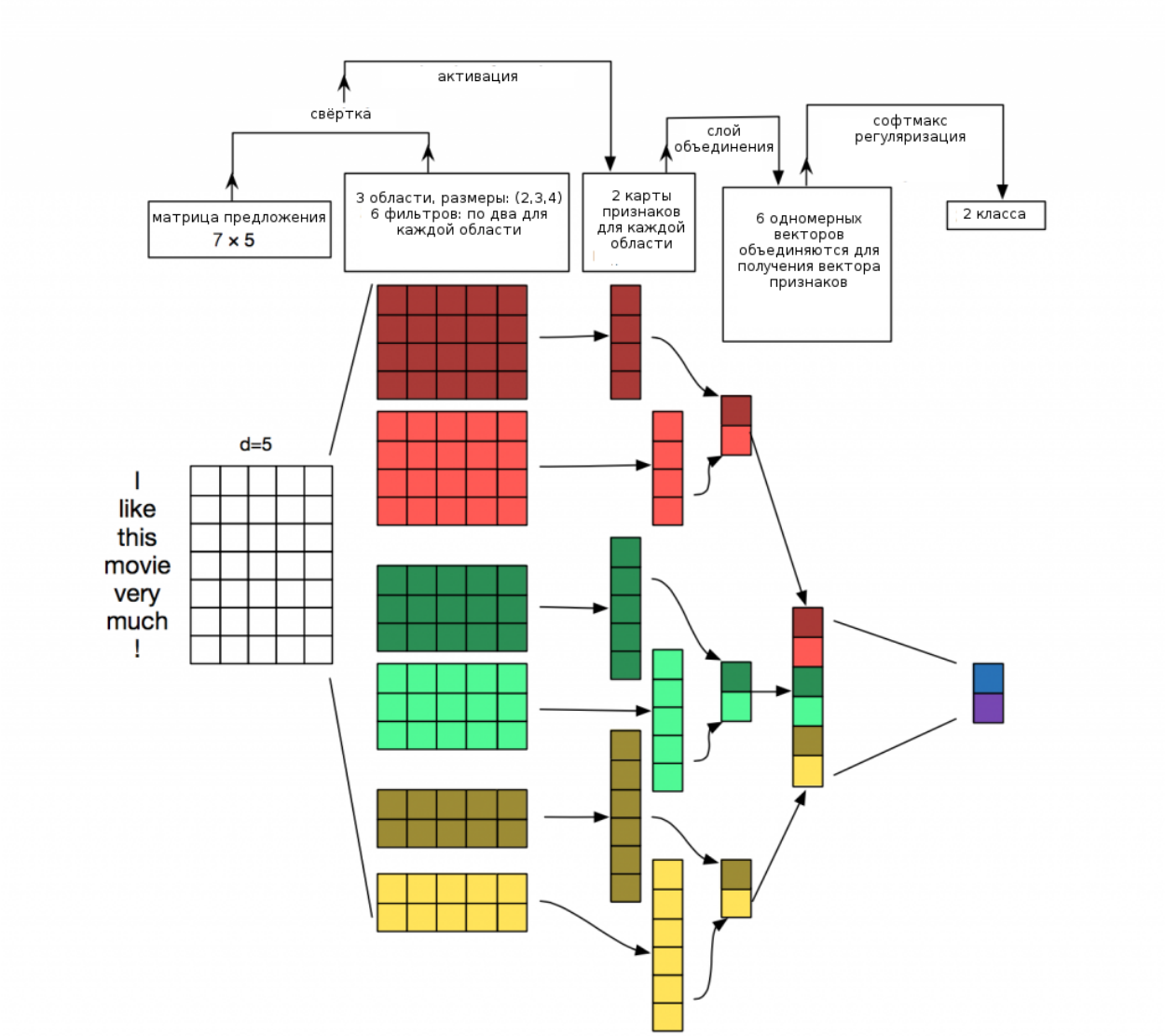

Рисунок 3.8 — Пример архитектуры свёрточной нейронной сети для классификации предложений

Следующий слой свёртки принимает на вход 4-мерный тензор (батч, ширина, высота и канал). К полученному на предыдущем шаге тензору просто прибавляется новое измерение [None, sequence_length, embedding_size, 1]. Каждый фильтр проходит полностью через всё представление, отличие только в том, сколько он обрабатывает слов. В данной работе используется узкая свёртка (narrow convolution) — поэтому на выходе тензор имеет форму [1, sequence_length - filter_size + 1, 1, 1].

После применения пулинга над выходом определенного фильтра получится тензор формы [batch_size, 1, 1, num_filters] По существу это и есть вектор признаков, последнее измерение которого соответствует нуж-

ным нам признакам. После получения всех тензоров, нужно объединить их в один длинный вектор признаков формы [batch_size, num_filters_total]

Самым популярным методом регуляризации свёрточных нейронных сетей является dropout. Dropout блокирует группу нейронов случайным образом. Это заставляет их "изучать"полезные признаки самостоятельно. Группа нейронов, которая принимает участие в обучении, определяется dropout_keep_prob входом в сети.

После получения обработанного вектора признаков, можно делать предсказания к какому классу относится данная рецензия. Необходимо перемножить матрицы и выбрать класс с наибольшим результатом. Слой softmax поможет нормализовать полученные вероятности.

Используя полученные значения, становится возможным определить функцию потерь. Наша цель — минимизировать потери, которые как раз измеряют ошибку (погрешность) нейронной сети. Для этого используется перекрёстная энтропия.

В работе используется функция softmax_cross_entropy_with_logits. Затем берётся среднее значение потерь.

Как классифицируются предложения с помощью данной архитектуры:

- Определить размерности трёх фрагментов данных: 2, 3 и 4, для каждого фрагмента существует два фильтра;
- Каждый фильтр выполняет свёртку над матрицей предложений и генерирует карты признаков (feature maps);
- Слой 1-max pooling обрабатывает каждую карту, то есть сохраняется наибольшее число из каждой карты. Полученный одномерный вектор признаков из карт объединяется в вектор признаков для предпоследнего слоя;
- Последний (softmax) слой получает этот вектор на вход и использует его для классификации предложения (бинарная классификация).

Благодаря использованию свёрточных слоев, количество параметров в них резко сокращается, и обучить сеть становится гораздо проще. А используя различные виды регуляризации (особенно dropout), получилось значительно уменьшить переобучение сети. Наконец, использование ReLU

вместо сигмоидальных нейронов помогло ускорить обучение.

Может показаться, что идея свёрточных нейронных сетей не очень применима для задач естественного языка: действительно, если на изображении соседние пиксели в основном являются частью одного и того же объекта, то это неверно в случае слов в предложении (части фраз могут быть разделены другими словами). Возможно, рекуррентные нейронные сети — более интуитивно понятная модель (предложение представлено в виде дерева разбора), однако это не значит, что свёрточные сети совсем не применимы для поставленных в работе задач.

Рекуррентные нейронные сети

Обзор

В сетях прямого распространения используется единственный вход, который полностью определяет активации всех нейронов в оставшихся слоях. Такую сеть невозможно обучить предсказывать события, например, в сюжете фильма — неясно, как бы могла быть использована информация о предыдущих событиях в фильме. Рекуррентные нейронные сети призваны решить эту проблему. Имея внутри циклы, RNN позволяет информации сохраняться: поведение скрытых нейронов будет определяться не только активацией в других скрытых слоях, но и полученными ранее активациями самих нейронов.

RNN может быть представлена в качестве множества копий одной и той же нейронной сети, где каждая копия передает сообщение следующей копии. То есть, имея цепеобразную структуру, как последовательсти или списки, RNN является естественной архитектурой нейронной сети, используемой для таких данных.

RNN способны обусловливать модель по всем предыдущим обработанным словам из корпуса текстов. На рис. 3.10 прямоугольник является скрытым слоем на временном шаге t. Каждый слой содержит нейроны (см. рис. 3.9), каждый из которых выполняет операцию линейной матрицы на своих входах, за которой следует нелинейная операция (например, tanh).

На каждом временном шаге выходные данные предыдущего шага вместе со следующим вектором слова x_t текста, представляет собой вход-

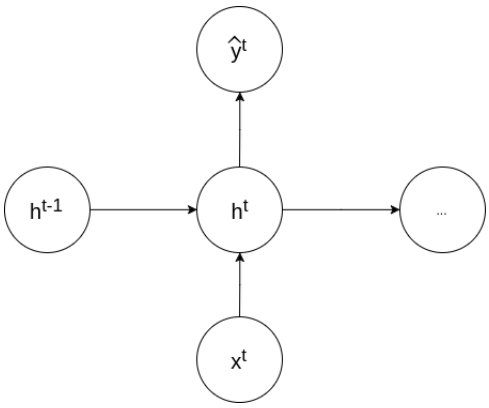

Рисунок 3.9 — Нейрон рекуррентной сети

ные данные для скрытого слоя для создания предсказания $\hat{y}_t$ и признаков h_t:

$$h_t = \sigma(W^{(hh)} h_{(t-1)} + W^{(hx)} x_t) \tag{3.37}$$

$$\hat{y}_t = \text{softmax}(W^{(S)} h_t) \tag{3.38}$$

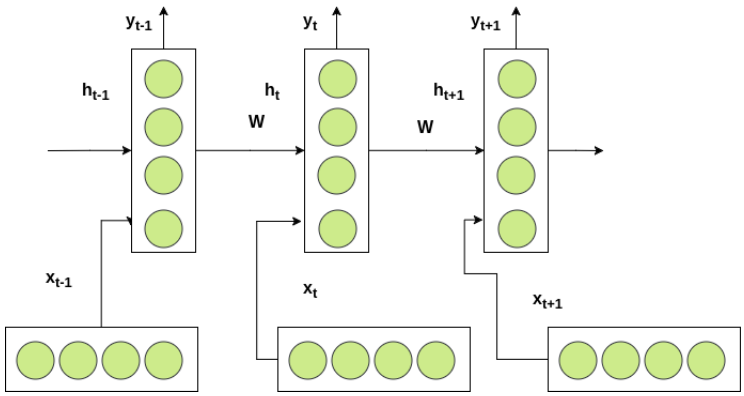

Рисунок 3.10 — Рекуррентная нейронная сеть, три временных шага

- $x_1, \ldots, x_t, x_{t+1}, \ldots, x_T$ — векторы слов; словарь состоит из T слов;
- формула h_t позволяет вычислить выходные признаки скрытого слоя на каждом временном шаге t:
  - $x_t \in \mathbb{R}^d$: слово, поданное на вход в момент времени t;
  - $W^{hx} \in \mathbb{R}^{D_h \times d}$: матрица весов для обусловливания входного слова x_t;
  - $W^{hh} \in \mathbb{R}^{D_h \times D_h}$: матрица весов, использованные для обусловливания выходных данных на предыдущем временном шаге h_{t-1};

- $h_{t-1} \in \mathbb{R}^{D_h}$: выход нелинейной функции на предыдущем временном шаге. h_0 — вектор инициализации для скрытого слоя в момент времени t = 0;
- σ — нелинейная функция.

- $\hat{y}_t$: выходное распределение вероятности по словарю на каждом временном шаге t. По сути $\hat{y}_t$ является следующим предсказанным словом на основе оценки контекста докумена h_{t-1} и последнего наблюдаемого вектора слова x_t.

В качестве функции потерь используется функция ошибки перекрёстной энтропии (cross entropy error)

$$J^t(\theta) = -\sum_{j=1}^{|V|} y_{t,j} \times \log(\hat{y}_{t,j}), \text{ на временном шаге t} \tag{3.39}$$

$$J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j} \times \log(\hat{y}_{t,j}), \text{ над корпусом текстов размера T} \tag{3.40}$$

Объем памяти, необходимый для запуска слоя RNN, пропорционален количеству слов в корпусе текстов. То есть предложение, состоящее из k слов, будет храниться в памяти как k векторов. Размер матрицы весов W не масштабируется в соответствии с размером корпуса текстов. Для рекуррентной сети, состоящей из 1000 рекуррентных слоёв, размер матрицы всегда будет 1000 × 1000, в независимости от размера корпуса текстов.

Рекурсивная и рекуррентная нейронные сети

Рекуррентные нейронные сети повторяются (recurring) с течением времени. Пусть необходимо предсказать следующий символ последовательности x=['h','e','l','l']. Эта последовательность даётся на вход единственному нейрону, который имеет связь только с самим собой.

На первом этапе подаётся на вход символ 'h', на следующем этапе 'e' и так далее как показано на рис. 3.11.

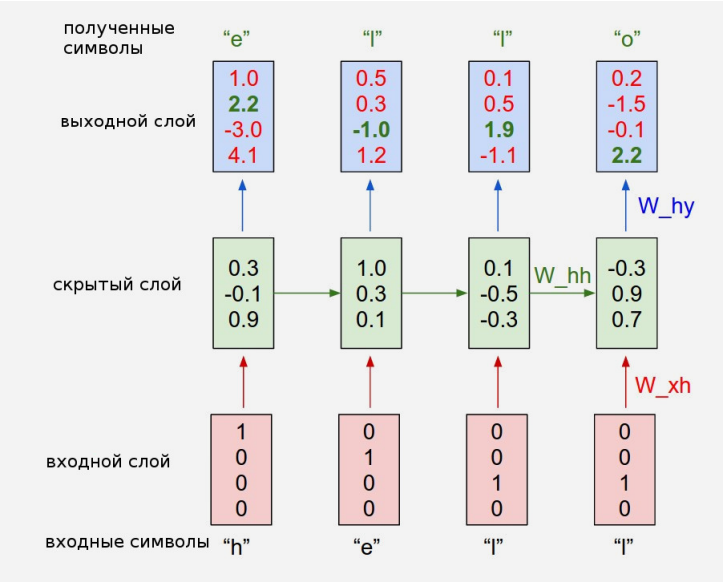

Рисунок 3.11 — Схема нейрона, принимающего на вход последовательность “hell” и “угадывающего” следующую букву слова — “o”

Важно различать понятия рекуррентной и рекурсивной нейронной сети. Рекурсивная нейронная сеть – это обобщение рекуррентной. В рекуррентной сети веса общие (и размерность остаётся одинаковой) по всей длине последовательности. В рекурсивной сети веса также общие в каждом узле. Это значит, то все W_xh веса будут одинаковыми (общими) и таким же будет вес W_hh из-за того, что всё происходит в единственном нейроне, развёртывающимся во времени (см. рис. 3.12).

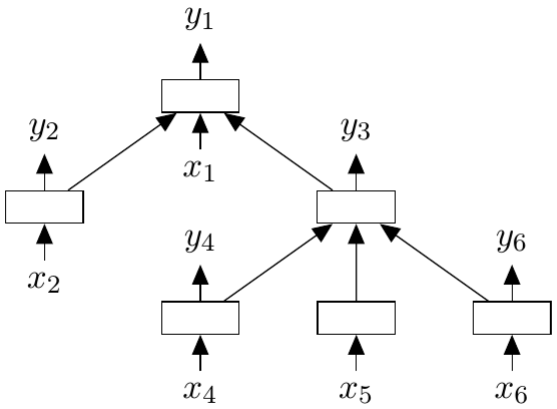

Рисунок 3.12 — Схема рекурсивной нейронной сети

Если сети используются для генерации новых символов, то подойдут рекурсивные сети. Однако для генерации дерева разбора лучше использо-

вать рекуррентные сети.

Проблема исчезающего градиента на примере языковой модели

Пусть имеется языковая модель, с помощью которой необходимо предсказать следующее слово, используя предыдущие. Когда разрыв между важной информацией и отрывком, где она необходима, невелик, RNN можно обучить использовать информацию, полученную ранее ("Облака в **небе**").

Но если необходим контекст, как в примере "Я вырос во Франции. Я свободно говорю **по-французски**", то разрыв между существенной информацией и местом вставки становится шире.

К сожалению, по мере роста разрыва, RNN невозможно обучить связывать информацию. Одной из проблем RNN является то, что её ранние модели очень сложно обучать из-за неустойчивого градиентного спуска. Обучение в предыдущих слоях происходит очень медленно, так как градиент становится всё меньше и меньше при обратном распространении. То есть если сеть работает довольно долго, то градиент может стать крайне неустойчивым.

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W} \tag{3.41}$$

$$\frac{\partial E_t}{\partial W} = \sum_{k=1} t \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W} \tag{3.42}$$

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W^T \times \mathrm{diag}[f'(h_{j-1})] \tag{3.43}$$

$$\frac{\partial E}{\partial W} = \sum_{i=1}^{T} \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} (\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}) \frac{\partial h_k}{\partial W} \tag{3.44}$$

Итоговое выражение

$$||\frac{\partial h_t}{\partial h_k}|| = ||\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}|| \leq (\beta_W \beta_h)^{t-k} \tag{3.45}$$

Произведение $(\beta_W \beta_h)^{t-k}$ становится очень малой или слишком боль-

шой величиной, если $\beta_W \beta_h$ гораздо меньше или значительно больше единицы, а значение $(t - k)$ — значительно велико.

Как только значение градиента становится чрезвычайно большим, оно вызывает переполнение (NaN), которое легко обнаруживается во время обучения: это вызвано проблемой градиентного взрыва (gradient explosion problem). Однако, когда значение градиента становится равным нулю, оно может оставаться необнаруженным, однако качество обучения модели для отдаленных слов в корпусе резко уменьшается. Это вызвано проблемой исчезающего градиента (vanishing gradient problem).

В теории RNN может обрабатывать подобные долговременные зависимости, однако на практике внимательный подбор параметров для решения игрушечных проблем не срабатывает. Эта проблема разрешается добавлением модулей долгой краткосрочной памяти (long short-term memory) в рекуррентную сеть.

Долгая краткосрочная память

LSTM-модуль – рекуррентный модуль сети, предназначенный для запоминания значений как на короткие, так и на длинные промежутки времени. Поскольку модуль не использует функцию активации внутри своих компонентов, то хранимое значение не размывается во времени, а градиент не исчезает при использовании алгоритма обратного распространения ошибки во время обучения сети.

LSTM-модули были разработаны для того, чтобы иметь более устойчивую память, тем самым упрощая для RNN фиксировать долгосрочные зависимости. Все RNN имеют форму цепи повторяющихся блоков нейронной сети. В стандартных рекуррентных нейронных сетях этот повторяющийся блок будет иметь очень простую структуру, например единственный tanh слой.

LSTM тоже имеют подобную цепочечную структуру, однако повторяющийся блок имеет 4 слоя сети, влияющих друг на друга (см. рис. 3.13).

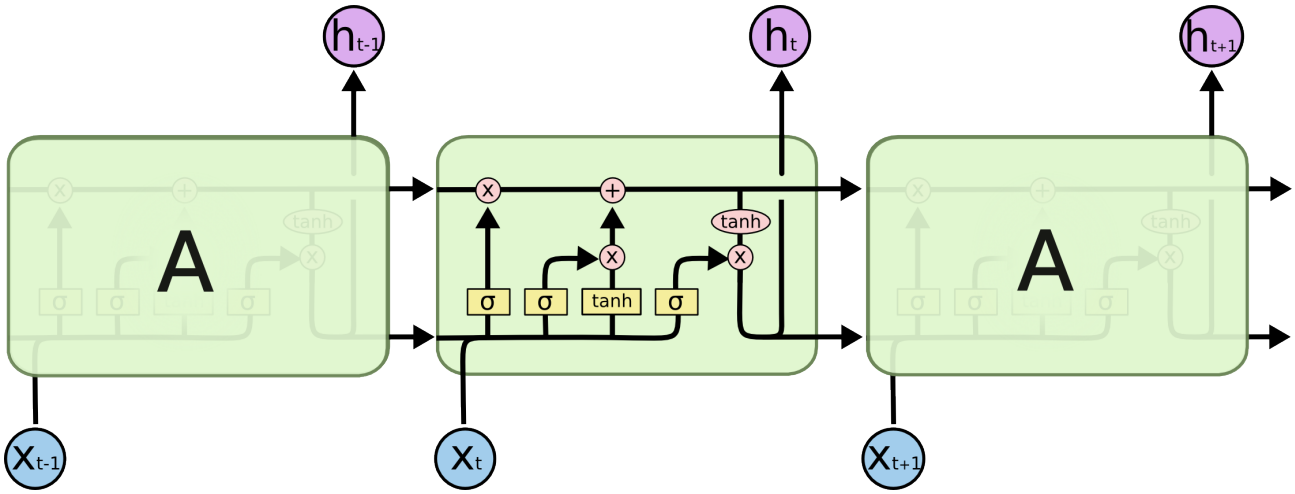

Рисунок 3.13 — Повторяющийся блок в lstm

У LSTM есть способность удалять или добавлять информацию в состояние ячейки, что регулируется структурами под названием “вентили” (gates).

Архитектура LSTM-блока

- Генерация новой памяти: используется входное слово x_t и прошлое скрытое состояние h_{t-1} для образования новой памяти $\widetilde{c}_t$, которая включает в себя аспекты нового слова x_t;
- Входной вентиль: проверяет стоит ли сохранить слово прежде, чем сгенерировать новую память и использует для этого слово, поданное на вход, и предыдущее скрытое состояние. Производит показатель этой информации i_t;
- Вентиль забывания: оценивает, полезен ли предыдущий блок памяти для вычисления текущего блока памяти, и производит показатель f_t;
- Окончательный блок памяти: сначала берётся показатель f_t и в соответствии с ним “забывается” часть предыдущего блока памяти c_{t-1}. Затем та же операция применяется к показателю i_t и блоку новой памяти $\widetilde{c}_t$. Сумма результатов представляет собой окончательный блок памяти c_t;
- Выходной вентиль: назначение выходного вентиля состоит в разделении окончательного блока памяти и скрытого состояния. Окончательный блок памяти c_tсодержит большое количество информации,

которую в скрытое состояние сохранять необязательно. Скрытые состояния используются в каждом вентиле LSTM, поэтому выходной вентиль оценивает, какие части памяти c_t должны присутствовать в скрытом состоянии h_t.

$$i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1}), \text{ входной вентиль} \tag{3.46}$$
$$f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1}), \text{ вентиль забывания} \tag{3.47}$$
$$o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1}), \text{ выходной вентиль} \tag{3.48}$$
$$\widetilde{c}_t = tanh(W^{(c)}x_t + U^{(c)}h_{t-1}), \text{ новый блок памяти} \tag{3.49}$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \widetilde{c}_t, \text{ окончательный блок памяти} \tag{3.50}$$
$$h_t = o_t \circ tanh(c_t) \tag{3.51}$$

Различные варианты LSTM

LSTM с "глазками": вентили "подглядывают" в состояние ячейки

$$f_t = \sigma(W^{(f)}x_t + U^{(f)}c_{t-1} + b_f) \tag{3.52}$$
$$i_t = \sigma(W^{(i)}x_t + U^{(i)}c_{t-1} + b_i) \tag{3.53}$$
$$o_t = \sigma(W^{(o)}x_t + U^{(o)}c_{t-1} + b_o) \tag{3.54}$$
$$c_t = f_t \circ c_{t-1} + i_t \circ tanh(W_c x_t + b_c) \tag{3.55}$$
$$h_t = o_t \circ c_t \tag{3.56}$$

Вентильный рекуррентный узел (GRU): объединяет входной вентиль и вентиль забывания в вентиль обновления (update gate), а также соединяет состояние ячейки и скрытое состояние (см. рис. 3.14):

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}), \text{ вентиль обновления} \tag{3.57}$$
$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}), \text{ вентиль сброса} \tag{3.58}$$
$$\widetilde{h}_t = tanh(r_t \circ Uh_{t-1} + Wx_t), \text{ новая память} \tag{3.59}$$
$$h_t = (1 - z_t) \circ \widetilde{h}_t + z_t \circ h_{t-1}, \text{ скрытое состояние} \tag{3.60}$$

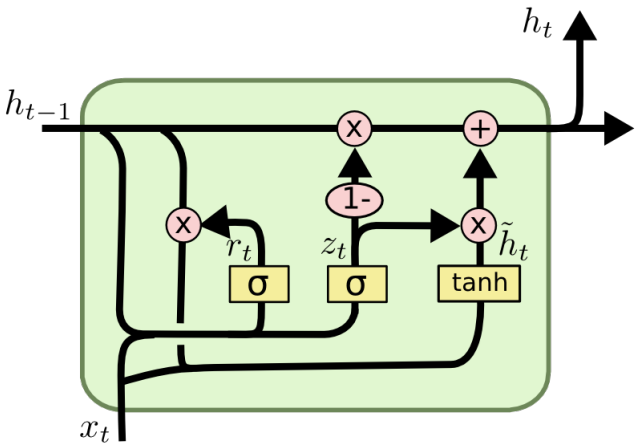

Рисунок 3.14 — Вентильный рекуррентный узел

Обработка естественного языка

Обработка естественного языка направлена на преобразование человеческого языка в формальное представление для облегчения обработки информации компьютером. Текущие приложения включают в себя извлечение информации, машинный перевод, обобщение, поиск и человеко-компьютерное взаимодействие. Они варьируются от синтактических задач, например частичная речевая маркировка, разборов текста(parsing), до смысловых, например смысловое значение слова, тональность текста, разрешение анафоры (зависимость интерпретации некоторого выражения от другого выражения, обычно ранее встречавшегося в тексте).

Простая рекуррентная сеть (simple recurrent network) Джеффа Элмана, описанная в 1990 году, была отправной точкой для решения задач, зависимых от последовательностей (например машинный перевод и модели языка). Однако основную сложность при использовании таких сетей вызывал исчезающий градиент (или gradient exploding). Использование LSTM-блоков решает эту проблему. [11]

Свёрточные нейронные сети — другая широко используемая архитектура для анализа тональности текста, особенно неструктурированного.

Рекурсивные нейронные сети показали заметно высокую производительность благодаря древовидной структуре. Например, для мультиклассовой маркировки, точность, измеренная для каждого уровня дерева, достигает 81%. Для каждого же предложения уровень точности соответствует 46% [12].

Для использования глубоких нейронных сетей в качестве бинарного классификатора для анализа тональности текста, необходимо представить слова этого текста в виде векторов. Векторное представление слов — параметризованная функция, сопоставляющая слова на некотором языке с высокоразмерными векторами : $W : words- > R^n$. Как правило, эта функция представляет собой таблицу поиска, параметризованную матрицей θ, со строкой для каждого слова: $W_\theta(w_n) = \theta_n$. Векторные представления слов и фраз способны значительно улучшить качество работы некоторых методов автоматической обработки естественного языка. В работе рассмотрены две модели векторного представления слов: мешок слов и Word2Vec.

Мешок слов

В модели мешка слов текст произвольной длины представляется в виде вектора. Для этого составляется словарь всех рецензий. Таким образом каждый элемент вектора, представляющего текст, это количество появлений слова в рецензии. Например, если словарь { the, cat, sat, on, hat, dog, ate, and }, то предложение "The cat sat on the hat"будет представлено в виде 2, 1, 1, 1, 1, 0, 0, 0 . Поскольку лексика рецензий достаточно велика, нужно ограничить её размер до, например, 5000 слов. То есть каждое признаковое описание (feature vector) будет иметь размерность (5000,1). Для получения признаковых описаний использована функция CountVectorizer из библиотеки scikit-learn (модуль sklearn.feature_extraction.text).

Word2Vec

Word2Vec — модель, разработанная Google, которая основана на дистрибутивной семантике и векторном представлении слов. Принцип работы Word2Vec — установить значения и семантические отношения между словами. Принцип работы похож на глубинные методы, такие как рекурентные нейронные сети, но Word2Vec более эффективен с точки зрения вычислений.

На вход принимается коллекция текстов, на выходе — векторны представления слов из словаря коллекции. Преимуществами word2vec являются:

- Векторные представления строятся в пространствах размерности по-

рядка десятков и сотен;

- Векторные представления для семантически близких слов находятся на небольшом расстоянии друг от друга.

Word2Vec является предсказательной моделью. Такие модели напрямую пытаются предсказать слово от с помощью соседних слов с точки зрения уже выученных векторных представлений слов. Существуют две модели Word2Vec: непрерывный мешок слов (CBOW) и Skip-gram (см. рис. 3.15). CBOW — модель мешка слов, учитывающая четырёх ближайших соседей (два предыдущих и два последующих слова); порядок следования слов не учитывается. Принципом работы CBOW является предсказание слова при данном контексте.

В k-skip-n-gram предсказывается контекст при данном слове: анализируется последовательность длиной n, где слова находятся на расстоянии не более, чем k друг от друга. Идея Skip-Gram: максимизировать классификацию слова, основываясь на другом слове в этом же предложении. В работе [13] модель Skip-gram формализована следующим образом: необходимо среднюю логарифмическую вероятность

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} log\ p(w_{t+j}|w_t), \tag{3.61}$$

где

$$p(w_O|w_I) = \frac{exp(v_{w_O}^{\mathsf{T}} v_{w_I})}{\sum\limits_{w=1}^{W} exp(v_w^{\mathsf{T}} v_{w_I})} \tag{3.62}$$

Согласно исследованиям [14], CBOW может быть полезна для небольших наборов данных (короткие предложения, но большое количество примеров). Skip-gram же рассматривает каждую пару контекст-цель как новое наблюдение, поэтому на больших наборах данных его производительность будет выше (длинные предложения, но примеров гораздо меньше).

Построение векторнх моделей слов с помощью Gensim

Gensim — набор инструментов с открытым исходным кодом для построения векторных моделей и тематического моделирования. В данной

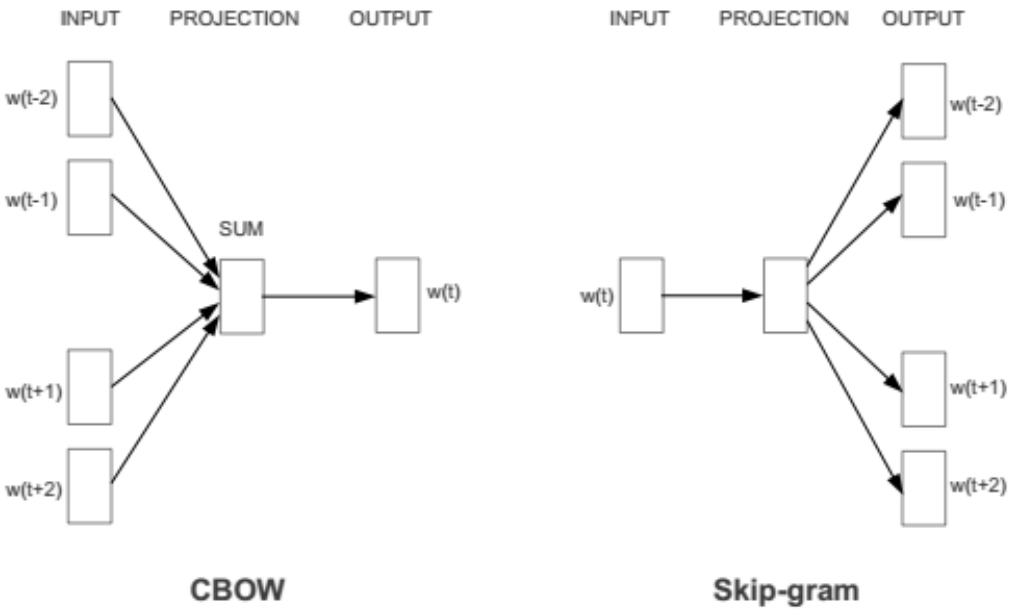

Рисунок 3.15 — Схематичное изображение CBOW и Skip-Gram

работе модель Word2Vec, использованная в практической части, была получена с помощью Gensim на исходных данных.

Примеры работы с готовой моделью Word2Vec, полученной с помощью Gensim (см. рис. 3.16 и 3.17).

```
>>> model.wv.doesnt_match("breakfast cereal dinner lunch".split()
   )
'cereal'
```

Рисунок 3.16 — Фрагмент кода для нахождения лишнего слова в представленном наборе слов

```
>>> model.wv.most_similar_cosmul(positive=['woman', 'king'],
   negative=['man'])
[('queen', 0.71382287), ...]
```

Рисунок 3.17 — Фрагмент кода для нахождения подходящего по смыслу слова, если задано смысловое соответствие для другой пары слов

Описание разработки

Библиотека TensorFlow

Обзор

TensorFlow - это библиотека программного обеспечения с открытым исходным кодом для задач машинного обучения, разработанная Google. Она позволяет создавать и обучать нейронные сети различной архитектуры для обнаружения и распознавания образцов и поиска взаимосвязей. TensorFlow также включает в себя TensorBoard, который представляет собой средство визуализации в браузере для оценки эффективности обучения и сетевых параметров модели.

TensorFlow достигает своей производительности благодаря распараллеливанию задач между центральным и графическими процессорами. Ядро каждой операции реализовано на C++ с использованием библиотек Eigen и cuDNN для лучшей производительности.

Каждое вычисление в TensorFlow представляется как граф потока данных, он же граф вычислений. Граф вычислений является моделью, описывающей как будут выполняться вычисления. Важно заметить, что составление графа вычислений и выполнение операций в заданной структуре — два разных процесса. Граф состоит из плэйсхолдеров(tf.Placeholder), переменных(tf.Variable) и операций. В нём производится вычисление тензоров — многомерных массивов, которые впрочем могут быть числом или вектором.

Графы выполняются в сессиях (tf.Session). Существуют два типа сессий — обычные и интерактивные (tf.InteractiveSession); интерактивная сессия подходит для выполнения в консоли. Сессия хранит состояние переменных (Variables) и очередей (queues). Явное создание сессий и графов гарантирует надлежащее освобождение ресурсов памяти [15]
В графе каждая вершина имеет 0 или больше входов и 0 или больше выходов, и представляет собой реализацию операции. Тензоры представляют собой рёбра графа, а именно массивы произвольного размера (тип массива указывается во время построения графа). Особые вершины, управляющие

зависимости (control dependencies), также могут быть в графе: они указывают, что исходный узел для контрольной зависимости должен закончить выполнение до того, как узел получателя контрольной зависимости начнет выполняться.

Каждая операция имеет название и представляет собой абстрактное вычисление (например, суммирование). У операции могут быть аттрибуты: например, возможность сделать операцию полиморфной для разных типов тензоров. Ядро — специфическая реализация операции, которая может выполнена на определенном типе устройства (центральный или графический процессор).

Переменная — особый вид операции, возвращающий указатель на постоянно меняющийся тензор: такая переменная не исчезает после единичного использования графа. Указатели на подобные тензоры передаются многочисленным операциям, которые затем изменяют указанный тензор. В задачах машинного обучения, параметры модели обычно хранят тензоры в переменных, которые обновляются на каждом шаге обучения.

Данная работа выполнена на единственном устройстве с использованием CPU.

Примеры

Пусть стоит задача классифицировать цветы 3-х видов, используя ширину и длину их чашелистника. Для решения задачи будет использована однослойная нейронная сеть со смещением b и двухэлементным вектором весов W для получения двухэлементного входного вектора X. В качестве активационной функции использована сигмоидальная функция. В обучении использован градиентный спуск, а в качестве функции ошибок — кросс-энтропия (cross entropy loss) $E(x) = -(z \log(y(x)) + (1 - z) * \log(1 - y(x)))$, где z — фактический класс.

Сначала необходимо явно определить переменные, которые будут использованы в графе вычислений [16] (см. рис. 4.1):

```
import tensorflow as tf
...
W = tf.Variable(tf.random_normal([2, 1], stddev=.01), name="W")
b = tf.Variable(tf.random_normal([1], stddev=.01), name="b")
X = tf.placeholder(tf.float32, [None, 2], name="X")
Z = tf.placeholder(tf.float32, name="Z")
```

Рисунок 4.1 — Фрагмент кода для определения переменных в графе

Затем определить активационную функцию (см. рис. 4.2):

```
Y = tf.nn.sigmoid(tf.matmul(X, W) + b)
```

Рисунок 4.2 — Фрагмент кода для определения активационной функции

Скалярное произведение и функция потерь (см. рис. 4.3):

```
tf.scalar_product = lambda a, b: tf.matrix_determinant(\
        tf.matmul(tf.expand_dims(a, 1), b, transpose_a=True))

E = — (tf.scalar_product(Z, tf.log(Y)) + \
        tf.scalar_product(1 — Z, tf.log(1 — Y)))
```

Рисунок 4.3 — Фрагмент кода для определения скалярного произведения и функции потерь

Функция оптимизации (градиентный спуск) (см. рис. 4.4):

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate).
   minimize(E)
```

Рисунок 4.4 — Фрагмент кода для определения функции оптимизации

Обучение сети (см. рис. 4.5):

```
sess = tf.Session()
sess.run(tf.initialize_all_variables())
for _ in range(epochs):
        sess.run(optimizer, feed_dict={X: data.X, Z: data.Z})
sess.close()
```

Рисунок 4.5 — Фрагмент кода для обучения сети

Особенности

В Tensorflow существуют несколько форм параллелизма:

- Параллелизм в отдельных операциях (например, tf.nn.conv2d () и tf.matmul ()). Эти операции имеют эффективные параллельные реализации для многоядерных процессоров и графических процессоров, и TensorFlow использует эти реализации во всех возможных случаях;
- Параллелизм между операциями. TensorFlow использует представление графа вычислений и там, где есть два узла, которые не связаны прямым путем, они могут выполняться параллельно;
- Параллелизм между копиями моделей. Стандартная схема для параллельного обучения — разделить данные между workers, провести одинаковые вычисления для разных данных и обменяться параметрами модели между workers.

Уникальность Tensorflow заключается в возможности проводить частичные подграфовые вычисления. Эта особенность позволяет сделать разбиение нейронной сети, а значит можно использовать распределенное обучение. Также

- Tensorboard: визуализация модели и возможность исследовать порядок вычислений в графе;
- Tensorflow может использована как на мобильных, так и на более мощных устройствах.

В Tensorflow производные задаются автоматически: этот процесс называется автоматическим дифференцированием.

Традиционные методы оценки производной сложно реализовать на практике, так как они имеют ряд недостатков. Например, использование

метода конечных разностей требует обоснованного выбора значения приращения аргумента. Однако существует способ автоматического вычисления вместе с функцией $f(x_0)$ её производной $f'(x_0)$ при некотором значении аргумента $x = x_0$. Данный метод называется автоматическим дифференцированием, так как вычисления значение $f(x_0)$ и $f'(x_0)$ осуществляется одновременно на основе исходного кода только функции $f(x)$. Он позволяет получить точное (до ошибок округления) значения производной, а программу вычислений достаточно выполнить только один раз [17]. Tensorflow использует обратный режим автоматического дифференцирования для операций градиентов и метода конечных разностей для тестов, которые проверяют правильность работы градиента.

Обычно в системах автоматического дифференцифирования оператор (сумма, разность) определён вместе с его производными. То есть после написания функции, в которой определено несколько операторов, программа может сама выяснить, как вычислить соответствующие производные (используя граф вычислений и цепное правило). Выгода очевидна, так как не нужно самостоятельно разрабатывать математические операции и численно проверять каждую производную (см. рис. 4.6).

```
tf.reset_default_graph()
x = tf.Variable(0.)
y = tf.square(x)
z = tf.gradients([y], [x])
```

Рисунок 4.6 — Фрагмент кода для демонстрации автоматического дифференцирования

В графе (см. рис. 4.7) находятся как 2x, так и x^2.

Классические методы классификации

Процесс предобработки данных (каждой рецензии) состоит из следующих шагов:

- Удалить разметку HTML;

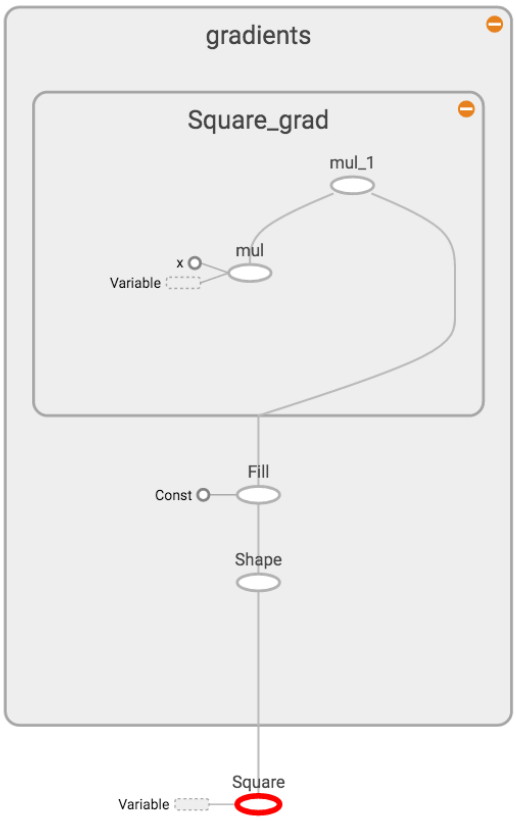

Рисунок 4.7 — Граф Tensorflow для простой функции вычисления градиента

- Удалить все символы, кроме букв и пробелов;
- Из полученного набора слов удалить стоп-слова.

Следующей задачей является преобразование каждой рецензии в векторное представление. Для оценки эффективности каждого из методов будут использованы две модели векторного представления слов: мешок слов и Word2Vec.

Логистическая регрессия

Статистическая модель, используемая для предсказания вероятности возникновения некоторого события.

Scikit-learn (sklearn.linear_model.LogisticRegression). В работе добавлен единственный параметр: random_state = 1.

Наивный байесовский классификатор

Простой вероятностный классификатор, основанный на применении Теоремы Байеса со строгими предположениями о независимости элементов вектора признаков. Достоинством наивного байесовского классификатора является малое количество данных для обучения, необходимых для оценки параметров, требуемых для классификации.

Scikit-learn (sklearn.naive_bayes.GaussianNB). В работе используем наивный байесовский классификатор Гаусса — распределение вероятностей признаков совпадает с функцией Гаусса (то есть нормальное распределение). σ_y и μ_y расчитаны по методу максимального правдоподобия.

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}) \tag{4.1}$$

Случайный лес (random forest)

Алгоритм заключается в использовании комитета решающих деревьев. Классификация объектов проводится путём голосования: каждое дерево комитета относит классифицируемый объект к одному из классов, и побеждает класс, за который проголосовало наибольшее число деревьев. Оптимальное число деревьев подбирается таким образом, чтобы минимизировать ошибку классификатора на тестовой выборке. Использует усреднение для повышения точности прогнозирования и контроля избыточной подгонки. Размер подвыборки всегда совпадает с размером оригинальной выборки.

Scikit-learn (sklearn.ensemble.RandomForestClassifier). Количество деревьев (n_estimators) в работе равно 100. Чем их больше, тем лучше, но может возникнуть как проблема переобучения, так и проблемы с памятью устройства.

Метод опорных векторов (SVM)

Основная идея метода — перевод исходных векторов в пространство более высокой размерности и поиск разделяющей гиперплоскости с максимальным зазором в этом пространстве. Стандартная функция из scikit-learn (sklearn.svm.SVC) не подходит, поскольку временная сложность дан-

ного алгоритма квадратична. Эффективность метода опорных векторов значительно снижается, если количество признаковых описаний очень велико. Он имеет большую гибкость в выборе штрафов и функций потерь и должен лучше масштабироваться для большого количества образцов. Оптимизация функции потерь SVM с помощью градиентного спуска:

$$L(w, D) = \frac{1}{2}||w||_2^2 + C\sum_{i=1}^{N} max(0, 1 - y_i(w^{\mathsf{T}}x_i + b)), \tag{4.2}$$

где

$$D = \{(x_i, y_i)\}_{i=1}^{N}, x_i \in Rd \text{ and } y_i \in \{-1, +1\}. \tag{4.3}$$

Оптимизация функции потерь — (regularization_loss + hinge_loss).

Для случая word2vec будет использован SGDClassifier из sklearn.linear_model с ошибкой l2.

Свёрточная нейронная сеть

Векторные представления данных осуществлены с помощью модели Word2Vec (Skip-gram Model). Архитектура свёрточной сети представлена на рис. 4.8.

Параметры сети

Параметрами свёрточной нейронной сети являются:

- Размерность векторного представления слова = 150;
- Размер батча = 50;
- Темп обучения = 0.001;
- Количество шагов обучения = 2950;
- Dropout = 0.8;
- Размерность фильтров = (2, 3, 4);
- L2 regularization = 4.

В качестве функции оптимизации используется Adam (Adaptive Moment Estimation). Его отличительными особенностями являются:

- оценка первого момента вычисляется как скользящее среднее;
- так как оценки первого и второго моментов инициализируются нулями, используется небольшая коррекция, чтобы результирующие оценки не были смещены к нулю.

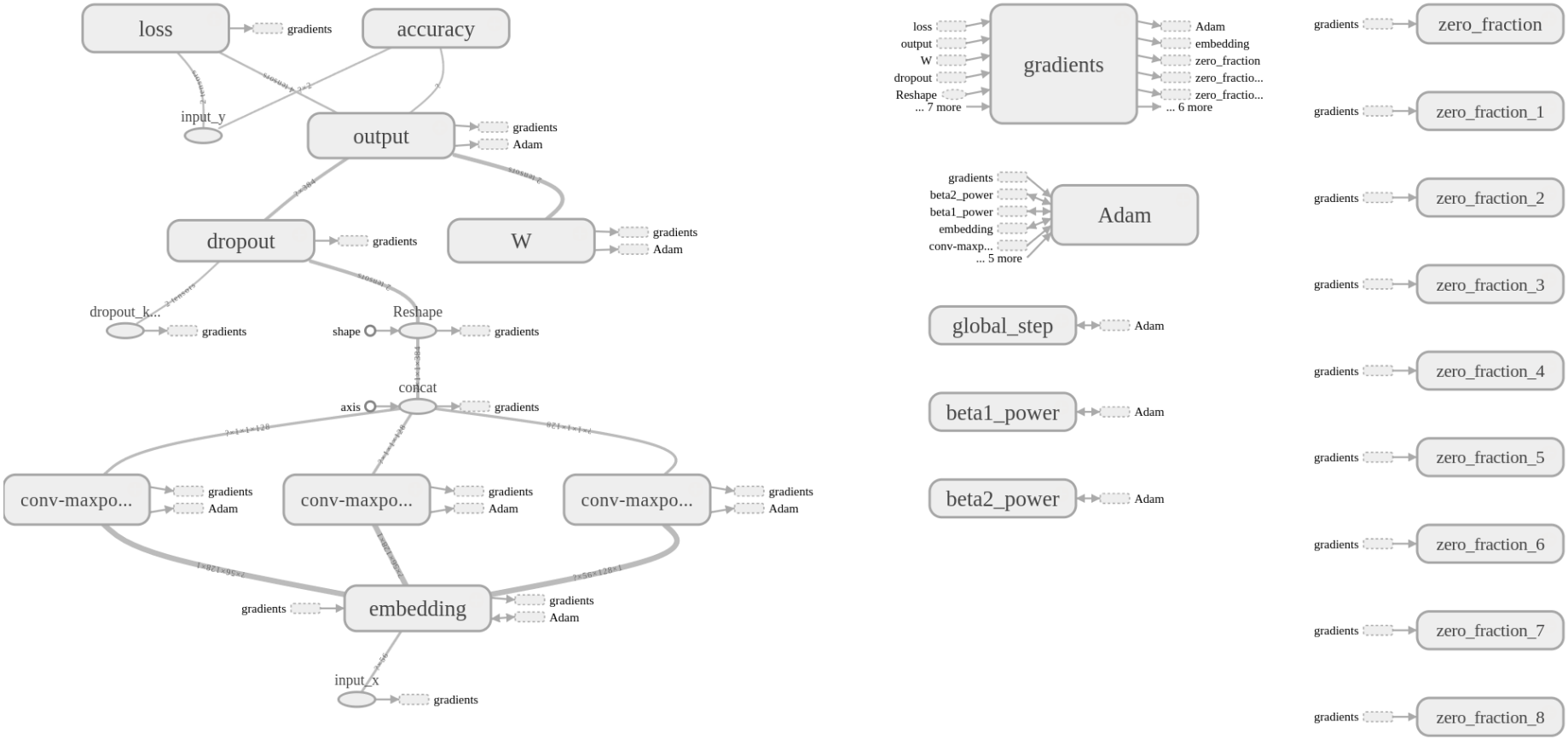

Рисунок 4.8 — Архитектура свёрточной сети, представленная в tensorboard

Метод также инвариантен к масштабированию градиентов [18].

Рекуррентная нейронная сеть с LSTM-блоками

Векторные представления данных осуществлены с помощью модели Word2Vec (Skip-gram Model). LSTM считаются [4] наилучшей архитектурой для анализа тональности текста. Нейронные сети, составленные из LSTM-модулей, особенно хорошо обрабатывают отрицание (negation), если в ячейке есть projection unit (то есть больше памяти для сети). Архитектура реализованной сети (см. рис. 4.9):

- Слой векторного представления слов: преобразует каждый вход (тензор из k слов) в тензор k N-мерных векторных представлений слов (N — размер представления). Каждому слову соответствует вектор весов, который необходимо изучить во время процесса обучения сети:
  - Из каждой рецензии удаляются все символы, кроме пробелов и букв; текст представлен только строчными буквами;
  - Создаётся словарь, из каждой рецензии удаляются самые редко встречающиеся слова (которые скорее всего появились в результате грамматической ошибки);
  - Создаётся тензор, представляющий каждую рецензию;
  - Каждый тензор дополняется нулями соответственно рецензии, имеющей максимальную длину.
- RNN слой: создан из LSTM-блоков с обёрткой dropout'а. Веса LSTM изучаются во время обучения. RNN слой развёртывается динамично, беря на вход k векторных представлений и выдаёт M-мерные вектора, где M — количество LSTM-модулей в блоке;
- Слой softmax: выход RNN слоя усредняется через k временных шагов. Выдаёт тензор размерности M, который используется для вычисления вероятностей для задачи классификации.

Параметры сети

Параметрами рекуррентной нейронной сети являются:

- Размер слоя векторного представления = 50;
- Размер батча = 100;

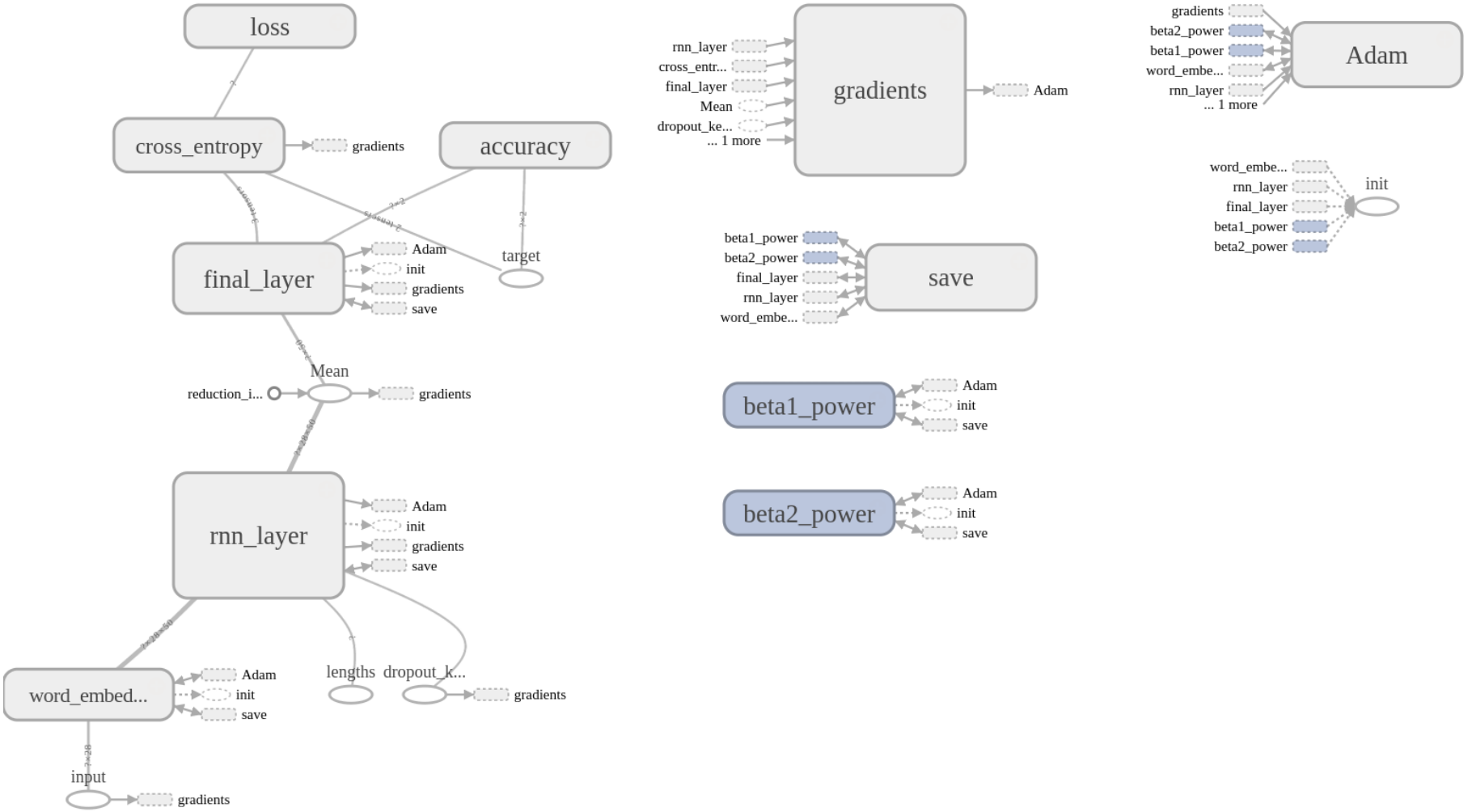

Рисунок 4.9 — Архитектура рекуррентной сети с LSTM блоками, представленная в tensorboard

- Темп обучения = 0.1;
- Количество шагов обучения = 1000;
- Dropout = 0.5;
- Скрытый размер LSTM слоя (количество LSTM-модулей в блоке) = 50.

Показатели качества

Доля корректных прогнозов (accuracy) — процент ошибок, допускаемых классификатором.

Следующие показатели будут использованы только для классических методов классификации:

- Мера точности (precision) — отношение t_p к (t_p+f_p), где t_p — количество истинных положительных величин, а f_p — количество ложных положительных величин. То есть мера точности характеризует сколько полученных от классификатора положительных ответов являются правильными;
- Мера полноты (recall) — отношение t_p к $(t_p + f_n)$, где f_n — количество ложных отрицательных величин. Мера полноты определяет способность классификатора угадывать как можно большее число положительных ответов из ожидаемых;
- Мера F_1 — среднее гармоническое меры точности и меры полноты. Характеризует пороговое качество классификатора;
- Носитель меры (support) — количество данных каждого из классов. Более строгое определение: наименьшее замкнутое множество, на котором сосредоточена мера.

Результаты разработки

Результаты эксперимента

Линейные и нелинейные методы классификации

С использованием модели **мешка слов** (см. табл. 5.1, 5.2, 5.3 и 5.4).

Таблица 5.1 — Логистическая регрессия

Класс	Мера точности	Мера полноты	Мера F_1	Носитель меры
0	0.76	0.76	0.76	1092
1	0.75	0.75	0.75	1039
total	0.75	0.75	0.75	2131
Точность 0.754				

Таблица 5.2 — Наивный байесовский классификатор

Класс	Мера точности	Мера полноты	Мера F_1	Носитель меры
0	0.74	0.72	0.73	1092
1	0.72	0.74	0.73	1039
total	0.73	0.73	0.73	2131
Точность 0.73				

Таблица 5.3 — Случайный лес

Класс	Мера точности	Мера полноты	Мера F_1	Носитель меры
0	0.71	0.74	0.72	1092
1	0.71	0.68	0.69	1039
total	0.71	0.71	0.71	2131
Точность 0.708				

Таблица 5.4 — Линейный метод опорных векторов

Класс	Мера точности	Мера полноты	Мера F_1	Носитель меры
0	0.79	0.56	0.65	1092
1	0.65	0.85	0.73	1039
total	0.72	0.70	0.69	2131
Точность 0.699				

С использованием модели **Word2Vec** (см. табл. 5.5, 5.6, 5.7 и 5.8).

Таблица 5.5 — Логистическая регрессия

Класс	Мера точности	Мера полноты	Мера F_1	Носитель меры
0	0.77	0.77	0.77	1092
1	0.76	0.76	0.76	1039
total	0.86	0.86	0.86	5000
Точность 0.766				

Таблица 5.6 — Наивный байесовский классификатор

Класс	Мера точности	Мера полноты	Мера F_1	Носитель меры
0	0.73	0.72	0.72	1092
1	0.71	0.72	0.71	1039
total	0.72	0.72	0.72	2131
Точность 0.718				

Таблица 5.7 — Случайный лес

Класс	Мера точности	Мера полноты	Мера F_1	Носитель меры
0	0.74	0.75	0.75	1092
1	0.74	0.72	0.73	1039
total	0.74	0.74	0.74	2131
Точность 0.739				

Таблица 5.8 — Линейный метод опорных векторов

Класс	Мера точности	Мера полноты	Мера F_1	Носитель меры
0	0.83	0.63	0.71	1092
1	0.69	0.86	0.77	1039
total	0.76	0.74	0.74	2131
Точность 0.742				

Свёрточная нейронная сеть

Точность 0.799.

Графики точности модели и функции потерь представлены на рис. 5.1 и 5.2 соответственно.

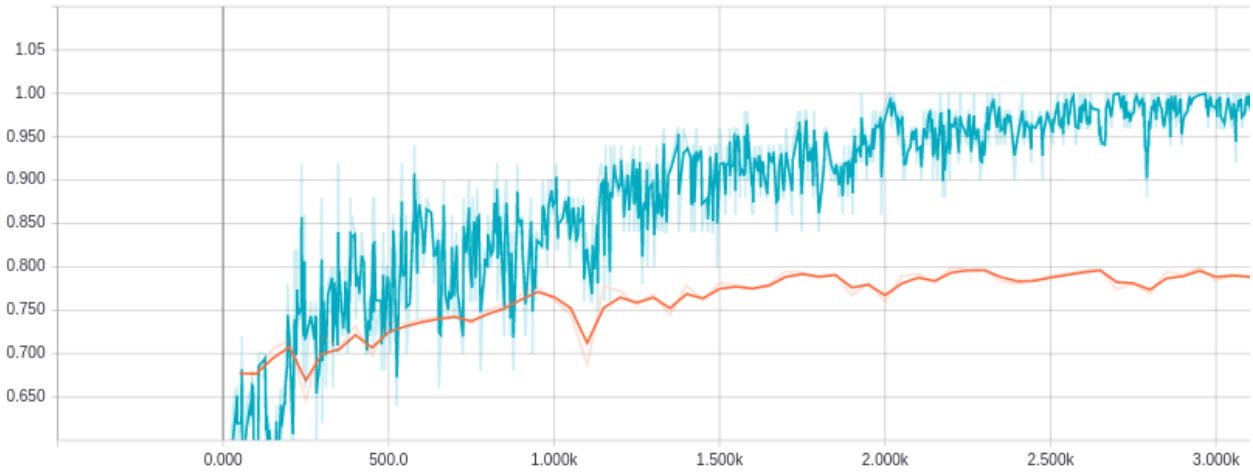

Рисунок 5.1 — Доля корректных прогнозов (accuracy): синий график – обучение, красный — проверка

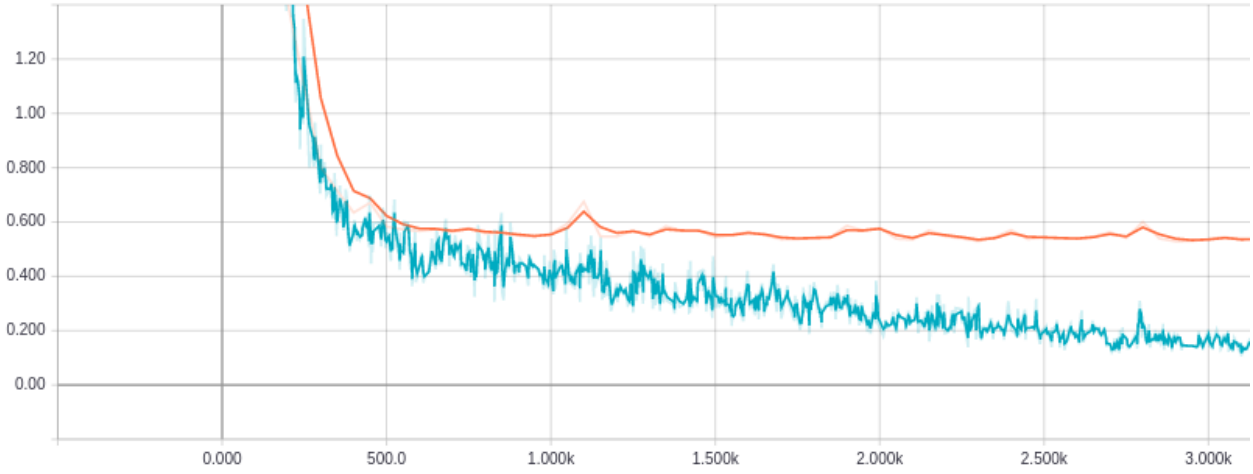

Рисунок 5.2 — Функция потерь: синий график – обучение, красный — проверка

Рекуррентная нейронная сеть с LSTM-блоками

Решающим в обучении модели оказался выбор функции минимизации градиентного спуска. Сначала был использован алгоритм RMSProp (root mean square propogation), идея которого заключается в масштабировании градиента.

Однако при прочих равных условиях (размере скрытого LSTM слоя и темпе обучения) использование алгоритма оптимизации Adam (который помимо идеи масштабирования градиента использует идею инерции) позволило достичь максимальной точности относительно уже реализованных методов в данной работе — 83%.

Графики точности модели и функции потерь представлены на рис. 5.3 и 5.4 соответственно.

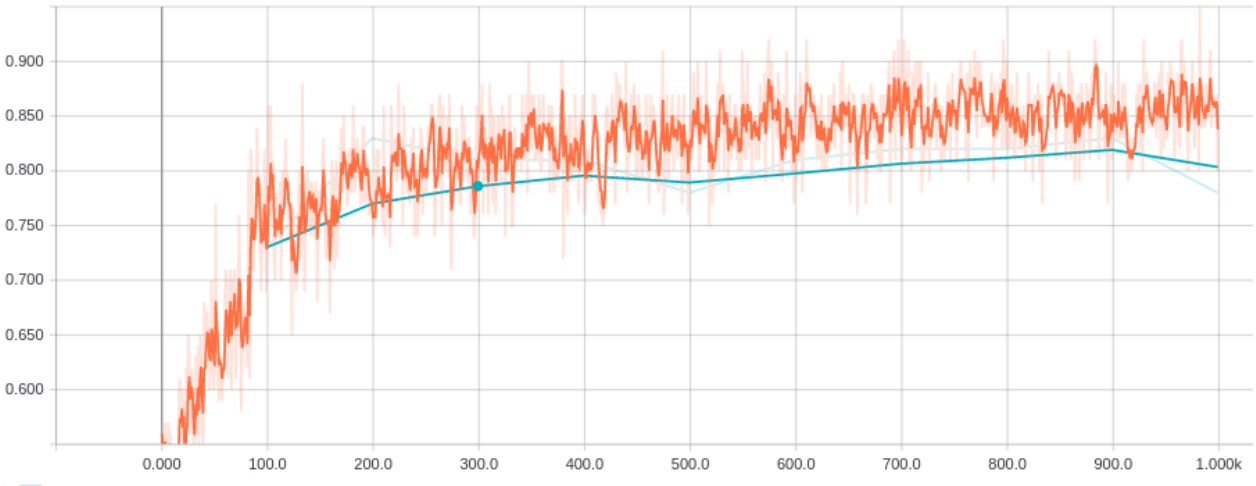

Рисунок 5.3 — Доля корректных прогнозов (accuracy): красный график – обучение, синий — проверка

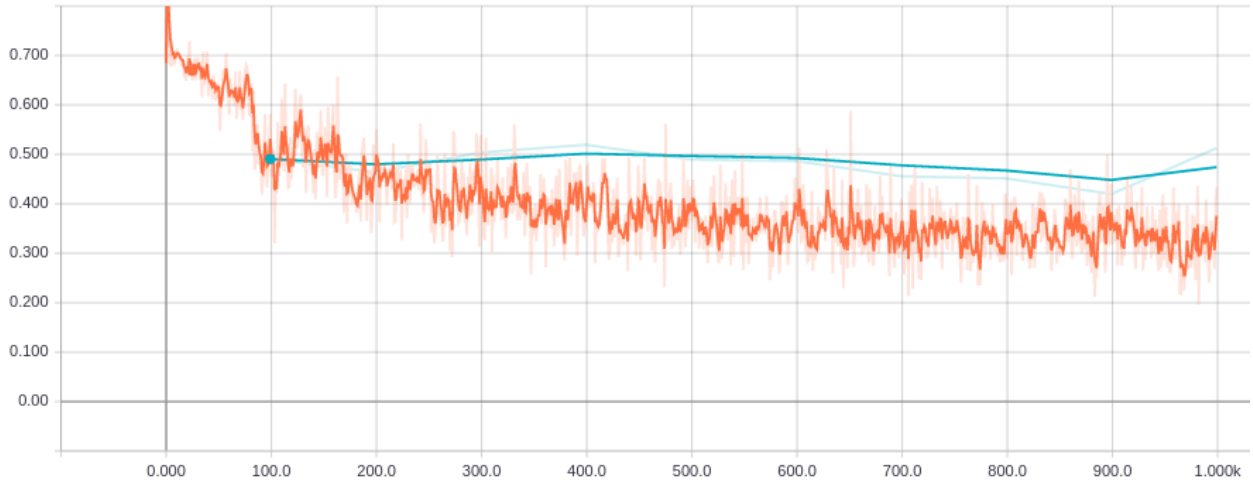

Рисунок 5.4 — Функция потерь: красный график – обучение, синий — проверка

Сравнение и оценка результатов

Для классических алгоритмов классификации использование мешка слов в качестве модели векторного представления слов точность предсказаний не превышала 75.4% (логистическая регрессия), минимальная точность составила 69.9% (линейный метод опорных векторов).

Благодаря использованию модели Word2Vec удалось улучшить точность предсказаний почти для всех методов (кроме наивного байесовского классификатора): например, точность линейного метода опорных векторов стала 74.2%. Тем не менее самым эффективным бинарным классификатором оказалась логистическая регрессия: её точность с использованием модели Word2Vec равна 76.6%. При использовании свёрточных нейронных сетей с моделью Word2Vec удалось получить точность 79.9%.

Самой эффективной архитектурой для анализа тональности текста оказалась рекуррентная нейронная сеть с LSTM-блоками. Её максимальная точность составила 83.0%.

Полученные экспериментальные данные показывают более высокую эффективность работы глубоких нейронных сетей по сравнению с классическими алгоритмами для анализа тональности текста.

Информационный маркетинг

Введение

Маркетинг представляет собой сочетание процессов создания, продвижения и предоставление услуги или продукта покупателям, а также управление взаимоотношениниями с клиентами с максимизацией выгоды для организации.

Информационный маркетинг — это применение классических методов маркетинга для продуктов и услуг информационной индустрии. Реализация маркетинговых исследований осуществляется с использованием информационных технологий, а сам маркетинг выполняется с учётом специфических особенностей продукта и рынка продвижения.

Проведение предпроектных исследований

Маркетологу необходимо постоянно быть в курсе восприятия рынком предлагаемого продукта или услуги. Продуктовые компании получают отзывы на естественном языке о продуктах из опросов, жалоб клиентов, отзывов о магазине и т. д. В данной работе были реализованы различные архитектуры нейронных сетей для того, чтобы маркетолог мог сократить этот огромный объем информации и получить сжатый информативный отчёт. На выходе сетей будут лишь простые числовые оценки.

В век информации получить данные уже обработанных отзывов, рецензий, комментариев очень легко. Однако анализировать такие массивы информации чрезвычайно сложно. В данной работе анализируется тональность текста: какие эмоции испытывал человек, оставивший данный отзыв, то есть стоит задача бинарной классификации — определить, позитивный или негативный отзыв оставил клиент.

Реализованы различные архитектуры нейронных сетей, а также проведено их сравнение (точность) со стандартными методами обучения без учителя.

Благодаря средствам Tensorflow, а именно Tensorboard, процесс обучения легко визуализировать, и соответственно графикам обучения (функции потерь, точности) изменять параметры сети для достижения наилуч-

ших результатов. Поскольку подбор параметров является очень нетривиальной задача, то в данном программном продукте параметры сети легко меняются, а эффективность сети отслеживается.

В программе можно использовать следующие методы для анализа тональности рецензий:

- Набор реализованных алгоритмов классификации из библиотеки scikit-learn);
- Свёрточная нейронная сеть;
- Рекуррентная нейронная сеть с блоками LSTM.

Также приводятся точность каждого из методов и другие показатели качества. Время работы каждого из методов зависит от наличия графического процессора и производительности центрального процессора. В случае методов, использующих нейронные сети, на время выполнения влияют значения параметров сети, заданные пользователем. Программу может использовать любой программист, умеющий читать документацию и устанавливать нужные для работы программные модули. Графический интерфейс отсутствует.

Данный программный продукт целесообразно выполнять двум-трём программистам: больше не нужно, поскольку проект достаточно небольшой, однако просмотр кода и дискуссии довольно важны. Непринципиально, будет ли этой задачей заниматься junior разработчик (junior developer), однако знания основ машинного обучения, особенностей языков и библиотек для них, знание стиля кода (styleguide) для успешной реализации необходимы. Зарплата разработчиков зависит от региона их проживания и, естественно, опыта работы. По данным сайта http://www.payscale.com (статистика американских зарплат) средняя годовая заработная плата инженера по разработке алгоритмов машинного обучения составляет 100,000$.

Поскольку в проекте используется библиотека Tensorflow с открытым исходным кодом, никаких затрат на покупки лицензий нет. Однако возникает вопрос, что целесообразнее использовать для обучения моделей нейронных сетей:

- Создание собственного кластера машин для использования библиотек;

- Использование универсальных PaaS (Amazon);
- Использование специализированных Paas (Google Cloud).

Поскольку данный проект реализован с использованием Tensorflow, а работа над проектом ведётся удалённо, то наиболее оптимальным вариантом является использование Google Cloud.

Определение затрат на выполнение и внедрение проекта

Расчёт затрат производится по следующей формуле:

$$К_{ПР} = К_{ПЕРС} + К_{СВТ} + К_{ИПС} + К_{ПРОЧ},$$

где

- $К_{ПЕРС}$ - затраты на оплату труда персонала и связанные с этим выплаты;
- $К_{ИПС}$ - затраты на инструментальные программные средства;
- $К_{СВТ}$ - затраты на средства вычислительной техники, используемые для проектирования;
- $К_{ПРОЧ}$ - прочие расходы.

Затраты на оплату труда

Расчет затрат на оплату труда, учитывая что разработчики имеют примерно одинаковую квалификацию:

$$К_{ПЕРС} = Z_{ЗП} * (1 + Н + Ф) * d_{загр} * n_{П} * m_{П},$$

где

- $Z_{ЗП}$ - заработная плата одного разработчика за месяц;
- Н - процент накладных расходов, исчисляемых к сумме зарплаты разработчика;
- Ф - процент отчислений в фонды, относимых к единому социальному налогу;
- $d_{загр}$ - доля загрузки разработчика работой по проекту АИС;
- $n_{П}$ - количество месяцев, в течение которых разработчик был занят проектом;

- $m_{\text{П}}$ - число разработчиков, задействованных в проекте.

Пусть заработная плата одного программиста составляет $Z_{\text{ЗП}}$=150 тысяч рублей в месяц.

Всего над проектом работают 3 разработчика.

Суммарная ставка отчислений в социальные фонды - 30%.

Процент накладных расходов устанавливается бухгалтерией и равен примерно 42-43%. Пусть в работе процент накладных расходов составляет 42.5%. Работа является удалённой: так можно минимизировать довольно большие накладные расходы (например, на аренду офисов). Пусть все проектировщики заняты разработкой полный рабочий день: доля загрузки каждого составляет 1. Необходимо использовать набор функций и инструментов Tensorflow и реализовать эффективный (точность значительно превышает стандартные алгоритмы scikit-learn библиотеки и использование модели word2vec на этих алгоритмах) программный продукт для автоматического анализа тональности текста. Объем работ, необходимый для реализации и тестирования, составляет 2 месяца.

Таким образом:

$$\text{К}_{\text{ПЕРС}} = 150000 * (1 + 0.425 + 0.30) * 1 * 2 * 3 = 1552500 \text{ рублей}$$

Затраты на оплату труда и связанные выплаты составили 1 552 500 рублей.

Затраты на инструментальные программные средства

Разработка будет проходить на языке Python. Удобной средой для разработки является PyCharm. Стоимость подписки на два месяца для команды из трёх человек составляет 120$, то есть около 6840 рублей.

Затраты на средства вычислительной техники

При выборе уровня STANDARD_1 и цене 0.54$ за каждый узел машинного обучения (machine learning unit), необходимо использовать 500 узлов при выполнении работы за 1.5 часа:

$$(500 * 0.54) * 90/60 = 405\$ \text{ для обучения модели} \tag{6.1}$$

1000 предсказаний стоят 0.11$. Количество предсказаний равно 20000

(2500 в неделю):

$$\frac{20000}{1000} * 0.11 = 2.2\$ \text{ за } 1000 \text{ предсказаний} \tag{6.2}$$

Каждое предсказание обрабатывается узлом (node) в среднем за 0.72 секунды ($0.44 per node hour):

$$2500 * 8 * 0.72 = 14400 \text{ минут} \tag{6.3}$$

Работа узла за час оценивается в 0.44$:

$$14400 * 0.44/60 = 105.6\$ \text{ для предсказаний} \tag{6.4}$$

Примерные расчёты использования платформы для обучения и предсказания всех моделей составляют около 505$, то есть 29104 рублей:

$$(405 + 105.6) * 57 = 29104 \text{ рублей} \tag{6.5}$$

Результат

Таким образом, полностью затраты на разработку составят 1587625 рублей.

$$\text{К}_{\text{ПР}} = 1552000 + 29104 + 6840 = 1587944 \text{ рублей}$$

Расчет цены услуг

Основные затраты после покупки продукта будут производиться на поддержку его работы. Сначала требуется установка подходящего программного обеспечения для сборки проекта, затем написание документации для файлов (их форматирования), которые будут использовать для анализа в проекте. Затем простой командой в терминале запускается программа обучения на данных, которая выполняет вычисления и выдаёт подробный отчёт об эффективности каждого из методов. Для предсказания на других наборах данных, используя только что обученную модель, необходимо запустить другой скрипт.

Достаточно одного человека, поддерживающего систему, например, одного из разработчиков программы, который досконально знает, как она

функционирует. Так как оплата труда разработчика составляет 150 000 рублей, а для поддержки одного проекта программист может затратить около недели работы, то оплата за это время составит 37 500 рублей. Значит за год для оплаты труда программиста для развёртывания и поддержки системы небходимо затратить 450 000 рублей. Годовой расход на PaaS составляет около 5 000 рублей для одного клиента. Также необходимо заплатить 30% налога от зарплаты специалиста. Таким образом издержки для одного клиента составляют 590 000 рублей.

Используя затратный метод ценообразования, рассчитаем цену услуг:

$$P = C * (1 + N_c), \tag{6.6}$$

где

- P — цена;
- C — издержки оказания услуги;
- N_c - норма рентабельности.

  Пусть $N_c = 0.25$. Тогда цена оказания услуги равна 737 500 рублей.

$$P = 590000 * (1 + 0.25) = 737500 \text{ рублей в год} \tag{6.7}$$

Расчет показателей конкурентоспособности разработанной продукции

Рассчитаем оценку конкурентоспособности разработанного продукта, используя формулу

$$K_{\text{ок}} = \frac{Q}{E}, \tag{6.8}$$

где

- Q — показатель конкурентоспособности по характеристикам качества;
- E — показатель конкурентоспособности по экономическим характеристикам.

Конечный продукт предоставляется бесплатно с открытым исходным кодом, поэтому показатель E примем за 1.

Показатель конкурентоспособности по характеристикам качества Q может определяется по формуле:

$$Q = \sum_{i=1}^{n} w_i \times \frac{q_i}{q_i^0} + \sum_{j=1}^{m} w_j \times \frac{q_j^0}{q_j}, \tag{6.9}$$

где

- i - номер показателя качества товара или услуги, большее абсолютное значение которого соответствует более высокому уровню качества товара или услуги (i = 1,n);
- j - номер показателя качества товара или услуги, большее абсолютное значение которого соответствует более низкому уровню качества товара или услуги (j = 1, m);
- q^i и q^j - абсолютное значение i-го и j-го показателей качества анализируемого товара или услуги;
- q_i^0, q_j^0 - абсолютное значение i-го и j-го показателей товара или услуги, принятых за базу сравнения (эталон);
- w_i, w_j - весовые коэффициенты (коэффициенты весомости) i-го и j-го показателей качества.

Определим показатели качества для разработанной системы (см. табл. 6.1).

Таблица 6.1 — Значения показателей качества

	q^0	q_i	w_i
Простота программирования системы	1	0.7	$\frac{1}{6}$
Производительность системы	1	0.9	$\frac{1}{2}$
Простота добавления новых функций в систему	1	2	$\frac{1}{3}$

Тогда

$$Q = \frac{1}{6} * \frac{0.7}{1} + \frac{1}{2} * \frac{0.9}{1} + \frac{1}{3} * \frac{2}{1} = 1.23 \tag{6.10}$$

$$K_{\text{ок}} = \frac{1.23}{1} = 1.23 \tag{6.11}$$

Отобразим значение показателей конкурентноспособности [19] на графике (см. рис. 6.1):

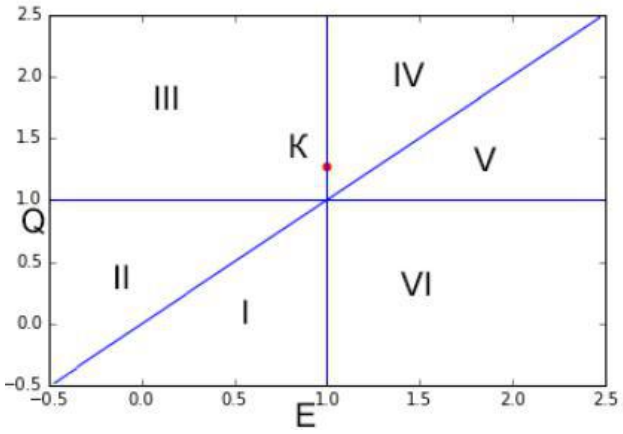

Рисунок 6.1 — Оценка показателей конкурентноспособности

Значение коэффициента конкурентоспособности находится в зоне IV. Значит следует выбрать стратегию развития, ориентированную на сегмент потребителей, для которых приоритет имеет качество продукта, а не его цена.

Предложения по продвижению разработанной продукции

Существуют несколько категорий пользователей, которые влияют на формирование мнения о продукте:

- профессиональные разработчики;
- программисты-любители;
- пользователи-энтузиасты, стремящиеся использовать все компьютерные новинки в интересующих их областях;
- пользователи-непрограммисты;
- остальное экономически активное население, невзаимодействующее с продуктом напрямую, а пользующееся результатами его работы.

Поскольку продукт довольно непрост в эксплуатации, имеет смысл продвигать его с помощью публикаций, в которых описываются особенности реализации и примеры его работы, на различных медиа-платформах, аудиторию которых в большинстве своём составляют программисты-любители и профессионалы. Не лишним будет опубликовать открытый исходный код (boilerplate) проекта.

Для охвата большего количества пользователей необходимо приду-

мать нетривиальные примеры работы данного продукта, не вдаваясь в технические подробности.

Определение кода разрабатываемого программного изделия

Разработанная программа соответствует ОКП 50 2800 7 “Программные средства для систем искусственного интеллекта”.

ЗАКЛЮЧЕНИЕ

В соответствии с поставленной целью — разработка алгоритмов глубокого обучения для анализа тональности текста и сравнение их эффективности с другими классификаторами на основе алгоритмов машинного обучения — были реализованы архитектуры свёрточной нейронной сети, рекуррентной нейронной сети с LSTM-блоками, а также проведено сравнение показателей качества их классификации с другими классификаторами.

При использовании модели мешка слов точность различных методов была значительна выше случайной (около 70%), однако применяя модель Word2Vec, удалось значительно улучшить точность работы алгоритмов (на несколько единиц). Однако нейронные сети показали лучшие результаты. Точность классификатора на основе свёрточной нейронной сети оказалась 79.9%. Самую высокую точность показал классификатор на основе рекуррентной сети с LSTM-блоками — 83.3%.

Результаты исследования показывают, что использование глубоких нейронных сетей значительно улучшает точность анализа тональности текста. Преимущество рекуррентной сети на основе LSTM над свёрточной нейронной сетью в области анализа тональности уже было доказано в различных исследованиях (например [11]), однако важно отметить, что в данной работе были реализованы простейшие архитектуры глубоких нейронных сетей. Улучшение параметров модели, использование более расширенной модели векторного представления слов Word2Vec, применение attention-механизмов позволит значительно увеличить эффективность бинарного классификатора для анализа тональности на основе глубоких нейронных сетей.

Возможным направлением для дальнейшей работы может быть распознавания ключевых слов, вносящих наибольший вклад в положительный или отрицательный отзыв.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. B. Pang and L. Lee. Opinion mining and sentiment analysis. *Foundations and Trends in Information Retrieval archive*, 2008.

2. Данные рецензий, используемых в работе, sentence polarity dataset v1.0. `http://www.cs.cornell.edu/people/pabo/movie-review-data/`. (дата обращения 25.05.2017).

3. Y. Kim. Convolutional neural networks for sentence classification. *arXiv:1408.5882 [cs.CL]*, 2014.

4. K. S. Tai et al. Improved semantic representations from tree-structured long short-term memory network. *arXiv:1503.00075 [cs.CL]*, 2015.

5. Q. Le and T. Mikolov. Distributed representations of sentences and documents. *arXiv:1405.4053 [cs.CL]*, 2014.

6. K. Tran et al. Evaluation of deep learning toolkits. `https://github.com/zer0n/deepframeworks/blob/master/README.md`. (дата обращения 25.05.2017).

7. J. Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation 4(2). c. 234-242*, 1992.

8. S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation 9(8), c. 1735-1780*, 1997.

9. X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Aistats 9, , c. 249-256*, 2010.

10. I. Sutskever et al. On the importance of initialization and momentum in deep learning. *ICML'13 Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, c. 1139-1147*, 2013.

11. A. Karpathy. The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`. (дата обращения 25.05.2017).

12. R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. *ICML '08 Proceedings of the 25th international conference on Machine learning, c. 160-167*, 2008.

13. T. Mikolov et al. Distributed representations of words and phrases and their compositionality. *arXiv:1310.4546 [cs.CL]*, 2013.

14. T. Mikolov et al. Efficient estimation of word representations in vector space. *arXiv:1301.3781 [cs.CL]*, 2013.

15. Google Research Team. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467 [cs.DC]*, 2016.

16. M. Schrimpf. Should i use tensorflow? *arXiv:1611.08903 [cs.LG]*, 2016.

17. К. К. Семёнов. Автоматическое дифференцирование функций, выраженное программным кодом. *Вестник Саратовского государственного технического университета*, 2011.

18. В. Д. Чабаненко. Модификации метода стохастического градиентного спуска для задач машинного обучения с большими объемами данных. Master's thesis, Московский государственный университет имени М.В. Ломоносова, 2016.

19. В.И. Фомин. *Методические указания по выполнению дополнительного раздела "Информационный маркетинг"*. СПбГЭТУ «ЛЭТИ», 2015.

20. J. Turian et al. Word representations: A simple and general method for semi-supervised learning. *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, c. 384-394*, 2010.

21. R. Kadlec et al. Improved deep learning baselines for ubuntu corpus dialogs. *arXiv:1510.03753*, 2015.

22. D. Britz. Understanding convolutional neural networks for nlp. http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/. (дата обращения 27.05.2017).

23. C. Manning and R. Socher. Лекции Стэнфордского университета по курсу "natural language processing with deep learning". http://web.stanford.edu/class/cs224n/. (дата обращения 20.05.2017).

24. Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

25. C. Olah. Neural networks, recurrent neural networks, convolutional neural networks. http://colah.github.io/. (дата обращения 25.05.2017).

26. Stanford University. Unsupervised feature learning and deep learning tutorial. http://deeplearning.stanford.edu/tutorial/. (дата обращения 25.05.2017).

27. С. Осовский. *Нейронные сети для обработки информации*. Финансы и Статистика, 2004.

ПРИЛОЖЕНИЕ А. РЕАЛИЗАЦИЯ КЛАССИЧЕСКИХ МЕТОДОВ КЛАССИФИКАЦИИ

Модель Word2Vec и её использование для рецензий

```
# kaggle
from bs4 import BeautifulSoup
import re
from nltk.corpus import stopwords
import nltk.data
import numpy as np
from gensim.models import word2vec
from methods import random_forest
import pandas as pd
from sklearn.model_selection import train_test_split
import gensim.models

import logging
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(
    message)s',
                    level=logging.INFO)

tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')


def get_word_vectors(data):
    train, test = train_test_split(data, test_size=0.2,
        random_state=42)

    unlabeled_train = pd.read_csv(
        "unlabeledTrainData.tsv", header=0, delimiter="\t",
            quoting=3)
```

```
sentences = []

for review in train['review']:
    sentences += review_to_sentences(review, tokenizer)

for review in unlabeled_train["review"]:
    sentences += review_to_sentences(review, tokenizer)

model = word2vec.Word2Vec(sentences, workers=4,size=100,
   min_count=40,window=10, sample=1e-3)

model.wv.save_word2vec_format('150_features')

model.init_sims(replace=True)

clean_train_reviews = []
for review in train['review']:
    clean_train_reviews.append(review_to_words(review,
       remove_stopwords=True))
trainDataVecs = getAvgFeatureVecs(clean_train_reviews, model,
    num_features)

clean_test_reviews = []
for review in test['review']:
    clean_test_reviews.append(
        review_to_words(review, remove_stopwords=True))
testDataVecs = getAvgFeatureVecs(clean_test_reviews, model,
   num_features)

return trainDataVecs, testDataVecs, train['sentiment'], test[
   'sentiment']
```

```
def review_to_words(review, remove_stopwords=False):
    review_text = BeautifulSoup(review).get_text()
    review_text = re.sub("[^a-zA-Z]", " ", review_text)
    words = review_text.lower().split()
    if remove_stopwords:
        stops = set(stopwords.words("english"))
        words = [w for w in words if not w in stops]
    return words

def review_to_sentences(review, tokenizer, remove_stopwords=False
    ):
    raw_sentences = tokenizer.tokenize(review.strip())
    sentences = []
    for raw_sentence in raw_sentences:
        if len(raw_sentence) > 0:
            sentences.append(review_to_words(raw_sentence,
                remove_stopwords))
    return sentences


def makeFeatureVec(words, model, num_features):
    featureVec = np.zeros((num_features,), dtype="float32")
    nwords = 0.
    index2word_set = set(model.index2word)
    for word in words:
        if word in index2word_set:
            nwords = nwords + 1.
            featureVec = np.add(featureVec, model[word])
    featureVec = np.divide(featureVec, nwords)
    return featureVec
```

```
def getAvgFeatureVecs(reviews, model, num_features):
    reviewFeatureVecs = np.zeros((len(reviews), num_features),
       dtype="float32")

    for review in reviews:
        reviewFeatureVecs[counter] = makeFeatureVec(review, model
           ,num_features)
    return reviewFeatureVecs

if __name__ == "__main__":
    main()
```

```
from bs4 import BeautifulSoup
import re
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np
from sklearn.model_selection import train_test_split


def review_to_words(raw_review):
    review_text = BeautifulSoup(raw_review, 'lxml').get_text()
    words = re.sub("[^a-zA-Z]", " ", review_text).lower().split()
    stops = set(stopwords.words("english"))
    no_stop_words = [w for w in words if not w in stops]
    return (" ".join(no_stop_words))


def bag_of_words(reviews):
    vectorizer = CountVectorizer(analyzer="word",
                                 tokenizer=None,
                                 preprocessor=None,
                                 stop_words=None,
                                 max_features=5000)
    train_data_features = vectorizer.fit_transform(reviews)
    train_data_features = train_data_features.toarray()
    return train_data_features, vectorizer.get_feature_names()


def data_to_bow(raw):
    reviews = []
    for review in raw['review']:
        reviews.append(review_to_words(review))
```

```
    data_features, vocab = bag_of_words(reviews)
    dist = np.sum(data_features, axis=0)
    return data_features
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import SGDClassifier

from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score


def SGD(x_train, x_test, y_train, y_test):
    print('Running stochastic gradient descent')
    clf = SGDClassifier(loss="hinge", penalty="l2")
    clf = clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)
    y_pred[y_pred == -1] = 0
    print('SGD accuracy: ')
    metrics(y_test, y_pred)


def random_forest(x_train, x_test, y_train, y_test):
    print('Running random forest')
    forest = RandomForestClassifier(n_estimators=100)
    forest = forest.fit(x_train, y_train)
    y_pred = forest.predict(x_test)
    print('Random forest accuracy: ')
    metrics(y_test, y_pred)


def naive_bayes(x_train, x_test, y_train, y_test):
    print('Running naive bayes classifier')
    nb = GaussianNB()
    nb = nb.fit(x_train, y_train)
```

```
    y_pred = nb.predict(x_test)
    print('NB accuracy')
    metrics(y_test, y_pred)


def logistic_regression(x_train, x_test, y_train, y_test):
    print('Running logistic regression')
    lr = LogisticRegression(random_state=1)
    lr = lr.fit(x_train, y_train)
    y_pred = lr.predict(x_test)
    print('LR accuracy')
    metrics(y_test, y_pred)

def metrics(y_test, y_pred):
    accuracy = accuracy_score(y_test, y_pred)
    print(accuracy)
    target_names = ['class 0', 'class 1']
    print(classification_report(y_test, y_pred, target_names=
       target_names))
```

```
from bag_of_words import data_to_bow
from methods import *
from sklearn.model_selection import train_test_split
import pandas as pd
from word2vec_imp import get_word_vectors


def main():
    data = pd.read_csv("rotten_updated.csv", header=0, delimiter=
       ",")
    x_bow = data_to_bow(data)
    y_bow = data['sentiment']
    x_train_bow, x_test_bow, y_train, y_test = train_test_split(
       x_bow, y_bow, test_size=0.2, random_state=42)
    classifiers = [random_forest, logistic_regression,
       naive_bayes, SGD]
    #Running methods for BAG OF WORDS'
    for clf in classifiers:
        clf(x_train_bow, x_test_bow, y_train, y_test)

    x_train_w2v, x_test_w2v, y_train_w2v, y_test_w2v =
       get_word_vectors(data)
    #Running methods for word2vec
    for clf in classifiers:
        clf(x_train_w2v, x_test_w2v, y_train_w2v, y_test_w2v)


if __name__ == "__main__":
    main()
```

ПРИЛОЖЕНИЕ Б. РЕАЛИЗАЦИЯ СВЁРТОЧНОЙ СЕТИ

Граф вычислений для свёрточной нейронной сети

```
import tensorflow as tf
import numpy as np


class ConvolutionalNN(object):
    def __init__(self, sequence_length, vocabulary_size,
       embedding_size, filter_sizes, num_filters, l2_reg_lambda
       =0.0):
        self.x = tf.placeholder(tf.int32, [None, sequence_length
           ], name="x")
        self.y = tf.placeholder(tf.float32, [None, 2], name="y")
        self.dropout_keep_prob = tf.placeholder(tf.float32, name=
           "dropout_keep_prob")

        l2_reg_loss = tf.constant(0.0)

        # Слой векторного представления слов
        with tf.name_scope("embedding"):
            self.W = tf.Variable(tf.random_uniform([
               vocabulary_size, embedding_size], -1.0, 1.0), name
               ="W")
            self.embedded_words = tf.expand_dims(tf.nn.
               embedding_lookup(self.W, self.x), -1)

        pooled_outputs = []

        # Свёрточный слой и слой max-pooling'а для для каждого
           фильтра
        for i, filter_size in enumerate(filter_sizes):
```

```
        with tf.name_scope("conv-maxpool-%s" % filter_size):

            filter_shape = [filter_size, embedding_size, 1,
               num_filters]
            W = tf.Variable(tf.truncated_normal(filter_shape,
                stddev=0.1), name="W")
            b = tf.Variable(tf.constant(0.1, shape=[
               num_filters]), name="b")
            conv = tf.nn.conv2d(self.embedded_words,W,strides
               =[1, 1, 1, 1],padding="VALID",name="conv")

            # max(tf.nn.bias_add(conv, b), 0)
            h = tf.nn.relu(tf.nn.bias_add(conv, b), name="
               relu")

            pooled = tf.nn.max_pool(h, ksize=[1,
               sequence_length - filter_size + 1, 1, 1],
               strides=[1, 1, 1, 1], padding='VALID',name="
               max-pool")
            pooled_outputs.append(pooled)

    # Объединение признаков, полученных в результате пулинга
    num_filters_total = num_filters * len(filter_sizes)
    self.pool = tf.reshape(tf.concat(pooled_outputs, 3), [-1,
        num_filters_total])

    # Добавления дропаута
    with tf.name_scope("dropout"):
        self.drop = tf.nn.dropout(self.pool, self.
           dropout_keep_prob)

    # Final (unnormalized) scores and predictions
    with tf.name_scope("output"):
```

```
            W = tf.get_variable("W",shape=[num_filters_total, 2],
                initializer=tf.contrib.layers.xavier_initializer()
                )
            b = tf.Variable(tf.constant(0.1, shape=[2]), name="b"
                )
            l2_reg_loss += tf.nn.l2_reg_loss(W)
            l2_reg_loss += tf.nn.l2_reg_loss(b)
            self.scores = tf.nn.xw_plus_b(self.drop, W, b, name="
                scores")
            self.predictions = tf.argmax(self.scores, 1, name="
                predictions")

        # Минимизация потерь перекрёстной энтропии и её среднее
            значение
        with tf.name_scope("loss"):
            losses = tf.nn.softmax_cross_entropy_with_logits(
                logits=self.scores, labels=self.y)
            self.loss = tf.reduce_mean(losses) + l2_reg_lambda *
                l2_reg_loss

        # Полученная точность
        with tf.name_scope("accuracy"):
            correct_predictions = tf.equal(self.predictions, tf.
                argmax(self.y, 1))
            self.accuracy = tf.reduce_mean(tf.cast(
                correct_predictions, "float"), name="accuracy")
```

Фрагмент кода обучения нейронной сети

```
import tensorflow as tf
import numpy as np
import ConvolutionalNN
# модуль для получения обработанных данных
import get_data

tf.flags.DEFINE_string("word2vec", None,)
tf.flags.DEFINE_string("filter_sizes", "2,3,4")
tf.flags.DEFINE_integer("num_filters", 128)
tf.flags.DEFINE_integer("embedding_dim", 150)
tf.flags.DEFINE_float("dropout_keep_prob", 0.8)
tf.flags.DEFINE_float("l2_reg_lambda", 4)
tf.flags.DEFINE_integer("batch_size", 50)
tf.flags.DEFINE_integer(
    "num_epochs", 20, "Number of training epochs (default: 100)")
tf.flags.DEFINE_integer("evaluate_every", 100)

FLAGS = tf.flags.FLAGS

...
cnn = ConvolutionalNN(sequence_length=x_train.shape[1],
   num_classes=y_train.shape[1], vocab_size=len(vocab_processor.
   vocabulary_), embedding_size=FLAGS.embedding_dim, filter_sizes
   =list(
    map(int, FLAGS.filter_sizes.split(","))), num_filters=FLAGS.
       num_filters, l2_reg_lambda=FLAGS.l2_reg_lambda)
global_step = tf.Variable(0, name="global_step", trainable=False)
optimizer = tf.train.AdamOptimizer(1e-3)
grads_and_vars = optimizer.compute_gradients(cnn.loss)
train_op = optimizer.apply_gradients(grads_and_vars, global_step=
   global_step)
```

```
batches = get_data.batches(list(zip(x_train, y_train)),
                           FLAGS.batch_size, FLAGS.num_epochs)
sess = tf.Session()
sess.run(tf.initialize_all_variables())
saver = tf.train.Saver()

if FLAGS.word2vec:
    initW = np.random.uniform(-0.25, 0.25,
                              (len(vocab_processor.vocabulary_),
                                 FLAGS.embedding_dim))
    # load any vectors from the word2vec
    print("Load word2vec file {}\n".format(FLAGS.word2vec))
    with open(FLAGS.word2vec, "rb") as f:
        header = f.readline()
        vocab_size, layer1_size = map(int, header.split())
        for line in f:
            line = line.decode('ascii')
            word = line.split(' ', 1)[0]
            idx = vocab_processor.vocabulary_.get(word)
            if idx != None:
                numbers = line.split(' ', 1)[1]
                initW[idx] = np.fromstring(numbers, dtype=float,
                   sep=' ')
    sess.run(cnn.W.assign(initW))

for batch in batches:
    x_batch, y_batch = zip(*batch)
    train_step(x_batch, y_batch)
    current_step = tf.train.global_step(sess, global_step)
    if current_step % FLAGS.evaluate_every == 0:
        print("\nEvaluation:")
        validation_step(x_validation, y_validation,
                        writer=validation_summary_writer)
```

```
            print("")
        if current_step % FLAGS.checkpoint_every == 0:
            path = saver.save(sess, checkpoint_prefix,
                              global_step=current_step)
            print("Saved model checkpoint to {}\n".format(path))


    def train_step(x_batch, y_batch):
        feed_dict = {
            cnn.x: x_batch,
            cnn.y: y_batch,
            cnn.dropout_keep_prob: FLAGS.dropout_keep_prob
        }
        _, step, summaries, loss, accuracy = sess.run(
            [train_op, global_step, train_summary_op, cnn.loss, cnn.
               accuracy],
            feed_dict)
        time_str = datetime.datetime.now().isoformat()
        print("{}: step {}, loss {:g}, acc {:g}".format(
            time_str, step, loss, accuracy))
        train_summary_writer.add_summary(summaries, step)


    def validation_step(x_batch, y_batch, writer=None):
        feed_dict = {
            cnn.x: x_batch,
            cnn.y: y_batch,
            cnn.dropout_keep_prob: 1.0
        }
        step, summaries, loss, accuracy = sess.run(
            [global_step, validation_summary_op, cnn.loss, cnn.
               accuracy],
            feed_dict)
```

```
    time_str = datetime.datetime.now().isoformat()
    print("{}: step {}, loss {:g}, acc {:g}".format(
        time_str, step, loss, accuracy))
    if writer:
        writer.add_summary(summaries, step)
```

ПРИЛОЖЕНИЕ В. РЕАЛИЗАЦИЯ РЕКУРРЕНТНОЙ СЕТИ С LSTM БЛОКАМИ

Граф вычислений для рекуррентной нейронной сети с LSTM блоками

```
import tensorflow as tf

class LSTMNetwork(object):
    def __init__(self, hidden_size, embedding_size,
       vocabulary_size, max_length, learning_rate=0.01):
        # Параметры модели:
        # hidden_size: количество LSTM-модулей в блоке
        # embedding_size: размер векторного представления слов
        # vocabulary_sizfe: количество слов в словаре
        # max_length: максимальная длина входного тензора
        # learning_rate: темп обучения метода оптимизации Adam
        # Построение графа TensorFlow

        # Плэйсхолдер входных данных [batch_size, max_length]
        self.input = tf.placeholder(tf.int32, [None, max_length],
            name='input')

        # Плэйсхолдер длины последовательности, имеющий
          размерность [batch_size]. Содержит длину каждого тензора
          батча
        self.seq_len = tf.placeholder(tf.int32, [None], name='
           lengths')

        # Плэйсхолдер целевых значений [batch_size, 2]
        self.target = tf.placeholder(tf.float32, [None, 2], name=
           'target')

        # коэффициент дропаута
```

```
self.dropout_keep_prob =  tf.placeholder(tf.float32, name
   ='dropout_keep_prob')

# Слой векторного представления слов [vocabulary_size,
   embedding_size]
with tf.name_scope('word_embeddings'):
    embeddings = tf.Variable(tf.random_uniform([
       vocabulary_size, embedding_size], -1, 1, seed=seed
       ))
    self.word_embeddings = embedded_words = tf.nn.
       embedding_lookup(embeddings, x)

# Создание LSTM-слоёв
outputs = embedded_words
for h in hidden_size:
    outputs = self._rnn_layer(h, outputs, seq_len,
       dropout_keep_prob)

outputs = tf.reduce_mean(outputs, reduction_indices=[1])

# Построение полносвязного слоя
with tf.name_scope('final_layer/weights'):
    w = tf.Variable(tf.truncated_normal([self.hidden_size
       [-1], 2]))
    self.variable_summaries(w, 'final_layer/weights')

with tf.name_scope('final_layer/biases'):
    b = tf.Variable(tf.constant(0.1, shape=[2]))
    self.variable_summaries(b, 'final_layer/biases')
# Линейные активации для каждого класса [batch_size, 2]
with tf.name_scope('final_layer/wx_plus_b'):
    self.activations = tf.nn.xw_plus_b(outputs, w, b,
       name='activations')
```

```
        # Софтмакс активации для каждого класса [batch_size, 2]
        self.predict = tf.nn.softmax(self.activations, name='
            predictions')

        # Минимизация потерь перекрестной энтропии [batch_size]
        self.losses = tf.nn.softmax_cross_entropy_with_logits(
            logits=self.activations, labels=self.target, name='
            cross_entropy')

        # Среднее значение потерь перекрестной энтропии
        with tf.name_scope('loss'):
            self.loss = tf.reduce_mean(self.losses, name='loss')
            tf.summary.scalar('loss', self.loss)

        # Минимизация функции градиентного спуска
        self.train_step = tf.train.AdamOptimizer(learning_rate).
            minimize(self.loss)

        # Среднее значение точности для используемого батча
        with tf.name_scope('accuracy'):
            correct_pred = tf.equal(tf.argmax(self.predict, 1),
                tf.argmax(self.target, 1))
            self.accuracy = tf.reduce_mean(tf.cast(correct_pred,
                tf.float32), name='accuracy')
            tf.summary.scalar('accuracy', self.accuracy)

    def _cell(self, hidden_size, dropout_keep_prob, seed=None):
        # Построение LSTM-блока

        lstm_cell = tf.contrib.rnn.LSTMCell(hidden_size,
            state_is_tuple=True)
        dropout_cell = tf.contrib.rnn.DropoutWrapper(lstm_cell,
```

```
            input_keep_prob=dropout_keep_prob,output_keep_prob=
            dropout_keep_prob, seed=seed)
        return dropout_cell

    def _rnn_layer(self, hidden_size, x, seq_len,
        dropout_keep_prob, variable_scope=None):
        # Построение RNN слоя с LSTM-блоками
        with tf.variable_scope(variable_scope, default_name='
            rnn_layer'):
            lstm_cell = self._cell(hidden_size, dropout_keep_prob
                )
            outputs, _ = tf.nn.dynamic_rnn(lstm_cell, x, dtype=tf
                .float32, sequence_length=seq_len)
        return outputs
```

Фрагмент кода обучения нейронной сети

```
import LSTMNetwork
import tensorflow as tf
# модуль для получения обработанных данных
import get_data

tf.flags.DEFINE_integer('train_steps', 1000)
tf.flags.DEFINE_integer('batch_size', 100)
tf.flags.DEFINE_integer('train_steps', 300)
tf.flags.DEFINE_integer('embedding_size', 75)
tf.flags.DEFINE_integer('hidden_size', 75)
tf.flags.DEFINE_float('learning_rate', 0.1)
tf.flags.DEFINE_float('test_size', 0.2)
tf.flags.DEFINE_float('dropout_keep_prob', 0.5)
tf.flags.DEFINE_integer('sequence_len', None)

FLAGS = tf.flags.FLAGS
...

rnn = LSTMNetwork(hidden_size=[FLAGS.hidden_size], embedding_size
   =FLAGS.embedding_size, vocabulary_size=get_data.vocab_size,
   max_length=get_data.sequence_len, learning_rate=FLAGS.
   learning_rate)
sess = tf.Session()
sess.run(tf.initialize_all_variables())
saver = tf.train.Saver()
x_validation, y_validation, validation_seq_len = get_data.
   get_validation_data()
train_writer = tf.summary.FileWriter('logs/train')
validation_writer = tf.summary.FileWriter('logs/validation')
train_writer.add_graph(rnn.input.graph)

for i in range(FLAGS.train_steps):
```

```
    # Perform training step
    x_train, y_train, train_seq_len = get_data.batch(FLAGS.
       batch_size)
    train_loss, _, summary = sess.run([rnn.loss, rnn.train_step,
       tf.summary.merge_all()],
      feed_dict={rnn.input: x_train,
       rnn.target: y_train,
       rnn.seq_len: train_seq_len,
       rnn.dropout_keep_prob: FLAGS.dropout_keep_prob})
    train_writer.add_summary(summary, i)

    if (i + 1) % FLAGS.validate_every == 0:
        validation_loss, accuracy, summary = sess.run([rnn.loss,
           rnn.accuracy, tf.summary.merge_all()],
          feed_dict={rnn.input: x_validation,
            rnn.target: y_validation,
            rnn.seq_len: validation_seq_len,
            rnn.dropout_keep_prob: 1})
        validation_writer.add_summary(summary, i)
```