Санкт-Петербургский государственный университет

Кафедра системного программирования

Соколов Ярослав Сергеевич

Генерация фрагментов кода по описаниям на естественном языке

Курсовая работа

Научный руководитель:
к.т.н., доцент Брыксин Т.А.

Санкт-Петербург
2019

Оглавление

Введение

Программное обеспечение с каждым днём играет всё большую роль в автоматизации ручных процессов. Для создания программного обеспечения человеку необходимо обладать специфическими знаниями и навыками. Выходит, что на данный момент большинство людей не способны написать программы, воплощающие их идеи в реальность.

Одно из направлений снижения требований к человеку, который хочет создать программу, является построение алгоритмов синтеза кода. Такие алгоритмы принимают на вход описание в удобном для человека формате и выдают код на требуемом языке программирования.

На данный момент существуют решения, генерирующие код на DSL [10, 19]. Однако, такие языки программирования позволяют решать гораздо меньший круг задач, нежели языки программирования общего назначения. Задача же синтеза кода на языке общего назначения на данный момент является крайне трудной.

С развитием машинного обучения в других областях всё чаще данные подходы начинают применяться и в области синтеза кода. В особенности методы, использующие в своей основе глубокие нейронные сети, которые успели хорошо зарекомендовать себя в задачах генерации структурированных объектов [11, 8], таких как изображения [5, 16], тексты [7], структуры белка [4], звук [26] и др.

Следуя успехам соревнований в других задачах машинного обучения, начали появляться соревнования, задачей в которых является синтез кода по описанию на естественном языке. В частности, работа [24] посвящена сбору данных, подходящих под такую задачу, которые легли в основу соревнования CoNaLa - Code Natural Language Challenge [18]. Подобные соревнования позволяют исследователям сосредоточить свои силы на создании новых методов синтеза кода, не думая о сборе данных и тщательном сравнении результатов с существующими моделями.

Постановка задачи

Цель работы заключается в улучшении существующих подходов генерации кода. Для достижения этой цели сформулированы следующие задачи:

1. принять участие в соревновании;
2. проанализировать подходы машинного перевода и генерации кода;
3. сравнить и применить лучшие из них в соревновании;
4. улучшить один из подходов.

1. Обзор

1.1. Соревнование CoNaLa: The Code/Natural Language Challenge

В соревновании перед участниками ставится задача синтеза фрагмента кода на языке программирования Python по описанию на английском языке. Результат оценивается типичной для задачи машинного перевода метрикой *BLEU* (bilingual evaluation understudy) [2], умноженной на 100. Такая метрика, по сути, является процентом совпадающих n-грамм между реальным примером и гипотезой. При этом никак не учитывается, корректен ли сгенерированный код.

Участникам предоставляются три множества данных: тренировочное, тестовое и множество данных, автоматически собранных из интернета. Количество примеров в каждом множестве соответственно: 2379, 500, 598237. Данные из тренировочного и тестового множеств были составлены людьми вручную путем переписывания собранных из интернета данных. В третьем множестве данных каждому из примеров сопоставляется уверенность достоверности этого примера, процесс сбора данных подробно описан в статье [15].

поле	**значение**
пример 1	
описание	Two values from one input in python?
переписанное описание	assign values to two variables, ‘var1‘ and ‘var2‘ from user input response to ‘’Enter two numbers here: ‘ split on whitespace
фрагмент кода	var1, var2 = input(’Enter two numbers here: ’).split()
пример 1	
описание	how to sort by length of string followed by alphabetical order?
переписанное описание	sort list ‘the_list‘ by the length of string followed by alphabetical order
фрагмент кода	the_list.sort(key=lambda item: (-len(item), item))

Таблица 1: Примеры из множества данных, собранных вручную

поле	значение
пример 1	
описание	Summing 2nd list items in a list of lists of lists?
фрагмент кода	[sum([x[1] for x in i]) for i in data]
вероятность	0.668
пример 2	
описание	Customize x-axis in matplotlib
фрагмент кода	plt.show()
вероятность	0.613
пример 3	
описание	converting string to tuple
фрагмент кода	[tuple(int(i) for i in el.strip('()').split(',')) for el in s.split('),(')]
вероятность	0.608

Таблица 2: Примеры из множества данных, собранных автоматически

1.2. Вероятностные подходы машинного перевода

На данный момент все архитектуры нейронных сетей для задачи машинного перевода имеют общий вид: кодер-декодер. Кодер преобразует входную последовательность на оригинальном языке в некоторое векторное представление, которое затем передается декодеру для преобразования в последовательность на целевом языке. Архитектуры кодера и декодера, в свою очередь, принимают различный вид.

Наиболее известны три основные архитектуры кодера и декодера: рекуррентные [14], сверточные [6] нейронные сети и *Transformer* [1]. Наилучшее качество в большинстве задач обработки естественных языков на данный момент достигается с использованием архитектуры *Transformer*, хотя рекуррентные нейросети до сих пор остаются популярными.

1.2.1. Рекуррентные нейронные сети

Рекуррентные нейронные сети представляют собой архитектуру, позволяющую обрабатывать последовательности переменной длины. Это достигается за счет передачи вектора внутреннего состояния сети в себя

же. Таким образом, сеть, принимая уже обработанную часть последовательности в виде скрытого состояния, на каждом шаге принимает решение о модификации скрытого состояния с учетом обрабатываемой на данный момент единицы последовательности. Иллюстрация рекуррентной нейросети представлена на рисунке (1).

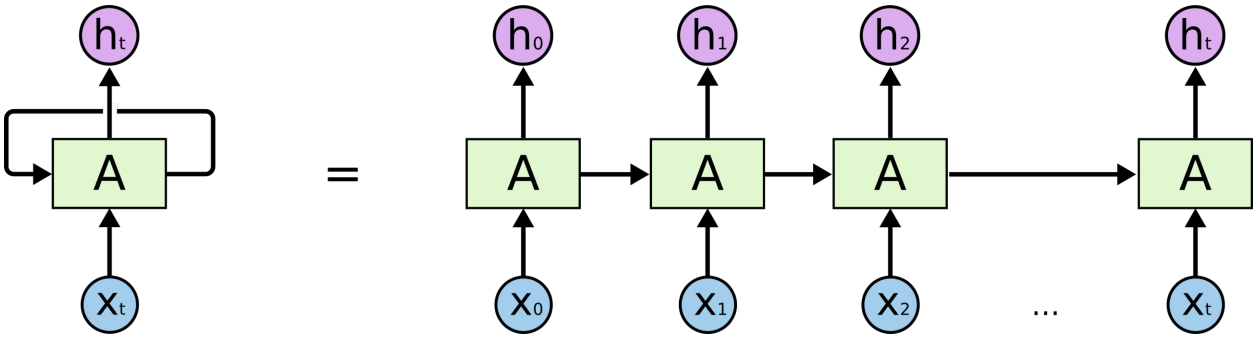

Рис. 1: Рекуррентная нейронная сеть слева и ее развернутое представление справа. Здесь $\{x_n\}_{n=0}^{t}$ это обрабатываемая последовательность, $\{h_n\}_{n=0}^{t}$ выходная последовательность, а A – параметризованная функция (блок нейросети). Связь между блоками обозначает передачу скрытых параметров.

1.2.2. Transformer архитектура нейронной сети

Transformer архитектура также позволяет работать с последовательностями переменной длины. Основное отличие *Transformer* от рекуррентных нейросетей заключается в представлении и обработке входной последовательности. Если рекуррентные сети последовательно обрабатывают входную последовательность, то *Transformer* делает это параллельно, порождая из n единиц входа n скрытых состояний. Также качественное отличие заключается во внутреннем устройстве архитектуры *Transformer*. В ней, например, используются более продвинутые механизмы внимания [3].

Такая параллельность обработки устраняет сразу несколько недостатков, присущих рекуррентным нейросетям: затухание или взрыв градиентов, потеря информации о первых элементах последовательности и низкая степень параллелизации [13].

1.3. Вероятностные подходы генерации кода

1.3.1. Классификация подходов

В обзорной работе [24] предлагается следующая классификация вероятностных подходов генерации кода.

Модели уровня токенов: данный класс моделей очень популярен благодаря своей простоте. В них делается предположение, что код является последовательностью токенов $c_1, c_2, ..., c_n$. Генерация происходит пошагово ввиду экспоненциального числа возможных вариантов кода длины n. Так, если множество возможных при генерации токенов, так называемый словарь, V имеет размер $|V|$, то количество всевозможных последовательностей кода длины n равно $|V|^n$. Поэтому большинство моделей генерируют последовательности итеративно. Модели строят вероятностное распределение $P(c_m|c_1c_2...c_{m-1})$, где вероятность каждого следующего генерируемого токена зависит от того, какие токены были сгенерированы до этого. Так как модели учатся генерировать код по описанию i, которое ему подается на вход, моделируемое распределение принимает вид $P(c_m|c_1c_2...c_{m-1}, i)$.

На сегодня именно такие модели показывают наилучшие результаты в задачах генерации текстов на естественном языке.

Синтаксические модели: модели из этого класса генерируют код в виде абстрактного синтаксического дерева. Они генерируют один узел за другим от корня сверху вниз и слева направо. В таких моделях также строится распределение $P(c_m|c_1c_2...c_{m-1}, i)$ каждого узла c_m, опираясь только на уже сгенерированные узлы $c_1, c_2, ..., c_{m-1}$ и описание i.

Преимущество таких моделей состоит в том, что они генерируют синтаксически корректный код. Однако обучение и применение таких моделей занимает гораздо больше времени по сравнению с предыдущим классом.

Семантические модели: такие модели генерируют программный код в виде графа тем или иным образом. Графовым представлением кода может быть, к примеру, граф потока данных в программе.

На данный момент не существует подходов синтеза кода из этого

класса.

1.3.2. Существующие подходы

Один из современных подходов генерации кода по описанию на естественном языке описан в статье [12]. Модель из этой статьи относится к классу моделей уровня токенов. В ней авторы предлагают решение, которое позволяет переносить имена переменных из запроса пользователя в генерируемый код.

В другой статье [27] предлагается использовать преимущества модели уровня токенов и синтаксической модели одновременно. Для этого код был представлен в виде абстрактного синтаксического дерева и затем преобразован в последовательность команд, по которым можно однозначно восстановить дерево. Таким образом, вероятностная модель решает задачу генерации последовательности команд. Скорость работы такой модели не замедляется, но по полученной последовательности команд можно восстановить синтаксическое дерево и, как следствие, синтаксически корректный код.

Авторы последней статьи улучшили [23] свой подход путем использования идеи частичного обучения без учителя на легкодоступном множестве описаний кода на английском языке с помощью нейросетевого фреймворка VAE (variational autoencoder) [11].

Все эти подходы используют в своей основе нейросети вида кодер-декодер с рекуррентными архитектурами кодера и декодера.

Также стоит упомянуть работу [29] с нашей кафедры. Автор поставил перед собой задачу, заключающуюся в создании инструмента для генерации кода для работы с API по описанию на английском языке. Хотя такая постановка задачи и имеет свои преимущества, однако она сильно сужает область применения полученного инструмента. По этой же причине предложенный подход нельзя использовать и в соревновании *CoNaLa*.

2. Описание реализации

2.1. Выбор архитектуры

На данный момент наилучшее качество в задаче машинного перевода показывают модели с *Transformer* архитектурой кодера и декодера. Однако, насколько нам известно, еще не было исследований, где для задачи генерации кода применялась бы архитектура *Transformer*. В таких исследованиях традиционно применяют рекуррентные нейронные сети, которые не только показывают худший результат в задаче машинного перевода, но и имеют ряд других недостатков: низкая степень параллелизации, настабильный процесс обучения и нестабильное поведение на слишком длинных последовательностях [1]. Таким образом было принято решение использовать архитектуру *Transformer*.

На рисунке (2) хорошо прослеживаются преимущества *Transformer* архитектуры. Модель быстрее, стабильнее и лучше сходится во время тренировки.

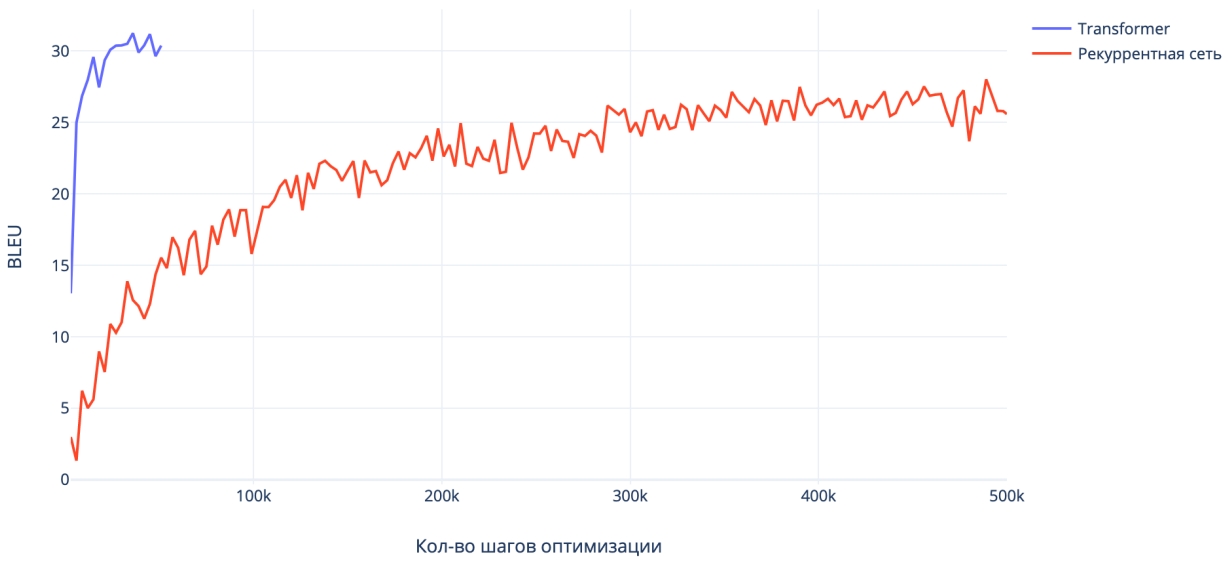

Рис. 2: Пример процесса обучения *Transformer* и рекуррентной нейросети на одинаковых данных. Значение метрики BLEU в зависимости от количества шагов обучения нейросети. Шаг у *Transformer* в 3 раза дольше, чем шаг рекуррентной сети

2.2. Построение валидационного множества

Окончательная проверка результатов в соревновании производится на тестовом множестве примеров. Так, для вычисления *BLEU* решения участник предоставляет фрагмент кода для каждого примера из тестового множества. После чего решение участника сравнивается с эталонным и считается оценка.

Общепринятой практикой в машинном обучении является построение валидационного множества для тестирования гипотез и выбора лучшей модели. При этом для корректности получаемых оценок на валидационном множестве необходимо построить его таким образом, чтобы данные в нем были из того же распределения, что и данные тестового множества.

Данные для тестового множества были составлены организаторами вручную. Таким же образом была написана часть тренировочных данных. Для получения несмещенной оценки своего решения на валидационном множестве, необходимо составить его так, чтобы оно было похоже на тестовое множество. Поэтому все примеры валидационного множества были выбраны из вручную написанного множества, а не из его комбинации с автоматически собранным множеством.

Также было принято решение очистить данные от дубликатов. Дубликатом считаются примеры, в которых совпадает либо описание кода, либо фрагмент кода. Это гарантирует отсутствие одинаковых примеров в тренировочном и валидационном множестве.

Размер валидационного множества был выбран аналогично размеру тестового множества: 500 примеров.

Стоит упомянуть проблему сильного расхождения оценок на валидационном и тестовом множествах по абсолютному значению, причины которой так и не удалось выяснить. Однако эти оценки все равно коррелировали между собой, что позволяло принимать решения, базируясь на оценке на валидационном множестве.

2.3. Предобработка данных

Как и в любой другой задаче машинного обучения, данные необходимо предобработать перед началом обучения моделей. От этого зависит как качество обучения, так и стабильность получаемых результатов.

2.3.1. Объем автоматически собранных данных

Данные, предоставленные участникам соревнования *CoNaLa*, не всегда корректные. Например, в первом примере автоматически собранного множества (2) фрагмент кода не соответствует описанию. Сами авторы соревнования в своем решении отсеивают автоматически собранные данные по уверенности классификатора, представленной для каждого примера. Эмпирически было установлено, что оптимальным является использование 30 000 примеров с максимальной вероятностью.

2.3.2. Фильтрация примеров

Также были отсеяны примеры с слишком длинными или слишком короткими описаниями или фрагментами кода по 5-ому и 95-ому перцентилям. Если среди примеров необычной длины были корректные, то вероятностные подходы всё равно не смогли бы описать их распределение ввиду их малого количества.

Затем было принято решение заменить по возможности юникодные символы на аналогичные ASCII символы. Если аналогичных символов не существует, то пример отсеивался. Это позволило увеличить среднее количество появления каждого токена во фрагментах кода и сократить количество различных токенов в словаре.

2.3.3. Нормализация описаний кода

Описания кода представлены в виде предложений на английском языке. Для такого рода данных существуют общепринятые техники нормализации: выделение основы слов, удаление стоп-слов, удаление

пунктуации, приведение в нижний регистр.

Специфичным для текущей задачи является наличие в предложениях описаний переменных и других объектов, которые пользователь ожидает увидеть в сгенерированном коде. Такие части заключены в кавычки. Совсем не ясно, необходимо ли проводить нормализацию для такой части предложений. Было выделено и протестировано несколько методов нормализации текста:

1. без нормализации;
2. нормализация текста, кроме описания переменных;
3. нормализация всего текста;
4. нормализация текста, замена описания переменных на ключевые токены;
5. нормализация текста, удаление описания переменных.

способ	пример	BLEU
1 способ	create a matrix from a list ‘ [1 , 2 , 3] ‘	11.76
2 способ	creat matrix list ‘ [1 , 2 , 3] ‘	12.96
3 способ	creat matrix list ‘ 1 2 3 ‘	13.56
4 способ	creat matrix list VAR	11.38
5 способ	creat matrix list	11.27

Таблица 3: Влияние способа нормализации на текст описания и метрику соревнования

Из таблицы (3) видно, что оптимальным является третий способ нормализации, который и решено было использовать.

2.3.4. Нормализация фрагментов кода

Предобработка кода производилась аналогично решению авторов соревнования. Код проверялся на синтаксическую корректность с помощью библиотеки *astor* [28]. Все примеры с некорректным кодом удалялись из тренировочного множества.

2.4. Аугментация данных

После предобработки остается довольно малое количество примеров: 19 767. В самом популярном параллельном корпусе из английских и немецких предложений, для сравнения, содержится 4.5 миллиона примеров [22]. Справиться с недостатком данных можно с помощью общеизвестной для работы с естественными языками техники аугментации данных. Она заключается в удалении одного из слов либо в описании, либо в коде. Качество примеров ухудшается, однако их количество возрастает во много раз.

тип аугментации	размер тренировочного мн-ва	BLEU
без аугментации	19767	13.56
только описания	154394	12.42
описания и фрагменты кода	2259076	13.97

Таблица 4: Количество примеров и BLEU в зависимости от способа аугментации

Некоторое улучшение (4) дает аугментация описаний и фрагментов кода. Было принято решение аугментировать описания и фрагменты кода.

2.5. Построение словаря

Словарь описаний на английском языке был построен стандартным для работы естественным языком методом: каждое слово и знак препинания после предобработки были рассмотрены как отдельный токен. Токен добавлялся в словарь, если он встречался в тренировочном множестве определенное количество раз.

Токенизация кода на языке *Python* производилась так же, как и в решении авторов соревнования. Использовалась библиотека astor, которая токенизировала код по формальной грамматике языка Python.

Без экспериментов эти пороги задать довольно затруднительно. Если выставить его слишком большим, то модель не будет иметь части слов в своем словаре и не будет их использовать вовсе, что сильно понижает ее выразительную способность. Однако при слишком низком

пороге в словарь начнут попадать токены, встречающиеся настолько редко в тренировочном множестве, что модель не сможет выучить их семантику и будет применять их неадекватно.

	порог				
	3	5	7	10	15
BLEU	15.42	13.97	13.91	13.74	10.49

Таблица 5: BLEU в зависимости от порога встречаемости токена для его вхождения в словарь

Наилучшее качество показывают модели с порогом вхождения токена в словарь 3. Важно отметить, что на самом деле при таком пороге самые редкие токены встречаются больше 3 раз из-за аугментации данных. На практике, при увеличении множества данных в 100 раз, следует рассчитывать, что встречаемость самого редкого токена из словаря будет примерно равна 300.

2.6. Использование предобученных векторных представлений

Известна общепринятая техника в обработке естественных языков: использование предобученных векторных представлений для слов. Такие представления обучаются отдельно на большом объеме данных. Основным их преимуществом является близость в векторном пространстве похожих по смыслу слов, что позволяет модели, которая использует такое векторное представление, обладать знаниями о похожести слов с самого начала своего обучения.

Для текущего набора данных есть возможность использовать предобученные представления для слов на английском языке. Наиболее известное из таких представлений *Global Vectors for Word Representation* [9]. Именно его и решено было опробовать.

Векторные представления немного улучшают качество модели (6), так что было решено их использовать.

векторное представление	BLEU
без векторного представления	15.42
GLoVe 100d	15.56
GLoVe 300d	15.72

Таблица 6: BLEU в зависимости от используемого предобученного векторного представления для английских слов. Представления имеют размерности 100 и 300

2.7. Реализация нейронной сети

Для разработки нейронных сетей существует множество фреймворков. Наиболее популярными являются *PyTorch* [20] и *Tensroflow* [25]. Однако эти фреймворки довольно низкоуровневые: в них доступен только базовый набор слоев нейронной сети. Существуют специализированные фреймворки для машинного перевода, где уже реализованы популярные архитектуры. Было решено использовать *OpenNMT−tf* [17], так как там уже реализован *Transformer*.

Было решено использовать фреймворк внутри *Docker* контейнера, так как он зависит от старых версий библиотек.

Были написаны скрипты на языке *Python* для предобработки и токенизации примеров данных, для подготовки посылки в соревновании. Для запуска тренировки и тестирования нейросети были написано скрипты на языке bash.

3. Апробация

Была обучена модель *Transformer*. Качества предсказаний удалось улучшить путем ограничения длины генерируемой последовательности снизу. Для этого в ходе генерации первые k шагов игнорируются стоп-токены, означающие конец генерируемой последовательности. Качество получаемых результатов в зависимости от параметра k отражены в таблице (7).

	k					
	1	5	7	10	12	15
BLEU	15.72	15.73	15.95	16.33	16.69	16.11

Таблица 7: BLEU в зависимости от ограничения длины последовательности снизу

Были сделаны посылки на соревнование и получены результаты, описанные в таблице (8).

Модель	BLEU
Рекуррентная сеть (авторы)	14.72
Transformer (наш)	16.69
Рекуррентная сеть + AST (лучшее)	26.4

Таблица 8: Результаты участия в соревновании

Последняя модель [23] в таблице (8) сильно превосходит наше решение и была описана в обзоре (1.3.2). Это решение было опубликовано в марте, из-за чего мы не успели использовать его преимущества. Однако там также можно применить *Transformer* вместо рекуррентной сети для повышения качества.

Среди моделей уровня токенов *Transformer* показывает наилучший результат.

Заключение

В рамках данной работы были получены следующие результаты.

- Произведен анализ существующих подходов к генерации кода.
- Произведен анализ существующих решений в задаче машинного перевода.
- Выявлены недостатки подходов в задаче генерации кода по сравнению с подходами к машинному переводу.
- Произведена предобработка данных.
- Построена модель с Transformer архитектурой.
- Подобраны оптимальные параметры для обучения модели.
- Улучшены подходы уровня токенов с помощью применения Transformer архитектуры.

Из полученных в соревновании результатов можно сделать вывод о том, что хоть архитектура нейронной сети и играет роль в качестве решения, но улучшения представления кода играет гораздо большую роль в достижении лучших результатов. Однако применение современных архитектур позволяет быстрее обучать модели и проверять больше гипотез за то же время. Возможно, это поможет исследователям быстрее получать результаты в области генерации кода.

3.1. Возможные улучшения

В последние годы в области обработки естественных языков исследователи активно строят большие модели, обученные на огромном объеме данных, которые затем настраиваются для конкретной задачи отдельно. Такой подход может помочь с проблемой недостатка данных в области генерации кода.

В данной работе не производилась нормализация фрагментов кода и его токенизация была довольно примитивной. Из-за чего, например,

часть кода [::2], имеющая хорошую семантику целиком, разбивалась на токены [, :, :, 2,], которые могут плохо описываться моделями по отдельности. Данную проблему можно попытаться решить довольно популярной в последнее время техникой: bytepair encoding [21]. В данной технике за основу словаря берутся всевозможные значения байтов, и затем в него добавляются комбинации байтов, если их частота в наборе данных выше определенного порога. Таким образом в подобных примерах несколько токенов могут объединяться в один семантически более целостный.

Список литературы

[1] Attention is all you need / Ashish Vaswani, Noam Shazeer, Niki Parmar et al. // Advances in neural information processing systems. — 2017. — P. 5998–6008.

[2] BLEU: a method for automatic evaluation of machine translation / Kishore Papineni, Salim Roukos, Todd Ward, Wei-Jing Zhu // Proceedings of the 40th annual meeting on association for computational linguistics / Association for Computational Linguistics. — 2002. — P. 311–318.

[3] Bahdanau Dzmitry, Cho Kyunghyun, Bengio Yoshua. Neural machine translation by jointly learning to align and translate // arXiv preprint arXiv:1409.0473. — 2014.

[4] Computational protein design with deep learning neural networks / Jingxue Wang, Huali Cao, John ZH Zhang, Yifei Qi // Scientific reports. — 2018. — Vol. 8, no. 1. — P. 6349.

[5] Conditional image generation with pixelcnn decoders / Aaron Van den Oord, Nal Kalchbrenner, Lasse Espeholt et al. // Advances in neural information processing systems. — 2016. — P. 4790–4798.

[6] Convolutional sequence to sequence learning / Jonas Gehring, Michael Auli, David Grangier et al. // Proceedings of the 34th International Conference on Machine Learning-Volume 70 / JMLR. org. — 2017. — P. 1243–1252.

[7] Dynamic evaluation of neural sequence models / Ben Krause, Emmanuel Kahembwe, Iain Murray, Steve Renals // arXiv preprint arXiv:1709.07432. — 2017.

[8] Generative adversarial nets / Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza et al. // Advances in neural information processing systems. — 2014. — P. 2672–2680.

[9] GloVe: Global Vectors for Word Representation // official webpage. — URL: https://nlp.stanford.edu/projects/glove/ (online; accessed: 13.05.2019).

[10] Gulwani Sumit, Marron Mark. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation // Proceedings of the 2014 ACM SIGMOD international conference on Management of data / ACM. — 2014. — P. 803–814.

[11] Kingma Diederik P, Welling Max. Auto-encoding variational bayes // arXiv preprint arXiv:1312.6114. — 2013.

[12] Latent Predictor Networks for Code Generation / Wang Ling, Edward Grefenstette, Karl Moritz Hermann et al. // CoRR. — 2016. — Vol. abs/1603.06744. — 1603.06744.

[13] Learning long-term dependencies with gradient descent is difficult / Yoshua Bengio, Patrice Simard, Paolo Frasconi et al. // IEEE transactions on neural networks. — 1994. — Vol. 5, no. 2. — P. 157–166.

[14] Learning phrase representations using RNN encoder-decoder for statistical machine translation / Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre et al. // arXiv preprint arXiv:1406.1078. — 2014.

[15] Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow / Pengcheng Yin, Bowen Deng, Edgar Chen et al. // International Conference on Mining Software Repositories. — MSR. — ACM, 2018. — P. 476–486.

[16] Oord Aaron van den, Kalchbrenner Nal, Kavukcuoglu Koray. Pixel recurrent neural networks // arXiv preprint arXiv:1601.06759. — 2016.

[17] OpenNMT-tf // github.com. — URL: https://github.com/OpenNMT/OpenNMT-tf (online; accessed: 16.04.2019).

[18] Pengcheng Yin Edgar Chen Bogdan Vasilescu Graham Neubig. CoNaLa: The Code/Natural Language Challenge // Страница соревнования. — 2018. — URL: https://conala-corpus.github.io (online; accessed: 16.04.2019).

[19] Program synthesis using natural language / Aditya Desai, Sumit Gulwani, Vineet Hingorani et al. // Proceedings of the 38th International Conference on Software Engineering / ACM. — 2016. — P. 345–356.

[20] PyTorch // official webpage/. — URL: https://pytorch.org/ (online; accessed: 13.05.2019).

[21] Sennrich Rico, Haddow Barry, Birch Alexandra. Neural Machine Translation of Rare Words with Subword Units // CoRR. — 2015. — Vol. abs/1508.07909. — 1508.07909.

[22] The Stanford Natural Language Processing Group project page // official webpage. — URL: https://nlp.stanford.edu/projects/nmt/ (online; accessed: 13.05.2019).

[23] StructVAE: Tree-structured Latent Variable Models for Semi-supervised Semantic Parsing / Pengcheng Yin, Chunting Zhou, Junxian He, Graham Neubig // CoRR. — 2018. — Vol. abs/1806.07832. — 1806.07832.

[24] A Survey of Machine Learning for Big Code and Naturalness / Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, Charles A. Sutton // CoRR. — 2017. — Vol. abs/1709.06182. — 1709.06182.

[25] TensorFlow // official webpage. — URL: https://www.tensorflow.org/ (online; accessed: 13.05.2019).

[26] WaveNet: A generative model for raw audio. / Aäron Van Den Oord, Sander Dieleman, Heiga Zen et al. // SSW. — 2016. — Vol. 125.

[27] Yin Pengcheng, Neubig Graham. A Syntactic Neural Model for General-Purpose Code Generation // CoRR. — 2017. — Vol. abs/1704.01696. — 1704.01696.

[28] astor library // github.com. — URL: `https://github.com/berkerpeksag/astor` (online; accessed: 13.05.2019).

[29] Евгеньевич Чебыкин Александр. Синтез программного кода с использованием машинного обучения / Чебыкин Александр Евгеньевич ; Санкт-Петербургский государственный университет. — 2018.