Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к выпускной квалификационной работе

«Индукция алгоритмов с помощью Тьюринг-полных сетей глубокого обучения»

Автор: Коликов Данил Александрович _____
Направление подготовки (специальность): 01.03.02 Прикладная математика и информатика
Квалификация: Бакалавр
Руководитель: Потапов А.С., докт. техн. наук, проф. _____

К защите допустить
Зав. кафедрой Васильев В.Н., докт. техн. наук, проф. _____

«___» _____ 20___ г.

Санкт-Петербург, 2018 г.

Студент Коликов Д.А. **Группа** М3439 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования

Направленность (профиль), специализация Математические модели и алгоритмы в разработке программного обеспечения

Консультанты:

а) Потапов А.С., докт. техн. наук, проф. _____

Квалификационная работа выполнена с оценкой _____

Дата защиты «___» _____ 20___ г.

Секретарь ГЭК *Павлова О.Н.* Принято: «___» _____ 20___ г.

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Исследования последних лет показали, что с использованием нейронных сетей можно находить алгоритмы обработки как конечных данных (например, изображений), так и последовательностей (временных рядов, аудиозаписей). В то же время обработать данные произвольной структуры (например, графов) при помощи стандартных архитектур довольно сложно. В связи с этим возникает задача расширения возможностей нейронных сетей, что позволит выводить алгоритмы обработки данных произвольной структуры.

Целью данной работы является разработка способа индукции подобных алгоритмов. В качестве исследуемого подхода было выбрано проектирование архитектур нейронных сетей для конкретных задач с использованием дифференцируемых языков программирования. Эта область относительно молода, и существующие языки данного класса основаны на императивной парадигме. В данной работе было изучено применение функционального подхода, что позволило расширить возможности нейронных сетей, позволив им выводить алгоритмы обработки данных произвольной структуры.

Результатом исследования является разработка функционального дифференцируемого языка программирования **FNN**. Написанные на нем программы компилируются в нейронные сети, способные обучаться алгоритмам обработки данных конечной или рекурсивной структуры. Данный подход был протестирован на ряде задач и показал свою работоспособность. Исследование опубликовано в сборнике тезисов докладов Конгресса Молодых Учёных 2018 [1].

Первая глава работы посвящена обзору существующих подходов к индукции алгоритмов при помощи машинного обучения. В данной главе рассматриваются ограничения известных архитектур и приводится обоснование выбора дифференцируемых языков программирования в качестве исследуемого подхода.

Во второй главе приводится теоретический анализ соответствия между лямбда-исчислением и архитектурами нейронных сети. В данной главе определяется способ представления структурированных данных, рассматривается алгоритм построения нейронных сетей на основе лямбда-выражения и дается определение полиморфных нейронных сетей.

Третья глава содержит практическое применение разработанной теории. В ней описывается дифференцируемый функциональный язык программиро-

вания **FNN**, спроектированный на основе результатов второй главы. Также в третьей главе рассматриваются технические аспекты процесса компиляции программ в нейронные сети и приводятся примеры экспериментов.

В заключении описывается текущее состояние разработанной теории и приводятся дальнейшие пути ее развития.

ГЛАВА 1. ИНДУКЦИЯ АЛГОРИТМОВ

Рост объема производимых человечеством данных ускоряется с каждым годом. Для интеллектуального анализа подобной информации требуется разрабатывать специализированные алгоритмы, что требует больших затрат времени. Поэтому возникает потребность в выводе подобных алгоритмов в автоматическом режиме.

Существующие подходы к выводу алгоритмов можно разделить на две категории: синтез алгоритмов и индукцию. Первый подход подразумевает вывод кода программы на основе формального определения того, что она должна делать. В данной работе исследуется второй подход – индукция, то есть вывод алгоритма на основе входных данных программы и ее ожидаемых ответов.

Для решения задачи индукции алгоритмов предлагается использовать машинное обучение, и в частности нейронные сети – один из классических, но перспективных классов моделей машинного обучения. На данный момент нейронные сети показывают высокую точность при решении определенных классов задач, однако возможности нейронных сетей ограничены тем, что с их помощью сложно обрабатывать данные произвольной структуры.

Целью данной работы является разработка подхода к индукции алгоритмов обработки данных произвольной структуры с использованием нейронных сетей.

1.1. Актуальность задачи индукции алгоритмов

В настоящее время машинное обучение успешно применяется для решения довольно широкого класса задач. Главное отличие данного подхода от привычного программирования состоит в том, что алгоритм, решающий конкретную задачу, не задается изначально, а находится в процессе обучения. Для его построения используются средства математической статистики, теории вероятности и численных методов. Преимущество данного подхода состоит в том, что в процессе обучения обнаруживаются эмпирические закономерности в данных, благодаря чему становится возможно решать задачи, которые прежде были под силу только человеку. В их число входят классификация изображений, автоматический перевод текстов, обработка естественных языков. Наилучшую точность при решении подобных задач показывают нейронные сети.

Однако класс задач, который может быть решен с их помощью, хоть и широк, но все же ограничен. Распространенные архитектуры нейронных сетей

способны обрабатывать данные конечного размера (к примеру, изображения) или последовательности (временные ряды, предложения на естественном языке). Данные из реального мира зачастую имеют сложную, нелинейную структуру (к примеру, графы друзей в социальных сетях), и обрабатывать их при помощи стандартных архитектур довольно тяжело.

В связи с этим возникает задача разработки подхода к проектированию нейронных сетей, которые способны выводить алгоритмы обработки подобных данных. Это позволит существенно расширить класс задач, которые возможно решить с использованием методов машинного обучения.

1.2. Обзор существующих подходов

Первой работой, заложившей основы для исследования нейронных сетей, можно считать опубликованную в 1943 г. статью Уоррена Мак-Каллока и Вальтера Питтса «Идеи логических вычислений в нервной деятельности» [2]. В данной работе рассматривалась математическая модель, описывающая нейрон головного мозга человека, и описывались ее возможности для моделирования логических функций.

Дальнейшие исследования показали, что возможности моделей, построенных по аналогии с нейронами головного мозга, не ограничиваются моделированием логических функций. Одна из наиболее простых архитектур нейронных сетей – сеть прямого распространения, состоящая из нескольких слоев нейронов, соединенных между собой, уже может быть обучена классифицировать объекты. Нейронные сети с большим числом слоев в свою очередь способны обучаться интеллектуальному анализу данных, распознаванию образов и прогнозированию [3].

Тенденция последних лет заключается в использовании так называемых «глубоких» нейронных сетей (или «глубокого обучения», в англоязычной литературе «Deep Learning»). Под «глубиной» подразумевается то, что такие сети содержат большее число слоев по сравнению с теми, что были использованы прежде. Подобные модели обладают весомым преимуществом – они показывают значительно лучшие результаты по сравнению с «неглубокими» сетями и в ряде задач уже приблизились к человеческому уровню точности. Однако обучать такие сети намного сложнее – они требуют больший объем данных для поиска закономерностей и склонны к переобучению. Несмотря на это, с каждым годом данные модели применяются ко все большему числу задач, и ряд спе-

циалистов уверены, что область применения глубоких нейронных сетей будет расширяться с каждым годом [4].

Как было отмечено ранее, сети прямого распространения любой глубины не могут обрабатывать последовательности данных. Обойти данное ограничение можно с использованием рекуррентных нейронных сетей.

Особенностью данной архитектуры является то, что пути распространения сигнала в данных сетях образуют циклы, благодаря чему сеть обладает подобием памяти и может обрабатывать последовательности данных. На данный момент одной из наиболее распространенных разновидностей рекуррентных сетей является **LSTM** (Long Short-term Memory) – модель долгой кратковременной памяти, предложенная Сэппом Хокхрейтером и Юргеном Шмидхубером в 1997 году в статье «Long short-term memory» [5]. Сети, использующие данную архитектуру, показывают хорошие результаты в прогнозировании временных рядов и распознавании несегментированного рукописного текста. В то же время можно заметить, что **LSTM** специализирована для обработки последовательностей, тогда как данные для конкретной задачи могут иметь более сложную структуру (например, древовидную). Поэтому можно сказать, что данная архитектура также не является универсальной.

Исследование «On the computational power of neural nets» [6], проведенное в 1995 году Хэво Шигельманом и Эдуардо Сонтагом, показало, что рекуррентные нейронные сети полны по Тьюрингу, поэтому с их помощью можно выразить любой алгоритм, реализуемый на компьютере. Однако построение такого алгоритма при помощи методов машинного обучения может занимать много времени. Таким образом, возникает задача поиска архитектуры нейронной сети, которая будет поддерживать обработку данных произвольной структуры и обучаться произвольным алгоритмам за разумное время.

В последние годы исследованию данного вопроса было посвящено большое количество исследований, в том числе был предложен ряд подходов к индукции алгоритмов при помощи глубоких нейронных сетей. Идеи, на которых они основаны, достаточно разнообразны, рассмотрим преимущества и недостатки данных подходов.

Одной из моделей, решающих задачу вывода алгоритмов, является **Neural Turing Machine** (Нейронная машина Тьюринга, **NTM**), разработанная в 2014 году группой Google DeepMind [7]. Архитектура данной модели состоит из двух

компонент – блока работы с памятью и контроллера. В качестве последнего выступает сеть на основе **LSTM**, которая в процессе обучения записывает и считывает результаты из памяти. Отвечающий за это блок сохраняет и загружает данные из области памяти, а не из конкретной ячейки, чем достигается требуемая для градиентного спуска гладкость. Сеть, построенная на данном принципе, способна обучаться копировать и перемещать блоки памяти, и также производить сортировку небольших объемов данных.

Развитием данной идеи стал **Differentiable Neural Computer** (Дифференцируемый Нейронный Компьютер), разработанный той же группой в 2016 году [8]. В данном подходе работа с памятью была улучшена по сравнению с **NTM**, в частности была добавлена возможность поддерживать ссылки на недавно записанные значения, в результате чего модель стала способна обучаться алгоритмам на графах. В частности, было показано, что сеть может научиться находить кратчайшие пути в графах.

Альтернативным подходом к решению задачи индукции алгоритмов является **Neural GPU** (Нейронный GPU), рассмотренный командой Google Brain в 2015 году [9]. Данный подход основан на подходе, отличном от **NTM**, и позволяет преодолеть часть его недостатков, таких как медленные скорости обучения и обработки данных. Идея данного подхода заключается в сочетании операции свертки и **GRU** (Gated Recurrent Unit, альтернатива **LSTM**). Благодаря свертке алгоритм работы сети может быть обобщен на данные произвольной длины, а **GRU** позволяет сети эффективно обучаться. Одним из главных результатов данного подхода является то, что **Neural GPU** способен обучаться складывать и умножать длинные числа произвольной длины. Однако у данного подхода есть ограничения, рассмотренные в статье Эрика Прайса «Extensions and Limitations of the Neural GPU» [10]. Среди наиболее существенных можно отметить зависимость точности результата от порядка операндов и вероятность неверного ответа при нестандартном представлении данных (добавлении ведущих нулей к операндам). Поэтому данный подход также не лишен недостатков.

Обзор прочих подходов к индукции алгоритмов при помощи сетей глубокого обучения, проведенный Нилом Кэнтом в статье «Recent Advances in Neural Program Synthesis» [11], показывает, что всем существующим на данный момент подходам свойственен ряд недостатков, таких как:

— Необходимость большого объема данных и вычислительных мощностей для обучения.
— Сильное влияние архитектуры и решаемой задачи на время обучения. Под этим подразумевается, что при неверном выборе архитектуры время поиска алгоритма может быть достаточно велико.
— Отсутствие архитектуры, подходящей под произвольные виды задач.

Можно заметить, что данные ограничения достаточно существенны и ограничивают возможность вывода алгоритмов с помощью данных архитектур нейронных сетей.

Существует альтернативная точка зрения на процесс индукции алгоритмов, основанная на аналогии между нейронными сетями и программами. Впервые данное соответствие была отмечено в 1994 году в работе Хэво Шигельмана «Neural programming language» [12]. В данной статье был рассмотрен процедурный язык программирования **NEL**, программы на котором могут быть скомпилированы в нейронные сети, причем результаты работы программы и сети будет совпадать. Недостатком данного языка является то, что построенные сети не способны обучаться, что существо ограничивает возможности языка **NEL**.

Данный недостаток был решен в последующих работах, что привело к возникновению понятия **дифференцируемого языка программирования** – языка, программы на котором могут быть выведены либо изменены в процессе градиентного спуска. Языки данного класса могут использовать модели глубоких нейронных сетей, и за последние годы был сделан ряд исследований, подтверждающих жизнеспособность данного подхода.

Дифференцируемые языки программирования могут быть использованы для «точной настройки» программ на основе входных и выходных данных, как это было сделано в работе Руди Банела и Албана Десмейсона «Adaptive Neural Compilation» [13]. Идея данного исследования состоит в том, что на основании языка программирования, похожего на Assembler, строилась нейронная сеть, которая обучалась на примерах. В процессе обучения структура программы могла быть перестроена, чтобы лучше соответствовать распределениям входных и выходных данных.

Другой подход к использованию дифференцируемых языков программирования состоит в синтезе программ, решающих относительно сложные задачи, на основе более простых программ. Дифференцируемость языка позволяет

производить данную операцию с использованием градиентного спуска и нейронных сетей. В работе Скотта Фрида и Нандо де Фрейтаса «Neural Programmer-Ineterpreters» [14] данный подход был применен к выводу алгоритмов сортировки и модификации 3D-моделей на основе предопределенных примитивных алгоритмов. Полученные решения впоследствии так же могут быть использованы в качестве строительных блоков для новых алгоритмов.

Одной из последних разработок в области дифференцируемых языков программирования является язык **Forth**, представленный в статье Себастьяна Ридела «Programming with a Differentiable Forth Interpreter» [15]. Forth представляет собой императивный язык программирования, программы на котором также могут быть скомпилированы в нейронные сети. Ключевой особенностью языка является то, что код на нем может содержать пропуски, которые заполняются в процессе обучения. Таким образом, программист может написать на языке **Forth** «набросок» алгоритма, а итоговый вид решения будет найдены в процессе работы алгоритма и сопоставления результатов с образцовыми. В статье было показано, что данный подход может быть применен к решению задач сортировки массива и сложения чисел.

Таким образом, дифференцируемые языки программирования также могут быть использованы для индукции программ. Данный подход имеет ряд преимуществ по сравнению с рассмотренным выше методом вывода алгоритмов при помощи специфичных архитектур нейронных сетей. К примеру, при построении «наброска» алгоритма при помощи дифференцируемых языков программисту не требуется думать об архитектуре сети, что на порядок упрощает проектирование решений. Кроме того, если часть решения известна заранее, то мы можем описать ее в коде программы. После компиляции мы получим сеть, уже содержащую данную часть решения, и сети не придется обучаться ей по примерам. Это может сократить время обучения модели.

В данной работе для решения задачи индукции алгоритмов мною был выбран метод, схожий с тем, которым пользовались создатели языка **Forth**. Отличие моего подхода состоит в том, что дифференцируемый язык программирования использует функциональную парадигму. Ее главное отличие от императивного стиля состоит в использовании лямбда-исчисления и теории типов, что позволяет проверять корректность программ с использованием методов математической логики (см. книгу [16]). Обоснование преимуществ такого подхода

в приложении к проектированию нейронных сетей можно найти в статье Кристофера Олафа «Neural Networks, Types, and Functional Programming» [17]. В ней приводятся аналогии между архитектурами глубоких сетей и функциями высших порядков, используемыми в функциональном программировании. В своем исследовании я развил данную идею, что позволило проектировать архитектуры нейронных сетей, способные выводить алгоритмов.

1.3. Уточненные требования к работе

Как было отмечено выше, после проведения обзора имеющихся подходов и изучения их достоинств и недостатков, мною был выбран подход, использующий дифференцируемые языки программирования. В связи с этим, требования к результату работы были уточнены:

— требуется разработать дифференцируемый язык программирования, основанный на функциональной парадигме;
— программы, написанные на данном языке, должны быть компилируемы в нейронные сети глубокого обучения;
— требуется, чтобы программы могли содержать пропуски, которые могли бы быть заполнены в процессе обучения по примерам;
— язык должен быть Тьюринг-полным и поддерживать возможность обработки данных произвольной структуры.

1.4. Термины и понятия

В данной работе используются понятия из областей машинного обучения, лямбда-исчисления и теории типов. Далее будет приведены определения наиболее важных понятий.

1.4.1. Машинное обучение

Машинное обучение – класс методов искусственного интеллекта, характерной чертой которых является решение задачи в процессе обучения по примерам.

Искусственная нейронная сеть – математическая модель, а также ее программное или аппаратное воплощение, построенные по принципу организации и функционирования сетей нервных клеток живого организма.

Искусственный нейрон – составная часть (**узел**) искусственной нейронной сети, являющийся упрощенной моделью естественного нейрона. Математически, искусственный нейрон представляет собой нелинейную функцию от линейной комбинации всех входных сигналов. Данную функцию называют **функцией активации**. Коэффициенты в линейной комбинации входных сигналов группируются в **вектор весов**. Значение результирующего сигнала нейрона вычисляется по формуле:

$$out = \sigma(weights \cdot in + bias) \tag{1}$$

где σ – функция активации, $weights$ – вектор весов, $bias$ – смещение.

Сигмоидная функция активации – гладкая нелинейная функция, монотонно возрастающая и имеющая форму буквы «S». Часто под данным термином понимают функцию:

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

Softmax (функция активации) – обобщение логистической функции для многомерного случая. Функция преобразует вектор z размерности K в вектор σ той же размерности, где каждая координата содержится в интервале $[0,1]$ и сумма координат равна 1. i-я координата результирующего вектора вычисляется по формуле:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

Сеть прямого распространения (Feed-forward network) – архитектура нейронных сетей. Сети данного класса представляют собой набор **слоев**, каждый из которых состоит из конечного числа искусственных нейронов. Все связи между слоями направлены строго от входных нейронов к выходным. Векторы весов, соответствующих нейронам каждого слова, группируются в **матрицы весов**. В процессе обучения находятся значения компонент данных матриц.

Рекуррентная нейронная сеть – вид нейронных сетей, в которых сигнал между слоями может распространятся от выходных нейронов ко входным.

Рекурсивная нейронная сеть – вид нейронных сетей, которые способны обрабатывать иерархические данные. Сети данного класса применяют один и тот же набор слоев к каждому уровню структуры данных.

Разделение весов (Weight Sharing) – подход к проектированию нейронных сетей, в котором одни и те же матрицы весов используются в нескольких слоях сети. Благодаря этому становится возможным сократить число параметров модели.

Градиентный спуск – метод нахождения локального экстремума функции при помощи движения вдоль ее градиента.

Метод обратного распространения ошибки – метод вычисления градиента, используемый при обновлении компонент матриц весов многослойной нейронной сети.

Функция потерь – функция, которая характеризует потери при неправильном принятии решений на основе наблюдаемых данных. В качестве функции потерь часто используется сумма квадратов расстояний от выходных сигналов сети до их требуемых значений – так называемая **квадратичная функция потерь**:

$$MSE(got, expected) = \sum_i (expected_i - got_i)^2$$

1.4.2. Лямбда-исчисление и теория типов

Тип данных – класс данных, характеризуемый членами класса и операциями, которые могут быть к ним применены. Далее запись $a : A$ будет означать, что объект a имеет тип A, то есть значение a принадлежит множеству, задаваемому типом A.

Алгебраический Тип Данных (АТД) – тип данных, представляющий собой тип-сумму, где слагаемые – типы-литералы или типы-произведения, чьи множители – АТД или типовые переменные. Типы данного семейства представимы в грамматике:

$$\begin{aligned} T &= TypeSumOp \ + \ TypeSumOp \\ TypeSumOp &= \Lambda \mid TypeProdOp \ \times \ TypeProdOp \\ TypeProdOp &= \tau \mid T \end{aligned}$$

За данным определением стоит следующая семантика:

— $a : \Lambda$ – объект соответствует типовому литералу Λ. Будем считать, что все объекты, соответствующие одному и тому же литералу, равны между собой, и будем обозначать их так же, как и сам литерал, например $True : True$

— $a : \tau$ – объект a имеет переменный тип. Точный тип объекта будет определен после замены переменной τ на соответствующий тип

— $a : A + B$ – тип объекта – сумма A и B. Это означает, что a может иметь либо тип A, либо тип B. Например $a : True + False$ означает, что a либо $True$, либо $False$

— $a : A \times B$ – тип объекта a – произведение A и B. Это означает, что объект a представляет собой пару объектов типов A и B. К примеру, объект $(True, False)$ иммет тип $Bool \times Bool$, где $Bool = True + False$

Определение типа можно тривиально расширить на конечные суммы и произведения типов. В дальнейших разделах мы будем рассматривать именно такие типы.

Рекурсивный АТД – обобщение алгебраического типа данных, позволяющее описывать данные, обладающие бесконечной структурой (к примеру, списки). Задается грамматикой:

$$R = \mu\ \rho.\ T \tag{2}$$

Где μ – комбинатор неподвижной точки для типа, ρ – типовая переменная, T может использовать в своей структуре переменную ρ. Для типов данного семейства определена операция «разворачивания», в процессе которой переменная ρ в структуре T заменяется на T. Данную операцию можно повторять сколько угодно раз, получая типы все большего размера. Благодаря этому возможно задавать типы бесконечного размера, используя рекурсивные АТД.

Полиморфные АТД – обобщение АТД, позволяющее типам содержать в своей структуре типовые параметры. Полиморфный АТД представляет собой **конструктор типа**, так как он позволяет создать новый тип, подставив типы на место параметров.

Сорт типа – «тип» конструктора типов. Сорт АТД, не содержащих типовых параметров, род равен *. Сорт полиморфных АТД представляет собой

выражение в грамматике:

$$Kind = * \mid Kind \ \rightarrow \ Kind$$

Лямбда-выражение – выражение, которое задается грамматикой:

$$E = x \mid \lambda \, x \, . \, E \mid E \; E \mid (E) \tag{3}$$

В данных обозначениях $(\lambda \; x \; . \; E)$ называется **лямбда-абстракцией**, $(E \; E)$ называется **аппликацией**.

Для лямбда-выражений определено понятие **редукции**, в процессе которой выражения вида $(\lambda \; x.E_1) \; E_2$ заменяются на $E_1[x \; = \; E_2]$ (все свободные вхождения x в E_1 заменяются на E_2). Благодаря данной операции язык лямбда-выражений становится **Тьюринг-полным**, то есть на нем становится возможным реализовать любую вычислимую функцию.

Просто типизированное лямбда-исчисление – лямбда-исчисление, где выражениям соответствуют специальные метки, называемые типами. Типы в данном исчислении задаются грамматикой:

$$T \; = \; a \mid T \rightarrow T \mid (T)$$

Где a – типовая переменная (либо АТД) и $T \rightarrow T$ соответствует функции.

Параметрический полиморфизм – свойство семантики системы типов, позволяющее обрабатывать значения разных типов идентичным образом, то есть исполнять один и тот же код для данных разных типов.

Система типов Хиндли-Милнера (ХМ) – система типов для лямбда-исчисления с параметрическим полиморфизмом. Для данной системы разработан **Алгоритм W**, позволяющий за полиномиальное время найти наиболее общий тип для лямбда-выражения.

Типы системы ХМ имеют вид $\forall a_1 \; a_2 \; ... \; a_n. \; T$, где a_n – типовые переменные, T – тип из просто типизированного лямбда-исчисления, который может содержать в своем составе типовые переменные a_i.

Использование данной типовой системы позволяет расширить грамматику для лямбда-выражений, добавив в нее **let-абстракцию**:

$$let\ x = E_1\ in\ E_2$$

В процессе редукции выражения вида $let\ x\ =\ E_1\ in\ E_2$ заменяются на $(\lambda\ x.\ E_2)\ E_1$, которые редуцируются согласно правилам редукции лямбда-выражений. Выражения, задаваемые let-абстракцией, имеют наиболее общий тип из возможных.

Функции высшего порядка – лямбда-выражение, имеющее тип функции, хотя бы один из аргументов которой также имеет тип функции ($T \to T$).

Комбинатор неподвижной точки (Y-комбинатор) – функция высшего порядка, вычисляющая неподвижную точку другой функции. Позволяет задавать рекурсивные программы при помощи лямбда-исчисления. Не типизируется ни в просто типизированном лямбда-исчислении, ни в системе типов ХМ.

Сопоставление с образцом – метод анализа и обработки структур данных, основанный на выполнении определенных инструкций в зависимости от совпадения исследуемого значения с тем или иным образцом. Образец задается в грамматике:

$$P = x \mid C \mid (C\ P_1\ P_2\ ...\ P_n) \tag{4}$$

Где x – переменная, C – конструктор, P_i – образцы-аргументы конструктора.

Абстрактное Синтаксическое Дерево (АСТ) – конечное ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья – с переменными и константами.

Выводы по главе 1

После проведения исследования литературы было установлено, что подходы к решению поставленной задачи можно разделить на две группы - разработку универсальной архитектуры нейронной сети, которая способна вывести любой алгоритм, и разработку архитектур под конкретные задачи. Анализ данных подходов показал, что первый обладает недостатками, из-за которых его проблематично использовать. В частности, для обучения сетей данного класса требуется большое количество времени и вычислительных ресурсов.

Для решения поставленной задачи был выбран второй подход, использующий дифференцируемые языки программирования. «Наброски» программ на данных языках задают структуру нейронной сети, в процессе обучения которой происходит индукция алгоритма, решающего задачу.

В данной работе исследуется функциональный дифференцируемый язык программирования. Данный подход обладает рядом преимуществ, которые упрощают процесс проектирования сети и позволяют обрабатывать данные произвольной структуры.

ГЛАВА 2. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ МЕТОДА

В данном исследовании для решения задачи вывода алгоритмов при помощи нейронных сетей применяется подход, основанный на использовании функционального дифференцируемого языка программирования. Причина такого выбора заключается в том, что между нейронными сетями и функциональными программами можно проследить ряд параллелей, которые были отмечены в статье Кристофера Олафа «Neural Networks, Types, and Functional Programming» [17]. Это исследование обозначило соответствие между архитектурами рекуррентных нейронных сетей и функциями высшего порядка, используемых в функциональных языках программирования. Детали соответствия приведены в Таблице 1.

Таблица 1 – Соответствие архитектур сетей и функций высшего порядка

Архитектура сети	Функция высшего порядка
Рекуррентный энкодер	*fold*
Рекурентный декодер	*unfold*
Рекуррентные нейронные сети	*mapAccum*
Рекурсивные нейронные сети	*cata*

Результатом исследования, приведенного в данной ВКР, является расширение данного соответствия, благодаря чему становится возможным строить нейронные сети на основе программ, написанных в функциональной парадигме. Кроме того, проводя обратную аналогию, становится возможным выводить алгоритмы обработки данных в процессе обучения нейронных сетей. Результаты исследования данного соответствия можно привести в Таблице 2, детали которой будут разъяснены в дальнейших пунктах данной главы:

Таблица 2 – Соответствие нейронных сетей и лямбда-выражений

Нейронные сети	Лямбда-выражения
Структура слоя	Алгебраический тип данных
Функция активации	Ограничения типа данных
Структура нейронной сети	Структура лямбда-выражения
Сети прямого распространения	Выражения без рекурсии
Рекурсивные нейронные сети	Выражения с Y-комбинатором
Полиморфные нейронные сети	Полиморфные выражения

2.1. Анализ существующих методов индукции алгоритмов

В обзоре имеющихся подходов (пункт 1.2) был перечислен ряд архитектур нейронных сетей, способных индуцировать алгоритмы. Проведем анализ нескольких архитектур и подходов, отметим их достоинства и недостатки.

Differentiable Neural Computer (DNC) – архитектура нейронных сетей, описываемая в одноименной работе [8]. Данная сеть является развитием идей **Neural Turing Machine**, она включает в себя блок чтения и записи в память, обучаемый контроллер памяти и поддерживает во время работы набор ссылок на данные, к которым был произведен доступ некоторое время назад. В процессе обучения сеть может как читать данные из произвольного места памяти, так и обратиться к недавним данным по ссылке. Подобная ссылочная структура позволила расширить возможности предшественника. Так, к примеру, **DNC** способен обучаться алгоритму нахождения кратчайшего пути в графах.

Среди преимуществ данной архитектуры можно отметить то, что она универсальна – **DNC** сможет произвести вывод произвольного алгоритма, но потребует на это большое количество времени и ресурсов. Это же является и ее недостатком, потому что сеть обучается медленно (см. статью [11]).

В данной ВКР используется идея ссылок на данные, поскольку это позволяет расширить набор данных, используемых сетью в процессе обучения. Отличие от **DNC** состоит в том, что определяемые ссылки поддерживаются во всё время работы алгоритма, а не стираются по прошествии определенного количества шагов. Данная идея используется для представления структурированных данных (см. пункт 2.2.4)

Neural GPU – альтернативная универсальная архитектура, способная выводить алгоритмы, определенная в одноименной статье [9]. Идея данной архитектуры состоит в использовании сверток для обработки данных произвольной длины. Преимущество **Neural GPU** состоит в высокой параллельности производимых операций, что ускоряет обучение.

Данная ВКР развивает идею использования свертки как способа обработки данных произвольной длины. Одним из результатов работы является то, что используя лямбда-выражения и алгебраические типы данных, можно обобщить понятие свертки с обработки векторов и матриц на обработку произвольных структурированных данных.

Differentiable Forth – дифференцируемый язык программирования Forth, определяемый в статье [15]. Программы на данном языке представляют собой наброски алгоритмов, на основе которых строятся нейронные сети, параметры которых находятся в процессе обучения. При помощи данного подхода возможно обучаться алгоритмам сложения чисел произвольной длины и сравнению элементов массива.

Идея данного языка легла в основу подхода, выбранного в данной ВКР. Язык **Forth** является довольно низкоуровневым, поэтому он малоприменим к использованию в реальном мире. В данной работе предлагается функциональный дифференцируемый язык программирования **FNN**, который, как и **Forth**, позволяет добавлять в программы пропуски, но в то же время использует результаты лямбда-исчисления и теории типов. Благодаря этому язык **FNN** позволяет в достаточно удобной форме проектировать нейронные сети, способные обучаться алгоритмам.

2.2. Обработка структурированных данных

Проведенное в данной работе исследование показало, что с использованием алгебраических типов данных (АТД) можно проектировать нейронные сети, которые способны обрабатывать данные со сложной структурой. В данном пункте будет дано подробное описание разработанного подхода к представлению и обработке структурированных данных.

2.2.1. Векторное представление данных

В данной работе приводится способ представления структурированных данных в форме, пригодной для обработки при помощи нейронных сетей. Благодаря этому становится возможным использовать их для вывода алгоритмов. В работе предлагается сопоставлять объектам векторы (либо деревья, содержащие векторы), компоненты которых заключены в интервале $[0, 1]$ и отвечают за присутствие в составе объекта соответствующих подобъектов.

Структурированные данные удобно описывать при помощи алгебраических типов данных. Напомню, что данные типы представляют собой конечную сумму, где каждый операнд – тип-литерал или тип-произведение, где все множители – типовые переменные или АТД.

Покажем, как объекты типов-сумм, где все аргументы – типовые литералы, можно представить в виде векторов. В качестве примера подобного типа

можно рассмотреть тип $Bool$, равный сумме $True + False$. Мы можем упорядочить все литералы в структуре таких типов в порядке их упоминания в определении (так $True$ становится первым, $False$ – вторым).

Если в определении типа встречается N литералов, то мы можем сопоставить каждому объекту данного типа вектор длины N. Рассмотрим объект, соответствующему j-му литералу в определении АТД, и поставим ему в соответствие вектор v, где

$$v[i] = \begin{cases} 1, & \text{если } i = j \\ 0, & \text{если } i \neq j \end{cases}$$

Таким образом объекту $a = True$ будет соответствовать вектор $(1\ 0)$, а объекту $b = False$ – $(0\ 1)$

Использование АТД, содержащих в своей структуре типы-произведения, позволяет определять иерархические типы данных. Согласно семантике типа-произведения, объекты подобных типов содержат в своей структуре объекты типов-множителей. В качестве примера можно привести пару из двух $Bool$: $Pair = Bool \times Bool$. Объекты такого типа должны будут содержать в себе два объекта типа $Bool$.

В данной работе разработаны два способа представления структурированных данных, которые будут рассмотрены в следующих пунктах:

— Представление в виде конечного вектора (пункт 2.2.2)
— Представление с сохранением структуры (пункт 2.2.4)

2.2.2. Представление данных конечной структуры

Определение 1. **АТД с конечной структурой** – алгебраический тип данных, представляющий собой тип-сумму, для которой верно одно из двух утверждений:

— Все слагаемые – типы-литералы
— Все типы-множители в слагаемых-произведениях – АТД с конечной структурой

Рассмотрим АТД с конечной структурой. Если мы заменим все АТД во множителях слагаемых-произведений на их определения (типы-суммы), то получим конечное дерево, где узлы – операции суммы или произведения, а в листьях расположены типовые литералы. Упорядочим все литералы в порядке об-

хода дерева и применим рассуждения из предыдущего пункта. Объектам типа, содержащего N литералов, будем ставить в соответствие векторы, содержащие N значений в диапазоне $[0, 1]$. Значения компонент вектора будут равны 1, если соответствующие компонентам литералы содержатся в структуре объекта, и 0 иначе.

В качестве примера рассмотрим следующие типы:

$$Bool = True + False$$
$$Answer = Yes + Maybe + No$$
$$Pair = Bool \times Answer$$

И приведем примеры векторов, соответствующих объектам типа $Pair$:

— $(True, Maybe) \leftrightarrow (1\ 0\ 0\ 1\ 0)$

— $(False, No) \leftrightarrow (0\ 1\ 0\ 0\ 1)$

Заметим, что чтобы представить объект в виде вектора, мы должны заранее знать число литералов в структуре типа. Из-за этого становится невозможно описывать данные бесконечной структуры (например, списки, деревья, натуральные числа) при помощи данного представления. Обобщение векторного подхода, позволяющее преодолеть данную трудность, рассматривается в пункте 2.2.4.

С использованием приведенного подхода можно разрабатывать нейронные сети, обрабатывающие данные конечного размера. При подобном подходе каждый полносвязный слой будет строиться по некоторому АТД, который будет задавать задавать размеры данных, возвращаемых слоем. Размеры матриц весов будут задаваться размерами соединенных слоев.

2.2.3. Связь функций активации с ограничениями типов

В предыдущих пунктах было показано, что объекты конечных АТД можно представить в виде векторов. Определение АТД накладывает ограничения на значения их компонент. Например, в векторах соответствующих типу $Bool$, только одна компонента будет равна 1, так как объект типа $Bool$ не может быть одновременно и $True$, и $False$. Типы-произведения также накладывают ограничения на компоненты, согласно которым в структуре объекта типа-произведения должен присутствовать объект каждого типа-множителя.

Ранее было показано, что АТД можно использовать для проектирования слоев нейронных сетей. Известно, что значения функций активации на каждом слое сети вычисляются по Формуле 1. Аргумент функции активации может быть произвольным, но значения компонент результата уже обязаны соответствовать ограничениям АТД.

Поэтому данные ограничения необходимо выразить в терминах функций активации. В частности, ограничениям типа-суммы типов-литералов соответствует функция активации Softmax, которая гарантирует, что сумма компонент результата будет равна 1. Ограничениями типа-произведения соответствует сигмоидная функция активации, которая обеспечивает, что значения компонент будут лежать в диапазоне $[0, 1]$.

Через Softmax и сигмоиду можно выразить не все возможные ограничения. Рассмотрим следующий тип и приведем пример его объектов:

$$
\begin{aligned}
First &= First \\
Second &= Second \\
Type &= First \ \times \ Second \ + \ Third \\
a &= (First, Second) && \leftrightarrow \ [1, 1, 0] \\
b &= Third && \leftrightarrow \ [0, 0, 1]
\end{aligned}
$$

На векторы, соответствующие объектам данного типа, накладываются более сложные ограничения. В частности, либо первые две компоненты должны быть одновременно равны 1, либо третья. Для осуществления данных ограничений требуется функция активации, основана на определении типа.

Определение 2. **Типизированная функция активации** – семейство функций активации, элементы которого могут быть построены на основе АТД.

В данном пункте мы будем рассматривать функции активации, построенные по АТД следующего вида:

$$Type = \sum_{i}^{k} C_i \ + \ \sum_{i}^{n} \prod_{j}^{m_i} T_{ij} \tag{5}$$

Где C_i – типовые литералы, T_{ij} – АТД, являющиеся типом-суммой одного литерала. Ограничения типа $Type$ гласят, что объект подобного типа может быть

либо одним из литералов C_i, либо одним из произведений, в котором должны присутствовать объекты всех типов-множителей.

Объектам подобного типа будут соответствовать вектора с компонентами в интервале $[0,\ 1]$. Для того, чтобы выразить ограничения, накладываемые на такие векторы, определим понятие соответствия:

Определение 3. **Соответствие** – величина в интервале $[0,\ 1]$, которая показывает, насколько точно вектор соответствует типовым ограничениям. Вычисляется по формуле:

$$Corr(type, v) = \begin{cases} v[i], & \text{если } type \text{ соответствует i-му литералу} \\ \sum\limits_{i}^{n} Corr(t_i, v), & \text{если } type = \sum\limits_{i}^{n} t_i \\ \prod\limits_{i}^{n} Corr(t_i, v), & \text{если } type = \prod\limits_{i}^{n} t_i \end{cases}$$

Используя данное понятие, мы можем выразить ограничения, накладываемые типом $Type$ на вектор, как

$$Corr(Type, v) = 1 \tag{6}$$

Данному ограничению удовлетворяют как вектора, построенные по объектам, так и более широкий класс векторов, которые объектам не соответствуют. К примеру, вектор $[0{,}5,\ 0{,}5]$ удовлетворяет ограничениям типа $Bool$, но не является ни $True$, ни $False$.

Определение 4. **Смешанный объект** – гипотетический объект, которому соответствуют вектор, удовлетворяющий ограничению Формулы 6.

Приведем пример типизированной функции активации, которая позволит получать экземпляры подобных смешанных объектов для типов, задаваемых Формулой 5. В векторах, построенных по подобным типам, i-я компонента соответствует некоторому типу, будем обозначать его $T(i)$.

Определение 5. **Множества позиций множителей** типа $Type$ – набор множеств, i-е из которых содержит позиции множителей в i-м слагаемом в опреде-

лении $Type$. Задается как

$$Prod(Type)[i] = \begin{cases} \{k\}, & \text{если i-е слагаемое в определении } Type \text{ – литерал } T(k) \\ \{k_1,k_2,...,k_n\}, & \text{если i-е слагаемое в определении } Type \text{ – произведение типов } T(k_j) \end{cases}$$

Определение 6. **Типизированная сигмоидная функция активации** – типизированная функция активации, действующая на векторы. Для позиции вектора, соответствующей типу, входящему в k-е слагаемое в определении $Type$, значение задается формулой

$$\tau(Type, v[i]) = \sum_{j \in Prod(Type)[k]} SM(v[j]) \cdot \sigma(v[i]) \tag{7}$$

где

$$SM(v[i]) = \frac{e^{v[i]}}{\sum_j e^{v[j]}}$$

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

Для компонент, отвечающим типам-литералам, данная функция активации представляет собой произведение Softmax и сигмоидной функции (так как сумма содержит только 1 член). Для компонент, отвечающих множителям в типах-произведениях, значение функции получается в результате умножения Softmax для всего произведения целиком и сигмоиды для конкретной компоненты. Если значения компонент $v[i]$ достаточно велики, то для результата применения данной функции активации будет верно Выражение 6, что обеспечит выполнение ограничений типов.

Частные производные для данной функции активации вычисляются по формуле:

$$\frac{\partial \tau(v[i])}{\partial v[j]} = \tau(v[i]) \cdot (\delta^i_j - \sigma(v[i]) - SM(v[j])) + \Delta^i_j \cdot SM(v[j]) \cdot \sigma(v[j])$$

$$\Delta_j^i = \begin{cases} 1, & \text{если i и j – множители одного произведения} \\ 0, & \text{иначе} \end{cases}$$

При помощи типизированной сигмоидной функции активации можно выражать ограничения АТД. Это позволяет нам проектировать нейронные сети, которые смогут принимать и возвращать смешанные объекты. Размеры полносвязных слоев в составе подобной сети будут определяться числом литералов в АТД, в качестве функции активации будет использована приведенная выше типизированная сигмоида. Это позволит нейронным сетям работать со смешанными объектами, что приближает нас к выводу алгоритмов при помощи нейронных сетей.

2.2.4. Представление данных произвольной структуры

Ранее был приведен способ векторного представления объектов конечных алгебраических типов данных. Обобщим данный подход для того, чтобы его можно было применять к рекурсивным АТД, что позволит нам в конечном итоге обрабатывать объекты произвольной структуры при помощи нейронных сетей.

Основная сложность представления данных произвольной структуры заключается в том, что на момент компиляции нам неизвестна точная структура входных данных. Из-за этого мы не можем представить объекты в виде векторов фиксированной длины.

В данной работе предлагается использовать иерархическое представление данных. В подобном представление объектам соответствуют деревья, где каждый узел содержит вектор значений, описывающий структуру части объекта, и также набор ссылок на другие узлы.

Определим ряд понятий, требуемых для описания данного представления. Рассмотрим АТД T, представляющий собой тип-сумму.

Определение 7. **Состав типа** T – список, состоящий из всех слагаемых-литералов и множителей слагаемых-произведений в типе T, упорядоченный в порядке их вхождения в определение.

Определение 8. **Длина определения типа** T – длина состава типа T. Если среди слагаемых в типе T встречаются N литералов, а все слагаемые-произведения суммарно содержат M множителей, то длина определения типа T равна $L = N + M$.

К примеру, рассмотрим тип $MaybeBoolPair = (Bool \times Bool) + Nothing$. Состав данного типа – $[Bool, Bool, Nothing]$, длина определения – 3.

Определение 9. Будем называть **Т-Деревом** (*Тензорным деревом*) структуру данных, представляющую собой дерево, каждый узел которого содержит в себе тензор $data$ (вектор или матрицу) и список $children$, содержащий ссылки на другие узлы Т-дерева. Каждый узел дерева будем называть **слоем**.

В данной работе мы будем использовать Т-деревья для описания объектов, которыми оперируют сети. Объекту $a : T$ будет соответствовать Т-дерево, содержащее в узле вектор $data$, описывающий объект a, а детьми будут деревья, описывающие объекты в составе a. Длины $data$ и $children$ будут равна длине определения типа T, компонента $data[i]$ будет описывать, присутствует ли в структуре объекта a объект, задаваемый $children[i]$. Поясним данную конструкцию на примере:

$$
\begin{aligned}
Bool &= True \ + \ False \\
MaybeBoolPair &= (Bool \ \times \ Bool) \ + \ Nothing \\
True &\leftrightarrow TensorTree([1, 0], [null, null]) \\
False &\leftrightarrow TensorTree([0, 1], [null, null]) \\
(True, False) &\leftrightarrow TensorTree([1, 1, 0], [\\
&\qquad TensorTree([1, 0], [null, null]), \\
&\qquad TensorTree([0, 1], [null, null]), \\
&\qquad null \\
&\quad]) \\
Nothing &\leftrightarrow TensorTree([0, 0, 1], [null, null, null])
\end{aligned}
$$

Здесь $TensorTree(data, children)$ задает Т-дерево, $null$ соответствует отсутствию ребенка. Длина векторов $data$ и $children$ равна длине определения типа и известна заранее. Если объект представляет собой i-й литерал в составе типа (случаи $True$, $False$, $Nothing$), то $data[i] = 1$, остальные компоненты равны 0, $children[j] = null$. Если же объект является составным (случай пары $(True, False)$), и содержит объект i-го по порядку типа в составе исходного, то $children[i]$ содержит описывающее его Т-дерево и $data[i] = 1$, остальные компоненты равны $null$ и 0 соответственно.

Таким образом, $children$ представляет собой массив ссылок на слои, описывающие объекты, входящие в структуру данного, а $data$ содержит величины присутствия литералов и объектов в структуре данного объекта.

Алгоритм построения Т-дерева для произвольного объекта приводится в Листинге 1.

Листинг 1 – Построение Т-дерева для объекта

```
function Represent(object, type)
    L ← длина определения типа type
    vector ← вектор длины L, все значения 0
    children ← вектор длины L, все значения пустые (null)
    if object – литерал then
        i ← позиция литерала object в определении типа type
        // Заносим в i-ю компоненту 1, так как литерал присутствует в структуре
        vector[i] ← 1
    else
        // object имеет тип произведения, состоит из нескольких объектов
        for all child – объект в составе object do
            i ← позиция child в определении типа type
            childType ← тип child
            layer ← Represent(child, childType)
            // Заносим в i-ю компоненту величину соответствия объекта child его типу
            vector[i] ← Corr(layer.data, childType)
            // Сохраняем ссылку на новый слой
            children[i] ← layer
        end for
    end if
    return TensorTree(vector, children)
end function
```

Длина векторов в каждом слое Т-дерева фиксирована и известна заранее, однако число слоев в структуре объекта может быть произвольным. Это позволяет описывать бесконечные типы данных (списки, деревья) при помощи Т-деревьев.

Таким образом предложенный подход обобщает представление конечных данных, благодаря чему становится возможным обрабатывать данные произвольной структуры при помощи нейронных сетей.

2.2.5. Операции над структурированными представлениями

Подобный способ организации данных позволяет нам оперировать данными произвольного размера, однако в то же время усложняется процесс обучения сетей. Данные организованы не в векторы, а в Т-деревья, для которых не определены стандартные операции умножения на матрицу, сложения и т. д. Для того, чтобы работать с данными, организованными в древовидные структуры, определим ряд операций.

Определение 10. Пусть – Т-дерево, $\alpha \in \mathbb{R}$. Тогда $S = \alpha \cdot T$ – Т-дерево, где

$$S.data = \alpha \cdot T.data$$

$$S.children[i] = \begin{cases} null, & \text{если } T.children[i] = null \\ \alpha \cdot T.children[i], & \text{если } T.children[i] \neq null \end{cases}$$

Определение 11. Пусть , S – Т-деревья, имеющие одинаковые размеры тензоров. Тогда $R = T + S$ – Т-дерево, где

$$R.data = T.data + S.data$$

$$R.children[i] = \begin{cases} null, & \text{при } T.children[i]{=}null \\ & \text{и } S.children[i]{=}null \\ T.children[i], & \text{при } T.children[i]{\neq}null \\ & \text{и } S.children[i]{=}null \\ S.children[i], & \text{при } T.children[i]{=}null \\ & \text{и } S.children[i]{\neq}null \\ T.children[i]+ & \text{при } T.children[i]{\neq}null \\ S.children[i] & \text{и } S.children[i]{\neq}null \end{cases},$$

Используя операции умножения на константу и сложения Т-деревьев, определим операцию умножения Т-дерева на матрицу:

Определение 12. Пусть T – Т-дерево, A – матрица такого размера, что произведение $T.data \cdot A$ определено. Тогда $S = T \cdot A$ – Т-дерево, где

$$S.data = T.data \cdot A$$

$$S.children[j] = \sum_{\substack{i \\ T.children[i] \neq null}} A[i,j] \cdot T.children[i]$$

За данной операцией стоит следующая интуиция. Матрица описывает способ получения объекта другого типа на основе исходного, в нашем случае результатом умножения является Т-дерево, где тензор равен матричному умножение тензора объекта и матрицы. В Т-деревьях значения тензора зависят от соответствия деревьев-детей их типам, поэтому мы умножаем детей на соответствующие числа матрицы и складываем. В результате, благодаря линейности всех операций, значения тензоров и присутствий детей оказываются согласованными.

Определим операцию умножения Т-деревьев, которая будет иметь смысл, схожий с умножением вектора на матрицу. Второе дерево содержит матрицы и описывает то, как на основе первого дерева построить другое. Структура второго дерева определяет, сколько слоев результата мы получаем на основе первого дерева. В процессе вычисления первое дерево умножается на матрицы, содержащиеся в слоях второго дерева, результирующее дерево составляется из результатов умножений.

Определение 13. Пусть T, S – Т-деревья, причем произведения $T \cdot S.data$ и $T \cdot S.children[i]$ определены. Тогда $R = T \cdot S$ – Т-дерево, где

$$R = T \cdot S.data + Q$$

$$Q.data = \begin{cases} 0, & \text{если } Q.children[i] = null \\ Corr(Q.children[i], childType), & \text{если } Q.children[i] \neq null \end{cases}$$

$$Q.children[i] = \begin{cases} null, & \text{если } S.children[i] = null \\ T \cdot S.children[i], & \text{если } S.children[i] \neq null \end{cases}$$

Как известно, в каждом узле нейронной сети вычисляется значение линейной комбинации входных значений. Если мы организуем результаты всех нейронов в вектор, то вычисление его значений будет производиться по формуле $out = in \cdot W + b$, где in – входные значения, W – матрица весов, b – вектор смещений. Для того, чтобы обобщить данную процедуру до Т-деревьев, введем ряд определений:

Определение 14. **О-дерево (Операторное дерево)** – дерево, в каждом узле которого содержится Т-дерево $tree$, содержащее матрицы, и список детей $children$. Каждое дерево $tree$ в составе О-дерева имеет одинаковую структуру – размеры тензора и количество детей.

O-дерево описывает, как на основе несколько слоев исходного дерево получить другое T-дерево, то есть обобщает понятие оператора до структурированных данных.

Определение 15. Пусть O – O-дерево, T – T-дерево, такие, что все приведенные ниже операции определены. Тогда $S = O \cdot T$ – T-дерево, где

$$S = T \cdot O.tree + \sum_{\substack{i \\ T.children[i] \neq null \\ O.children[i] \neq null}} T.data[i] \cdot (T.children[i] \cdot O.children[i])$$

Согласно определению данной операции, мы сопоставляем уровням O-дерева уровни T-дерева, затем перемножаем деревья, лежащие на данных уровнях, и складываем результаты. В результате перемножения деревьев одного уровня мы получаем объект результирующего типа. Из-за того, что результат умножения является суммой этих объектов, мы получаем, что результат всей операции учитывает все возможные способы получить дерево результирующего типа из уровней исходного T-дерева.

Можно заметить, что вектор соответствует однослойному T-дереву, а матрица – однослойному O-дереву, содержащему однослойное T-дерево. Таким образом, T-дерево обобщает понятие вектора для структурированных данных, а O-дерево обобщает понятие матрицы.

2.2.6. Структурированный полносвязный слой

Определение 16. **Структурированный полносвязный слой** – слой нейронной сети, параметры которого организованы в O-дерево $weights$, отвечающее за веса, и T-дерево $bias$, отвечающее за смещение. Дерево $bias$ может отсутствовать. Входные данные слоя (in) и выходные (out) представляют собой T-деревья, их значения вычисляются по формуле:

$$linear = \begin{cases} wieghts \cdot in, & \text{если } bias \text{ отсутствует} \\ weights \cdot in + bias, & \text{если } bias \text{ задан} \end{cases}$$

$$out = \tau(linear)$$

Данное понятие обобщает полносвязный слой, позволяя обрабатывать структурированные данные произвольного размера. Входные значения и ре-

зультаты данного слоя представляют собой Т-деревья, соответствующие смешанным объектам некоторого АТД.

Можно заметить, что векторы в составе $linear$ могут содержать значения, не удовлетворяющие ограничениям АТД, поэтому мы применяем к ним типизированную функцию активации. Типизированная сигмоида применяется к каждому слою дерева с передачей типа слоя, что обеспечивает соответствие результата ограничениям АТД.

Благодаря дифференцируемости функции активации и операций с деревьями, параметры слоев возможно находить в процессе градиентного спуска. Поэтому используя подобные слои можно проектировать сети, способные обучаться алгоритмам работы с структурированными данными.

2.3. Построение сети на основе лямбда-выражений

Проведенное исследование показало, что при помощи лямбда-выражений можно задавать архитектуру нейронной сети. Это возможно делать благодаря типовой информации, которая определяет структуру слоев и связи между ними. В данном разделе будет приведен алгоритм построения сети по лямбда-выражению.

Мы будем рассматривать выражения, которые можно типизировать в системе типов Хиндли-Милнера. Данное требование гарантирует, что аргументы функций будут иметь подходящие типы, но в то же время накладывает ограничение, не позволяющее определять рекурсивные программы. Вопрос о рекурсии будет рассмотрен в пункте 2.5.

В данном пункте мы будем рассматривать лямбда-выражения без let-абстракций. Вопрос о полиморфизме будет рассмотрен в пункте 2.6.1. Согласно определению, лямбда-выражения могут быть переменными, лямбда-абстракциями и аппликациями. Лямбда-абстракции вводят в область видимости новые переменные, переменные возвращают ранее определенные значения, аппликация «применяет» первое выражение ко второму. Переменные могут быть двух типов – *переменные-данные* и *переменные-сети*. Данное разделение происходит на основании типа переменной, к первой группе относятся переменные, чей тип данных – алгебраический, ко второй относятся объекты с типом $T \to T$.

2.3.1. Расширенные нейронные сети

В данной работе предлагается несколько расширить понятие нейронной сети, что позволит обучаться алгоритмам обработки структурированных данных.

Определение 17. **Расширенная нейронная сеть** – обобщение нейронной сети, слои которой могут принимать и возвращать как данные, так и нейронные сети.

Определение 18. **DataBag** – объект, содержащий в себе Т-дерево $data$ и список расширенных нейронных сетей $nets$.

Предлагаемая архитектура сетей является обобщением существующих сетей прямого распространения. Отличие между ними состоит в том, что сети прямого распространения принимают на вход данные (векторы определенных размеров), расширенные нейронные сети кроме данных будут принимать на вход **DataBag**, который кроме данных содержит также список сетей. Благодаря этому становится возможным менять архитектуру сети в зависимости от входных данных, «встраивая» сети-аргументы в тело основной сети. Данное обобщение позволяет естественным образом получить концепцию разделения весов (weight sharing), потому что матрицы весов сетей-аргументов будут общими для всех их вхождений в структуру основной сети.

Данное определение позволяет нам ввести новые виды слоев:

Определение 19. **Слой-переменная** – слой с фиксированным параметром pos. Принимает $DataBag$ и возвращает Т-дерево $DataBag.data.children[pos]$.

Определение 20. **Слой-сетевая переменная** – слой с фиксированным параметром pos. Принимает $DataBag$ и возвращает сеть $DataBag.nets[pos]$.

Семантика данных слоев состоит в том, что они возвращает некоторую часть входных данных, не применяя к ним никаких преобразований.

Тот факт, что теперь сеть может вернуть сеть, заставляет нас расширить понятие композиции сетей. Раньше мы просто «соединяли» выходы предыдущего слоя со входом следующего, но теперь нам следует рассмотреть 2 случая – когда первая сеть вернула данные и когда она вернула сеть. Для того, чтобы упростить дальнейшее изложение, определим слой, производящий данную операцию.

Определение 21. **Слой-применение** – слой, содержащий в себе две сети – net и $argument$, имеет фиксированный параметр $isData$, который показывает,

является результат $argument$ данными или сетью. Результат данного слоя – out – вычисляется по формуле:

$$
\begin{aligned}
argOut &= argument(in) \\
newArgs &= \begin{cases} DataBag(in.data \cup argOut, in.nets), & isData = true \\ DataBag(in.data, in.nets \cup argOut), & isData = false \end{cases} \\
out &= net(newArgs)
\end{aligned}
$$

Данный слой применяет сеть arg ко входным данным in. Результат дописывается к данным, если это данные, или к сетям в ином случае, и сеть net применяется к полученному $DataBag$. Благодаря тому, что мы дописываем результаты ко входным данным, сеть net может работать не только с новыми данными, но и с определенными ранее.

Заметим, что если бы мы использовали только конечные АТД, и лямбда-выражение, по которому строилась сеть, не содержало в своем составе функции высшего порядка, то мы могли бы обойтись обычными нейронными сетями. В этом случае входные данные представляли бы собой векторы (см пункт. 2.2.2), слои-переменные заменились бы на полносвязные слои с фиксированными матрицами весов, а слои-применения – на композицию слоев.

2.3.2. Построение расширенной нейронной сети

В данной работе рассматривается вывод алгоритмов обработки произвольных данных, поэтому мы в дальнейшем будет использовать расширенные нейоронные сети. Для их построения в предыдущем пункте был определен ряд слоев, используем их для проектирования сети на основе лямбда-выражения. Алгоритм построения приведен в Листинге 2.

Из структуры алгоритма следует, что построенные сети не содержат полносвязных слоев, поэтому не могут быть обучены. Полносвязные слои тесно связаны с сопоставлением с образцом, который будет рассмотрен в пункте 2.4. В этом же пункте архитектура сетей будет расширена, что позволит проектировать обучаемые сети.

2.4. Управляющие структуры

Управляющие структуры в языках программирования представляют собой циклы и ветвления. В функциональных языках циклы реализуются при по-

Листинг 2 – Построение нейронной сети на основе лямбда-выражения

```
// expr – лямбда-выражение
// data – отображение из имен переменных в позиции данных, соответствующих переменной
// nets – отображение из имен переменных-функций в позицию соответствующей сети
// pointer – пара из двух значений: data – количество аргументов-данных, nets – количество аргументов-сетей
// Возвращает сеть, принимающую DataBag
function BuildNet(expr, data, nets, pointer)
    if expr – переменная с именем name then
        if name ∈ data then
            pos ← data[name]
            return Слой-переменная с параметром pos
        else
            pos ← nets[name]
            return Слой-сетевая переменная с параметром pos
        end if
    else if expr – абстракция λ x. E then
        if тип x – T – АТД then
            pos ← position.data + 1
            newVars ← vars ∪ {x → pos}
            return BuildNet(E, newVars, nets, {pos, position.nets})
        elseтип x – функция
            pos ← position.nets + 1
            newNets ← nets ∪ {x → pos}
            return BuildNet(E, vars, newNets, {position.data, pos})
        end if
    else if expr – аппликация (E1 E2) then
        argNet ← BuildNet(E2, vars, nets, position)
        net ← BuildNet(E1, vars, nets, position)
        isData ← true если тип argNet – АТД, иначе false
        return Слой-применение с сетями net и argNet и параметром isData
    end if
end function
```

мощи рекурсии, ей посвящен раздел 2.5, ветвление реализовано при помощи сопоставления с образцом.

Для обработки структурированных данных ранее были определены T-деревья (пункт 2.2.4) и O-деревья (пункт 2.2.5), обобщающих понятия вектора и матрицы и позволяющих представлять данные в виде набора тензоров. В

данном пункте будет показано, как с использованием данных структур можно выразить полносвязные слои и реализовать операции ветвления и сопоставления с образцом.

2.4.1. Конструкторы алгебраических типов данных

Определение 22. **Определение АТД** – выражение, определяющее АТД, задаваемое в грамматике:

$$Def = C \mid C\ Def_1\ Def_2\ ...\ Def_n \mid Def\ '|'\ Def$$

Определение АТД представляет собой набор альтернатив (разделенных символом |), каждая из которых определяется **конструктором** – функций, которые создают объекты данного типа. Конструкторы без аргументов задают типы-литералы, конструкторы с аргументами задают типы-произведения. На основании данного определения задается АТД, который представляет собой тип-сумму альтернатив.

Конструктор без аргументов определяется позицией соответствующего типа-литерала в составе типа, конструкторы с аргументами определяются позициями аргументов в составе типа. Рассмотрим несколько типов и укажем позиции, определяющие конструкторы:

$$\begin{aligned} Bool &= True \mid False \\ True &\leftrightarrow 0 \\ False &\leftrightarrow 1 \\ Maybe &= Just\ Bool\ Bool \mid Nothing \\ Just &= \leftrightarrow [0, 1] \\ Nothing &= \leftrightarrow 2 \end{aligned}$$

Для того, чтобы добавить конструкторы АТД в лямбда-исчисление, определим для них правило редукции. Конструкторы типа A, не имеющие аргументов, редуцируются в объекты типа A. Аппликации вида $C\ E_1\ E_2\ ...\ E_n$, где C – конструктор с типом $T_1 \to T_2 \to ... \to T_n \to A$, все E_i имеют тип T_i – АТД и A- АТД, будут редуцироваться в объект типа A, где E_i расположены на соответствующих им позициях в структуре объекта.

В пункте 2.3 на основе лямбда-выражения была построена нейронная сеть. Расширим алгоритм построения для того, чтобы можно было выражать конструкторы при помощи нейронных сетей.

Рассмотрим тип с длиной определения L и построим структурированные полносвязные слои для его конструкторов:

— Если конструктор не имеет аргументов, то он определяется позицией литерала (pos) в составе типа. Данному конструктору соответствует структурированный полносвязный слой с линейной функцией активации, где $bias$ отсутствует, а $weights$ равен

$$
\begin{aligned}
weights = \{ & \\
& tree : \{ \\
& \quad data : data(\text{Вектор длины } L) \\
& \quad children : [null\ null\ ...\ null](\text{длины } L) \\
& \} \\
& children : [] \\
\} &
\end{aligned}
$$

$$
data[i] = \begin{cases} 0, & i \neq pos \\ 1, & i = pos \end{cases}
$$

— Если конструктор имеет N аргументов, то он определяется списком позиций данных аргументов в составе типа (список pos длины N). Такому конструктору соответствует структурированный полносвязный слой с линейной функцией активации, где $bias$ отсутствует, а $weights$ равен

$$
\begin{aligned}
weights = \{ & \\
& tree : \{ \\
& \quad data : data(\text{Матрица размера } N \times L) \\
& \quad children : [null\, null\, ...\, null](\text{длины } L) \\
& \} \\
& children : [] \\
\} &
\end{aligned}
$$

$$data[i,j] = \begin{cases} 0, & j \neq pos[i] \\ 1, & j = pos[i] \end{cases}$$

Определенные подобным образом слои соответствуют конструкторам из лямбда-исчисления. Согласно определению структурированного полсносвязного слоя, в результате применения подобных слоев аргументы будут помещены на требуемые позиции, остальные же позиции становятся равными 0. Данная операция соответствует конструированию объекта из составных частей.

Таким образом, конструкторы данных могут быть представлены при помощи расширенных нейронных сетей, благодаря чему становится возможным строить сети на основе нерекурсивных программ, содержащих подобные функции.

2.4.2. Сопоставление с образцом

В функциональном программировании операция ветвления выражается при помощи сопоставления с образцом. Для этого для каждого выражения задается несколько альтернатив, из которых выбираются те, которые подходят под образцы. Альтернативы задаются в грамматике:

$$A := name\ P_1\ P_2\ ...\ P_n = E$$

Где $name$ – имя лямбда-выражения, P_i – образец, с которым сопоставляется i-й аргумент выражения, E – само выражение. К примеру, оператор if можно задать следующим образом:

Листинг 3 – Задание оператора if

```
Bool = False | True

if True first second = first
if False first second = second
```

Если первый аргумент if представляет собой объект $True$, то выбирается первая альтернатива, иначе вторая. Данное поведение можно воспроизвести при помощи нейронных сетей, для этого стоит воспользоваться понятием *соответствия*. Напомню, что это величина, лежащая в интервале $[0, 1]$ и показывающая насколько точно значения вектора соответствуют ограничениям типа.

Расширенные нейронные сети работают с Т-деревьями, в которых каждый слой содержит вектор. Длина вектора равна длине определения типа, i-я компонента вектора равна величине соответствия объекта, описываемого i-м ребенком дерева, его типу. В качестве примера рассмотрим объект $True$ и его Т-дерево $TensorTree([0,\ 1],\ [null,\ null])$. В нем компонента, соответствующая $False$, равна 0, а соответствующая $True$ – 1.

Тот факт, что типы результатов всех альтернатив совпадают, а величина соответствия заключена в интервале $[0, 1]$, позволяет разработать дифференцируемый аналог сопоставления с образцом.

Определение 23. **Соответствие Т-дерева образцу** – число в промежутке $[0, 1]$, вычисляемое по формуле

$$Corr(t, pattern) = \begin{cases} 1, & \text{если } pattern{=}x \\ t.data[pos], & \text{если } pattern{=}C \\ \prod\limits_{\substack{i \\ j=pos[i]}} t.data[j] \cdot & \\ Corr(t.children[j], P_i) & , \quad \text{если } pattern{=}(C\ P_1\ P_2\ ...\ P_n) \end{cases}$$

где pos – позиции, соответствующие конструкторам.

Таким образом, для того, чтобы посчитать, насколько каждый аргумент соответствует образцу, мы перемножаем соответствия объектов, задаваемых требуемыми конструкторы. Если хотя бы один из них будет отсутствовать, величина соответствия образцу станет равной нулю.

Определение 24. **Слой-альтернатива** – слой нейронной сети, содержащий в себе образец $pattern$ и сеть net, обрабатывающую данные. Результат данного слоя вычисляется по формуле:

$$out = Corr(in, pattern) \cdot net(in)$$

Определение 25. **Сопоставляющий слой** – слой сети, который содержит в себе список слоев-альтернатив $nets$. Результат данного слоя вычисляется по формуле:

$$out = \sum_i nets[i](in)$$

Сопоставление с образцом реализуется при помощи сопоставляющего слоя. Он передает свои входные данные на вход сетей-альтернатив, которые независимо друг от друга дают набор ответов. Затем каждый ответ умножается на величину соответствия входных данных образцу этой альтернативы. Если данные для какой-то альтернативы не соответствуют образцу, то её результат умножится на 0 и не внесет вклад в результат сопоставляющего слоя. Только те альтернативы, образцам которых соответствуют входные данные, повлияют на результат слоя, что соответствует идее сопоставления с образцом.

Сопоставляющий слой поддерживает процедуру обратного распространения ошибки, благодаря тому, что сопоставление с образцом было сделано на основе дифференцируемых операций. Это позволяет нам расширить функционал нейронных сетей, добавив в них ветвления, зависящие от условий, накладываемых на входные данные. Это практически вплотную приближает нас к полноценному программированию на основе нейронных сетей.

2.4.3. Пропуски в лямбда-выражениях

Цель данной ВКР заключается в индукции алгоритмов при помощи нейронных сетей. Для того, чтобы добавить возможность вывода алгоритмов, расширим понятие лямбда-выражения.

Определение 26. **Расширенное лямбда-выражение** – лямбда выражение, содержащее пропуски. Выражения такого вида задаются в грамматике:

$$ExtLambda = E \mid ?$$

Где E – лямбда-выражение, $?$ – пропуск

Выражения подобного вида содержат в своем теле пропуски, которые требуется заполнить и тем самым завершить вывод алгоритма. Пропуск представляет собой функцию, чье тело неизвестно. В то же время после вывода типов нам известны типы каждого пропуска. В данной работе мы будем рассматривать пропуски, имеющие типы $T_1 \to T_2 \to ... \to T_n \to R$, где T_i и R – АТД.

В процессе построения сети на месте подобного пропуска будет строиться структурированный полносвязный слой, принимающий Т-дерево с n детьми и выдающий Т-дерево, соответствующее типу R. Параметры данного слоя будут организованы в О-дерево $weights$ и Т-дерево $bias$, в качестве функции активации используется сигмоида. Параметры данного слоя будут находиться в

процессе обучения, а результат вычисляться по формуле:

$$out = \sigma(weights \cdot in + bias) \tag{8}$$

Покажем, как построить О-дерево весов и Т-дерево смещений по типовой информации. Каждый пропуск, и как следствие и построенные деревья, будем описывать двумя числами:

— *inDepth* – показывает, сколько слоев аргументов сети использовать для вычисления результата

— *inDepth* – показывает, сколько слоев результата мы будем получать на основе аргументов

Приведем процедуру построения данных деревьев. Алгоритм 4 строит случайное Т-дерево на основе типа его структуры, глубины и числа строк в тензорах. Алгоритм 5 строит О-дерево на основе типа его структуры, глубины, типа Т-деревьев в составе и их глубины.

Листинг 4 – Построение случайного Т-дерева

```
// inSize – количество строк в тензорах
// outType – тип дерева
// outDepth – глубина
function BuildTensorTree(inSize, outType, outDepth)
    outSize ← длина определения outType
    tensor ← случайный тензор размера inSize × outSize
    children ← [null, ..., null] (длины outSize)
    if outDepth > 0 then
        // Должны создать еще уровень дерева
        for all childType – тип в составе outType do
            i ← позиция childType в составе типа
            if childType – АТД then
                children[i] = BuildTensorTree(inSize, childType, outDepth − 1)
            end if
        end for
    end if
    return TensorTree(tensor, children)
end function
```

Листинг 5 – Построение случайного О-дерева

```
// inType – тип структуры дерева О-дерева
// inDepth – глубина О-дерева
// outType – тип Т-деревьев в составе О-дерева
// outDepth – глубина Т-деревьев
function BuildOperatorTree(inType, inDepth, outType, outDepth)
    inSize ← длина определения inType
    tree ← BuildTensorTree(inSize, outType, outDepth)
    children ← [null, ..., null] (длины outSize)
    if inDepth > 0 then
        // Должны создать еще уровень дерева
        for all childType – тип в составе inType do
            i ← позиция childType в составе типа
            if childType – АТД then
                child=BuildOperatorTree(childType,inDepth-1,outType,outDepth)
                children[i] = child
            end if
        end for
    end if
    return OperatorTree(tree, children)
end function
```

Используя данные алгоритмы мы можем задать деревья $weights$ и $bias$ как:

$$
\begin{aligned}
inType &= (T_1, T_2, ..., T_n) \text{ – тип кортежа из } T_i \\
outType &= R \\
weights &= BuildOperatorTree(inType, inDepth, outType, outDepth) \\
bias &= BuildTensorTree(1, outType, outDepth)
\end{aligned}
$$

Результат данного слоя вычисляется по Формуле 8, которая содержит умножение входных данных на О-дерево $weights$ и прибавление $bias$. О-дерево $weights$ строится таким образом, что при умножении его на входные данные мы получим объект результирующего типа. $bias$ строится по типу результата и не зависит от входных данных, поэтому играет роль постоянного смещения результирующего объекта. Поэтому подобное определение аналогично операциям, проводимым в обычном полносвязном слое.

inDepth и *outDepth* позволяют регулировать количество параметров, которые содержатся в слое. Чем больше *inDepth*, тем глубже O-дерево и тем больше слоев входных данных мы используем для построения результата. Чем больше *outDepth*, тем глубже T-деревья и тем больше слоев результата мы можем получить на основе входных данных. Поэтому регулируя данные параметры можно настраивать то, насколько сложные зависимости можно будет находить с использованием данного слоя.

Благодаря тому, что операции, производимые при умножении, являются тензорными, к ним можно применить процесс обратного распространения ошибки, что позволит найти значения O- и T- деревьев методом градиентного спуска.

2.5. Рекурсивные нейронные сети

Используя результаты предыдущих пунктов возможно проектировать сети, обрабатывающие данные конечной структуры. Для обработки произвольных данных перенесем понятие рекурсии с лямбда-выражений на нейронные сети.

Известно, что комбинатор неподвижной точки, позволяющий задавать рекурсивные программы, не типизируется в системе типов XM. Данное ограничение можно обойти, введя оператор рекурсии в синтаксис лямбда-выражений:

Определение 27. **Рекурсивное лямбда-выражение** – лямбда-выражение, задаваемое в грамматике

$$R = ExtLambda \mid rec\ f.\ R$$

где $ExtLambda$ – расширенное лямбда-выражение, rec – оператор неподвижной точки.

В процессе редукции выражение $rec\ f.\ R$ заменяется на $R[f = R]$, то есть все вхождения f в R заменяются на R, чем достигается возможность рекурсивного применения выражения к самому себе.

Разработанный в предыдущих главах подход позволяет проектировать сети, принимающие в качестве аргументов другие сети. Это позволяет нам легко разработать сетевую альтернативу оператору неподвижной точки.

Определение 28. **Рекурсивный слой** – слой, содержащий в себе сеть E, результат применения которого вычисляется по формуле:

$$out = E(in.data, in.nets \cup E)$$

В процессе применения данного слоя внутренняя сеть добавляется в список сетей и передается самой себе в качестве аргумента. Поэтому она может быть встроена в граф вычислений неограниченное количество раз в процессе исполнения. В процессе построения сети по выражению, оператор неподвижной точки будет заменяться на данный слой.

Использование рекурсии в сочетании с сопоставлением с образом позволяет проектировать расширенные сети, обрабатывающие рекурсивные типы данных. Например, сеть, которая учится отображению элементов массива, может быть построена по коду из Листинга 6. Пропуск в теле данного выражения будет заполнен в процессе обучения построенной сети.

Листинг 6 – Отображение массива

```
MyType = ... // Definition of some type
List =  Empty | Cons MyType List

map Empty = Empty
map (Cons x rest) = Cons (? x) (map rest)
```

В данном примере одна и та же сеть, построенная на основе пропуска, применяется к данным, находящимся на разных уровнях структуры списка. Это соответствует идее рекурсивных нейронных сетей, которые обобщают рекуррентные сети. Используя данный подход можно строить сети, которые будут обрабатывать более сложные структуры данных и обучаться алгоритмам работы с ними.

На данный момент управляющим структурам из функциональных языков были найдены соответствия в классе расширенных нейронных сетей. Благодаря этому становится возможным строить подобные сети на основе программ, а обучаемость сетей позволит заполнять пропуски в коде программ, тем самым производить индукцию алгоритмов.

2.6. Полиморфные нейронные сети

В данном пункте мы перенесем понятие типового полиморфизма на нейронные сети, что позволит нам обрабатывать данные разной структуры при помощи одной и той же сети.

Определение 29. **Полиморфная нейронная сеть** – класс расширенных нейронных сетей, способных обучаться алгоритмам обработки данных, не зависящим от структуры данных.

2.6.1. Построение сети по полиморфному выражению

В пункте 2.3 был рассмотрен алгоритм построения расширенной сети по выражению, не содержащему let-абстракций. Из-за их отсутствия типы всех выражений не содержали переменных, что упрощало построение сети. В данном пункте мы добавим поддержку типовых переменных, полиморфных выражений и let-абстракции.

Полиморфное выражение имеет тип $\forall\ a_1\ a_2\ ...\ a_n.\ T$, где a_i - переменные, T – АТД. На место типовых переменных можно подставить другие АТД, вопрос подстановки функций на место переменных в данной работе не рассматривается. В результате подстановки мы получаем **экземпляр (instance)** выражения, сам процесс подстановки будем называть **инстанцированием**.

Выражение $let\ x\ =\ E_1\ in\ E_2$ определяет полиморфное выражение x, которое затем может быть использовано в E_2. Каждое вхождение x в E_2 инстанцируется соответствующими типами, процесс вывода типа гарантирует, что все переменные были заменены на корректные типы. Каждый экземпляр, как и само полиморфное выражение, задается выражением E_1, структура которого не меняется в процессе инстанцирования.

Мы можем строить сети по выражениям, содержащим let-абстракции, аналогично описанному ранее способу, так как с вычислительной точки зрения $let\ x\ =\ E_1\ in\ E_2$ эквивалентно $(\lambda\ x.\ E_2)\ E_1$. Единственное отличие состоит в том, что выражение x необходимо будет инстанцировать каждый раз, когда оно встречается в составе выражение E_2. Таким образом, мы можем заменить let-абстракции на $(\lambda\ x.\ E_2)\ E_1$, строить сеть по ним, и затем инстанцировать каждое вхождение сети, построенной по E_1, в составе E_2.

Рассмотрим процесс инстанцирования сети. Можно заметить, что в процессе построения сети по выражению для нас было важно, является выражение АТД или функцией, но мы не учитывали структуру АТД. Единственными ме-

стом, где структура данных играла роль, было заполнение пропусков. Поэтому процесс инстанцирования может влиять только на слои, стоящие на местах пропусков.

Как было показано раньше, пропуски заменяются на структурированные полносвязные слои, содержащие O-деревья и T-деревья со структурой, определяемой типами аргументов и результата. В случае полиморфных выражений данные типы могут также содержать параметры в составе типов-произведений.

Можно заметить, что алгоритмы построения O- и T- деревьев не зависели от наличия в типе переменных, поэтому мы можем строить деревья используя те же алгоритмы. Операции умножения O-деревьев и T-деревьев определены таким образом, что что операция инстанцирования не требует изменения структуры O-дерева.

Благодаря этому, при инстанцировании не требуется менять структуру сети. Поэтому мы можем обучить сеть обработке данных одной структуры, а затем применять ее данным другого формата, что соответствует идее полиморфизма.

2.6.2. Обработка данных полиморфной структуры

Также стоит рассмотреть АТД, содержащие параметры. Подобные типы задаются выражением:

$$PolyADT := name\ a_1\ a_2\ ...\ a_n = T$$

где $name$ – имя типа, a_i – типовые параметры, T – определение АТД, в структуре которого могут использоваться параметры. В данной работе рассматриваются АТД, где каждый типовый параметр имеет сорт *****.

Данные типы также инстанцируются при каждом использовании. Можно заметить, что типы входных данных всегда будут инстанцированным, потому что мы не может задать объект с произвольным типом согласно изоморфизму Карри-Ховарда и законам логики.

Отличие данного вида АТД от рассмотренного ранее состоит в том, что его конструкторы являются полиморфными выражениями. Например можно рассмотреть тип $Maybe$ и записать типы его конструкторов:

$$Maybe\ a = Just\ a \mid Nothing$$

$$Just : \forall\ a.\ a \to Maybe\ a$$
$$Nothing : \forall\ a.\ Maybe\ a$$

Конструкторы инстанцируются при каждом их вхождении в тело лямбда-выражения. Однако они как и прежде задаются позициями аргументов в структуре результирующего объекта:

$$Just \leftrightarrow\ [1]$$
$$Nothing \leftrightarrow\ 2$$

Данные позиции не будут меняться в процессе инстанцирования, поэтому сети, содержащие полиморфные конструкторы не будут перестраиваться при инстанцировании. То же самое можно сказать про сопоставление с образцом с использованием подобных конструкторов.

Аргументы и результирующие значения пропусков также могут быть полиморфными АТД. Они обрабатываются аналогично параметрам в пункте 2.6.1. Благодаря определениям О- и Т-деревьев и операциям умножения сети смогут обучаться в том числе и работе с подобными полиморфными АТД.

Выводы по главе 2

В данной главе были дано теоретическое описание соответствия между лямбда-выражениями и расширенными нейронными сетями – сетями, которые могут принимать в качестве аргументов как данные, так и другие сети. Данное обобщения понятия нейронной сети позволило найти дифференцируемые аналоги управляющих структур функциональных языков программирования, таких как сопоставление с образцом и рекурсия. Был приведен алгоритм построения расширенной сети на основе подобных лямбда-выражений, благодаря чему становится возможно строить сети, способные обучаться широкому классу алгоритмов.

Кроме того, в данной главе было дано определение Т-деревьев и О-деревьев, благодаря которым стало возможно обрабатывать структурированные данные при помощи нейронных сетей.

В завершении главы было дано понятие полиморфных нейронных сетей, которые являются аналогом понятия полиморфизма, используемого в програм-

мировании. Сети данного класса могут находить алгоритмы обработки данных, не зависящие от их структуры.

ГЛАВА 3. ЯЗЫК FNN И ИНДУКЦИЯ АЛГОРИТМОВ

В данной главе приводится описание и детали реализации функционального дифференцируемого языка **FNN**. Программы на данном языке могут содержать пропуски и компилируются в расширенные нейронные сети, которые могут заполнить пропуски в процессе обучения.

3.1. Описание языка FNN

FNN (**F**unctional **N**eural **N**etworks) – функциональный дифференцируемый язык программирования с синтаксисом, подобным языку Haskell. Программы на данном языке представляют собой набор определений алгебраических типов данных и лямбда-выражений. Язык поддерживает рекурсивные АТД, рекурсивные лямбда-выражения и сопоставление с образцом, что обеспечивает его Тьюринг-полноту.

Кроме того, язык поддерживает параметрический полиморфизм: определения АТД и типы лямбда-выражений могут содержать параметры, которые заменяются на конкретные типы в зависимости от контекста. На данный момент поддерживается замена типовых переменных на АТД и на другие переменные, замена переменной на функцию на данный момент не поддерживается.

Отличительной особенностью языка является то, что он может содержать пропуски - функции, код которых неизвестен. Пропуски могут быть заполнены с использованием машинного обучения, чем обеспечивается возможность индукции алгоритмов с использованием данного языка.

Код программ на языке **FNN** представляет собой набросок алгоритма, который компилируется в расширенную нейронную сеть. Известные участки кода заменяются на фиксированные слои, а пропуски – на структурированные полносвязные слои. Их параметры находятся в процессе обучения, в результате чего заполняются пропуски в наброске алгоритма.

3.1.1. Синтаксис и семантика

Рассмотрим синтаксис языка и объясним семантику его конструкций. Программы на языке **FNN** представляют собой набор выражений, разделенных символом «точка с запятой» (`;`). Каждое выражение может быть:

— определением АТД;
— декларацией типа лямбда-выражения;
— определением лямбда-выражения.

Имена всех выражений в структуре программы должны состоять из строчных и прописных латинских буква, цифр и знака подчеркивания. Имена типов и конструкторов должны начинаться с прописной буквы, имена типовых переменных и лямбда-выражений – со строчной.

Определения АТД задаются в грамматике:

$$
\begin{aligned}
TypeDefinition &:= \text{@}\textbf{type}\ TypeName\ TypeParam* \\
&\quad = SumOperand\ (|\ SumOperand)* \\
SumOperand &:= ConstructorName\ (ProdOperand)* \\
ProdOperand &:= TypeParam \text{ или } TypeName \\
TypeParam &:= \text{имя типовой переменной} \\
TypeName &:= \text{имя типа} \\
ConstructorName &:= \text{имя конструктора}
\end{aligned}
$$

Декларацию типа лямбда-выражения можно указать в грамматике:

$$
\begin{aligned}
TypeDeclaration &:= \text{@}\textbf{type}\ ExprName = PolyType \\
ExprName &:= \text{имя лямбда-выражения} \\
PolyType &:= \text{@}\textbf{forall}(TypeParam) +\ .\ Type \text{ или } Type \\
Type &:= TypeName\ (Type) * \text{ или } TypeParam \text{ или} \\
&\quad Type \to Type \text{ или } (Type)
\end{aligned}
$$

Лямбда-выражения задаются в грамматике:

$$
\begin{aligned}
ExprDefinition &:= ExprName\ Pattern* = Expr \\
Pattern &:= ExprName \text{ или } (ConstructorName\ Pattern*) \\
Expr &:= ExprName \text{ или } ConstructorName \text{ или} \\
&\quad \text{@}\textbf{learn}(Options)? \text{ или} \\
&\quad \backslash ExprName +\ .\ Expr \text{ или} \\
&\quad \text{@}\textbf{rec}\ LetBinding +\ \text{@}\textbf{in}\ Expr \text{ или} \\
&\quad \text{@}\textbf{rec}\ ExprName\ \text{@}\textbf{in}\ Expr \text{ или} \\
&\quad \text{@}\textbf{case}\ Extr\ \text{@}\textbf{of}\ Case +\ \text{ или} \\
&\quad (Expr+) \text{ или } (Expr\ :\ PolyType)
\end{aligned}
$$

$$
\begin{aligned}
Options &:= \{Option\ (, Option)+\} \\
Option &:= ExprName = Number \\
LetBinding &:= ExprName = Expr \\
Case &:= Pattern\ \rightarrow\ Expr
\end{aligned}
$$

Определения АТД позволяют задавать типы данных, с которыми будут работать сети, семантика данной конструкции соответствует данной в главе 2. Если определяемый тип встречается в правой части определения, то комбинатор неподвижной точки для типов подставляется автоматически. Декларации типа позволяют уточнить тип выражения, данная информация будет использована в процессе вывода типов.

Конструкции из определения лямбда выражения соответствуют грамматике лямбда-выражения из главы 2:

— **@learn** задает пропуск в коде, может принимать параметры;
— **\...** определяет лямбда-абстракцию;
— **@let ... @in** определяет let-абстракцию;
— **@rec ... @in** задает оператор неподвижной точки;
— **@case ... @of** задает сопоставление с образцом;
— (Expr+) позволяет задать аппликацию.

Отличия от определений состоят в том, что теперь все конструкции могут принимать не один, а несколько аргументов. Таким образом, абстракции могут определять несколько переменных, в аппликации может быть больше двух операндов. Данные конструкции автоматически раскрываются в последовательности их аналогов из главы 2. Также **@learn** может принимать параметры, которые описывают, сколько уровней структуры аргументов должно быть использованы при обучении и сколько уровней структуры результата могут быть получены на их основе: `from_depth` и `to_depth` соответственно.

Семантика конструкций языка соответствует данной в главе 2 – конструкции определяют слои расширенной нейронной сети согласно алгоритму пункта 2.3.2. Исполнение программы на данном языке соответствует распространению информации по слоям построенной сети.

3.1.2. Runtime-модуль для PyTorch

Отличие расширенных нейронных сетей от обычных состоит в том, что последовательность операций, которую выполняет сеть, строится в процессе

исполнения, а не задается заранее. Это повышает гибкость архитектуры, но усложняет процесс обратного распространения ошибки.

Для поддержки динамической последовательности операций требуется автоматическое дифференцирование с произвольным графом вычислений. Данная технология позволяет вычислять градиенты тензорных операций, последовательность которых не фиксируется заранее. На данный момент существует ряд библиотек, поддерживающих автоматическое дифференцирования, в данной работе из них был выбран **PyTorch**.

PyTorch представляет собой фреймворк для машинного обучения с использованием нейронных сетей. **PyTorch** написан на языке **Python**, и главной его особенностью является поддержка произвольных графов вычисления, которые определяются также на языке **Python**. Кроме того **PyTorch** поддерживает технологию автоматического дифференцирования (см. статью [18]), благодаря чему при использовании тензорных операций все градиенты вычисляются автоматически.

Для реализации архитектуры расширенных нейронных сетей был разработан модуль времени исполнения, определяющий структуры данных и слои из 2 главы:

- **TensorTree** – T-дерево, определенное в пункте 2.2.4. Для них также определены операции поточечного умножения, сложения, умножения дерева на матрицы и T-деревья, согласно их описанию в пункте 2.2.5.
- **OperaorTree** – O-дерево с операцией умножения на T-дерево, определенные в пункте 2.2.5.
- **DataBag** – структура данных, содержащая TensorTree и список сетей.
- **FunctionalModule** – основной модуль, требуемый для реализации расширенных нейронных сетей (пункт 2.3.1). Принимает DataBag и может вернуть как данные, так и сеть.
- **ConstantLayer** – слой, который строится по конструктору без аргументов (пункт 2.4.1).
- **ConstructorLayer** – слой, строящийся по конструктору с аргументами (пункт 2.4.1).
- **VariableLayer.Net** – слой-сетевая переменная (пункт 2.3.1).
- **VariableLayer.Data** – слой-переменная (пункт 2.3.1).
- **RecursiveLayer** – рекурсивный слой (пункт 2.5).

— **GuardedLayer** – сопоставляющий слой (пункт 2.4.2).
— **ApplicationLayer** – применяющий слой (пункт 2.3.1).
— **TrainableLayer** – структурированный полносвязный слой (пункт 2.4.1).

Данные слои реализованы согласно их определениям. Благодаря тому, что вся логика была выражена на основе тензорных операций, все слои поддерживают автоматическое дифференцирование. Это позволяет обучать построенные расширенные сети при помощи стандартных методов градиентного спуска.

3.1.3. Обучение и индукция алгоритмов

Язык **FNN** можно использовать для проектирования расширенных нейронных сетей, способных обучаться алгоритмам обработки структурированных данных. Покажем, за счет чего это достигается.

Основу сетей, получаемых в результате компиляции, составляют структурированные полносвязные, сопоставляющие и рекурсивные слои. В то время как сопоставляющие и рекурсивные слои задаются заранее и служат «каркасом», определяющим процесс обработки данных, структурированные полносвязные слои строятся на основе пропусков программ и модифицируются в процессе обучения.

Как было отмечено выше, все операции, лежащие в основе слоев, являются тензорными, поэтому **PyTorch** автоматически вычисляет все градиенты в процессе обратного распространения ошибки. Благодаря этому для обучения расширенной сети можно использовать стандартные алгоритмы градиентного спуска.

Таким образом, процесс индукции алгоритма состоит из следующих шагов:

1) Определить типы данных, с которым работает алгоритм.
2) Сделать набросок алгоритма, описывающий, использует ли алгоритм рекурсию и сопоставление с образцом. Набросок может содержать пропуски.
3) Скомпилировать набросок в расширенную нейронную сеть.
4) Подать на вход сети с данные из обучающей выборки, вычислить функцию потерь от результата и ожидаемого ответа, изменить параметры сеть согласно вычисленному градиенту.
5) Повторять предыдущий пункт, пока не достигнута требуемая точность.

Можно заметить, что в процессе обучения не находится точная структура алгоритма. Найденный алгоритм представляет собой «черный ящик», который принимает данные, обрабатывает их и выдает ответ. Однако благодаря тому, что мы дали точную семантику функциям активации каждого слоя, мы можем понять, что представляют собой промежуточные значения.

Расширенные нейронные сети и язык **FNN**, предназначенный для их проектирования, можно использовать для вывода алгоритмов. Примеры подобного использования будут приведены в пункте 3.3.

3.2. Процесс компиляции языка

Как было сказано ранее, на основе программ на языке **FNN** можно строить расширенные нейронные сети. Для автоматизации данной процедуры был разработан компилятор, принимающий программу на данном языке и генерирующий файлы на языке **Python**, содержащие определения построенных сетей.

Компилятор написан на языке **Kotlin**. Причина данного выбора заключается в том, что данный язык сочетает в себе относительно строгую типизацию с лаконичным синтаксисом (см. ресурс [19]).

Процесс компиляции программы можно разделить на следующие этапы:

1) Предварительный этап – разбор текста программы, построение АСТ, группировка определений лямбда-выражений и типов.
2) Вывод типов АТД, конструкторов и лямбда-выражений.
3) Построение спецификаций сетей и типов.
4) Генерация кода АТД и сетей.

Рассмотрим каждый из этапов подробно.

3.2.1. Предварительные этапы компиляции

Первый шаг компиляции состоит в разборе текста программы и построение абстрактного синтаксического дерева (АСТ). Для данных целей используется фреймворк **ANTLR**, позволяющий задать грамматику языка программирования и сгенерировать лексер и парсер для данной грамматики (см. книгу [20]). Грамматика для языка **FNN** была задана в соответствии с приведенной в пункте 3.1.1.

После разбора текста программы для него строится АСТ, которое затем разбивается на три набора выражений – определения АТД, декларации типа и определения лямбда-выражений. После этого определения лямбда-выражений

группируются по их именам, благодаря чему становится возможным вывести одинаковый тип для всех альтернатив одного выражения, использующего сопоставления с образцом.

Результатом данного шага являются три списка – список определений АСТ, список деклараций типов лямбда-выражений и список самих выражений, сгруппированных по их именам.

3.2.2. Вывод типов

Вывод типов производится для каждой группы выражений по очереди, что позволяет использовать в лямбда-выражениях АТД, определенные в произвольном месте файла.

В первую очередь производится вывод типовой информации для АТД. Для каждого типа выводится следующая информация:

— список типовых параметров, сорт АТД;
— типы конструкторов;
— рекурсивен ли тип.

Изначально для вывода сортов АТД предлагалось использовать алгоритм, определенный в статье [21], однако специфика задачи позволила применить ряд эвристик и упростить данный процесс. Так как в данной работе рассматриваются АТД, где все типовые параметры имеют сорт ***** (см пункт 2.6.2), то вывод сорта АТД тривиален - сорт любого АТД имеет вид $k_1 \to k_2 \to ... \to k_n$, где все $k_i = *$. Кроме того упрощается проверка соответствия сортов типов – для того, чтобы удостовериться, что АТД с параметрами корректно инстанцирован, достаточно проверить, что число параметров АТД совпадает с числом параметров, указанных в применении типа.

На основании определения АТД строится список конструкторов, для которых проверяется, что сорта аргументов сходятся с определениями типов. Кроме того, если имя типа встречается в определении его структуры, то тип становится рекурсивным. Данная особенность будет влиять на сгенерированный код.

После вывода всех АТД все декларации типов лямбда-выражений проверяются на корректность. Это подразумевает проверку, что использованные в них типы были определены, и все АТД применяются к соответствующему количеству параметров.

Итоговым шагом является вывод типа лямбда-выражений. Для этого используется алгоритм W, определенный в статье [22]. Алгоритм был расши-

рен для поддержки сопоставления с образцом. Напомню, что на данном этапе все альтернативы лямбда-выражения сгруппированы по имени. Все выражения каждой группы должны иметь один и тот же тип, который должен согласовываться с определенной декларацией типа. Для обеспечения этого вывод типа производится согласно Алгоритму 7:

Листинг 7 – Вывод типа лямбда-выражения

function InferType($expr$)
    $types \leftarrow []$
    if тип $expr$ определен заранее **then**
        $types \leftarrow types \cup \{$Тип $expr\ \}$
    end if
    for all $case$ – альтернатива выражения $expr$ **do**
        $patterns \leftarrow$ образцы в составе $case$, длина списка k
        $caseExpr \leftarrow$ выражение в составе $case$
        // Выведем типы образцов алгоритмом W на основе типов конструкторов
        $pTypes \leftarrow W(patterns)$
        // Выведем тип выражения алгоритмом W
        $exprType \leftarrow W(caseExpr)$
        $caseType \leftarrow pTypes[1] \rightarrow ... \rightarrow pTypes[k] \rightarrow exprType$
        $types \leftarrow types \cup caseType$
    end for
    // Унифицируем типы всех альтернатив
    $finalType \leftarrow unify(types)$
    return $finalType$
end function

В процессе вывода мы сохраняем информацию о типе каждого выражения в составе исходного, кроме того мы определяем, является ли выражение рекурсивным, и если да, то подставляем оператор неподвижной точки автоматически.

В результате данного этапа мы имеем 2 списка – один содержит определения АТД с выведенной информацией о сорте и конструкторах, другой – информацию о типах лямбда-выражений.

3.2.3. Построение спецификаций сетей

На основании типовой информации строятся *спецификации* типов и выражений, при помощи которых в дальнейшем происходит генерация кода. Спецификации сохраняют исходную структуру выражений, но добавляют к ней некую вспомогательную информацию.

Каждая спецификация имеет свою **сигнатуру** – пару из списка имен и списка типов, которая однозначно (с точностью до имен переменных) определяет спецификацию. Список типов содержит сигнатуры спецификаций АТД, на которые были заменены типовые переменные в исходном выражении. Список имен содержит «путь» к определению выражения – первый элемент в данном списке равен имени выражения, каждая вложенная **@let**-абстракция добавляет новое имя в путь. К примеру, рассмотрим пути в выражении:

$$foo = @\textbf{let}\ id1 = (@\textbf{let}\ id2 = \backslash x.\ x\ @\textbf{in}\ id2)\ @\textbf{in}\ id1$$
$$foo \leftrightarrow [foo]$$
$$id1 \leftrightarrow [foo, id1]$$
$$id2 \leftrightarrow [foo, id1, id2]$$

Спецификация АТД содержит в себе свою сигнатуру, список типовых параметров, описание структуры типа, где для каждого аргумента в произведении указана его сигнатура.

Для поддержки замыканий в языке **FNN** был реализован аналог стека областей видимости. Все определяемые в лямбда-абстракции переменные помещаются на вершину стека, при применении выражения с вершины стека снимается требуемое количество аргументов. Данное поведение соответствует тому, что мы поддерживаем список входов сети, каждая абстракция добавляет в его конец новые входы, затем при применении слоев мы с конца списка требуемое количество входов и «соединяем» их со входами нашего слоя. Стек переменных поддерживается в процессе компиляции, в сгенерированном коде будут сохранены только указатели на позицию на данном стеке. Спецификация лямбда-выражения помимо сигнатуры для выражения и типа также содержит данные указатели.

После построения спецификаций для всех лямбда-выражений мы составляем список сигнатур использованных типов и выражений. Благодаря этому мы будем генерировать код только для использованных в коде экземпляров АТД и выражений. В результате данного этапа мы имеем списки спецификации АТД и выражений, а также информацию о том, какие из них были использованы в программе.

3.2.4. Генерация кода

На данном этапе по спецификациям слоев генерируются файлы на языке **Python**, содержащие описания расширенных нейронных сетей и типов данных, которыми они оперируют. Данные файлы используют модуль времени исполнения, определенный в пункте 3.1.2. Код генерируется в соответствии с правилами форматирования **PEP 8**, что упрощает его чтение.

На основании спецификаций типов строятся объекты, описывающие типы данных. Подобные объекты позволяют сгенерировать выборки для обучения. Выборки представляют собой Т-деревья соответствующего типа, которые затем можно подавать на вход построенным сетям. Кроме того, объекты типов можно инстанцировать с использованием языка **Python**, что позволяет задавать новые типы, пользуясь возможностями языка.

На основании спецификаций лямбда-выражений строятся расширенные нейронные сети, использующие Runtime-модуль, определенный в пункте 3.1.2.

Пример исходного кода программы и результата компиляции приведен в Приложении 1.1.

3.3. Эксперименты

Для проверки возможности индукции алгоритмов обработки данных произвольной длины при помощи разработанного подхода был проведен ряд экспериментов. Их описание приведено в следующих пунктах.

3.3.1. Схема вывода алгоритма

Вывод алгоритмов решения конкретной задачи производился по следующей схеме:

1) На языке **FNN** определяется требуемый алгоритм и его набросок с пропусками
2) Данные программы компилируются в расширенные нейронные сети
3) Обучающая и тестовая выборки соответствующего типа генерируются при помощи Runtime-библиотеки.
4) На вход обучаемой и проверочной сети подаются данные из обучающей выборки.
5) Для результатов обучаемой и проверочной сети вычисляется функция потерь, производится шаг градиентного спуска. Данные шаги повторяются, пока не достигнута требуемая точность.

6) На вход обученной и проверочной сети подаются данные из тестовой выборки, для результатов вычисляется функция потерь.

В качестве функции потерь использовалось MSE, для градиентного спуска применялся метод ADAM с параметром `learning rate` = 10^{-2}.

3.3.2. Арифметические операции над натуральными числами

Определим натуральные числа и произведем вывод алгоритмов вычисляющих сумму, разность и произведение натуральных чисел произвольной длины, а также сравнивающих данные числа.

Натуральные числа определим согласно арифметике Пеано:

$$\mathbb{N} = 0 \mid \mathbb{N} + 1$$

Для вывода данных операций над натуральными числами зададим примерное описание алгоритмов. Код, задающий числа и наброски алгоритмов, приведен в Приложении 2.

Все алгоритмы обучались на выборке, состоящий из пар чисел от 0 до 5, тестовая выбора для суммы, разности и сравнения состояла из чисел от 0 до 100, для произведения тестовая выборка состояла из чисел от 0 до 10.

На каждом шаге обучения вычислялось значение MSE для результатов работы сети и результатов требуемого алгоритма. Также на основе найденных мягких решений находились жесткие:

$$hardAnswer[i,j] = \begin{cases} 1, & \text{если } answer[i,j] > 0{,}7 \\ 0, & \text{иначе} \end{cases}$$

и считалось, на скольких образцах из выборки жесткие результаты сети отличаются от требуемых. Графики зависимости MSE и количества неверных ответов от числа шагов алгоритма обучения приводятся на Рисунке 1.

В результате проведения экспериментов было получено, что после достаточного количества итераций расширенные нейронные сети обучается всем требуемым алгоритмам. В частности, проверка на тестовой выборке показала, что индуцированные алгоритмы не ошиблись ни в одном образце. Точные значения проверки алгоритмов на тестовой выборке приведены в Таблице 3.

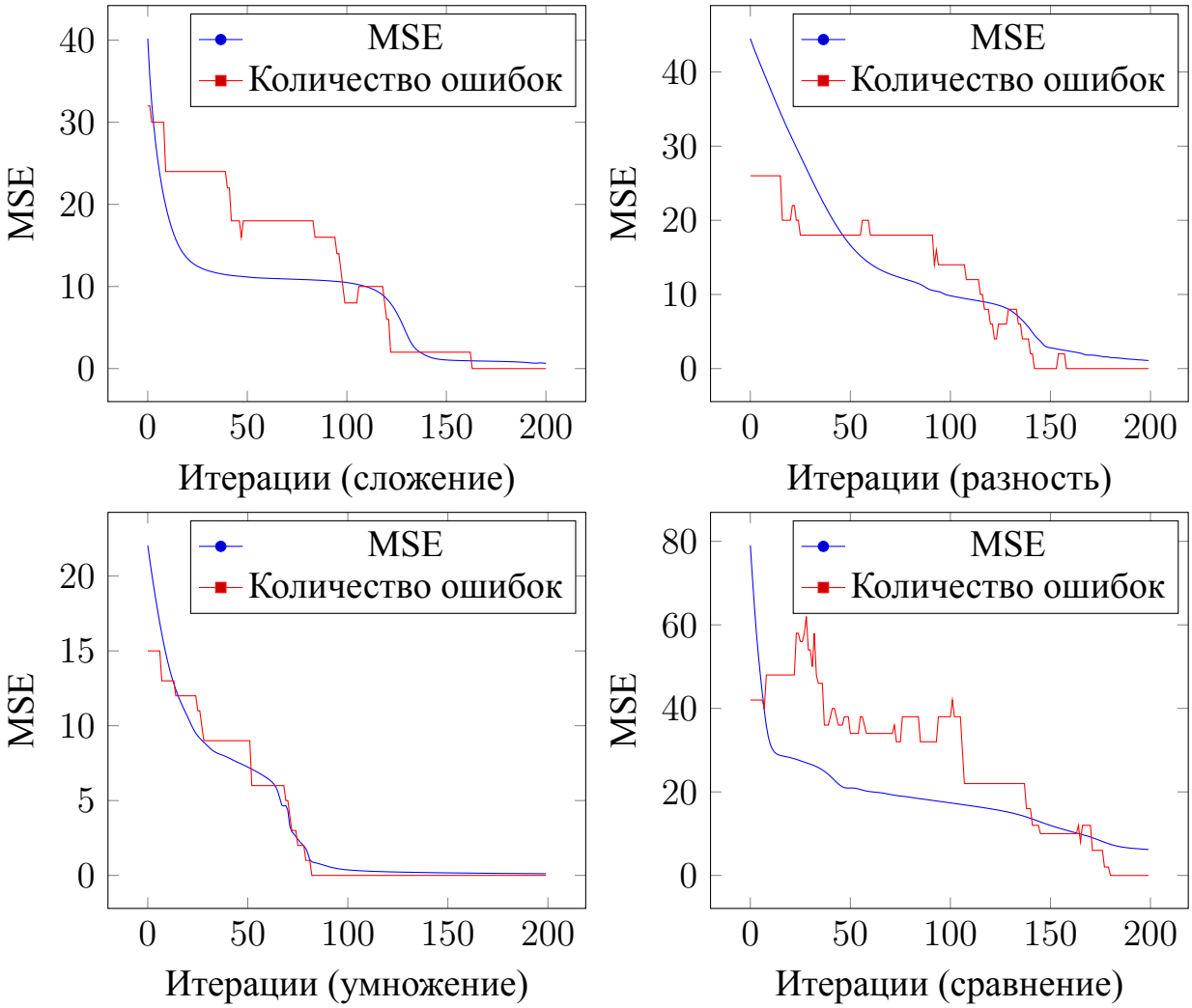

Рисунок 1 – Процесс обучения сети

Таблица 3 – Результаты обучения сети

Задача	MSE	Количество ошибочных ответов
Сложение	0,9407	0
Разность	1,1882	0
Умножение	3,2113	0
Сравнение	0,0288	0

Полученные результаты позволяют сказать, что используя сети, спроектированные на языке **FNN**, можно проводить индукцию алгоритмов при помощи машинного обучения.

3.3.3. Обработка списков

Приведем пример вывода рекурсивных полиморфных алгоритмов. В качестве таких задач выберем конкатенацию списков и индексацию списка. Пер-

вый алгоритм должен соединить два списка, поданных на вход, второй должен вернуть i-й элемент массива, либо значение по умолчанию, если список пуст.

Код, на основе которого производится построение обучаемых сетей, приведен в Приложении 3. Данный листинг содержит описания типов данных, наброска алгоритма и требуемого алгоритма индексации.

Обучение конкатенации и индексации производилось на списках, состоящих из данных типа $Bool$ длиной от 1 до 3. Тестовые данные состояли из 400 массивов длины от 1 до 6, содержащих элементы типов $Bool$, $Maybe\ Unit$, $Maybe\ Bool$, $Pair\ Bool\ Bool$ и $List\ Bool$ (длины от 1 до 3). Таким образом, всего было 5 проверочных выборок, каждая представляла собой набор списков одного из указанных типов данных.

В качестве функции оценки качества алгоритма была принята квадратичная функция потерь, вычисляемая между ожидаемым результатом и результатом сети. Зависимости MSE от номера итерации в процессе обучения приведены на Рисунке 3.

После обучения сетей им на вход были поданы данные из тестовой выборки, тип которых отличались от того, на каком была обучена сеть. Результаты были сопоставлены с требуемыми, в качестве критерия оценки качества был также выбран MSE. Результаты оценки работы сетей приведены в Таблице 4

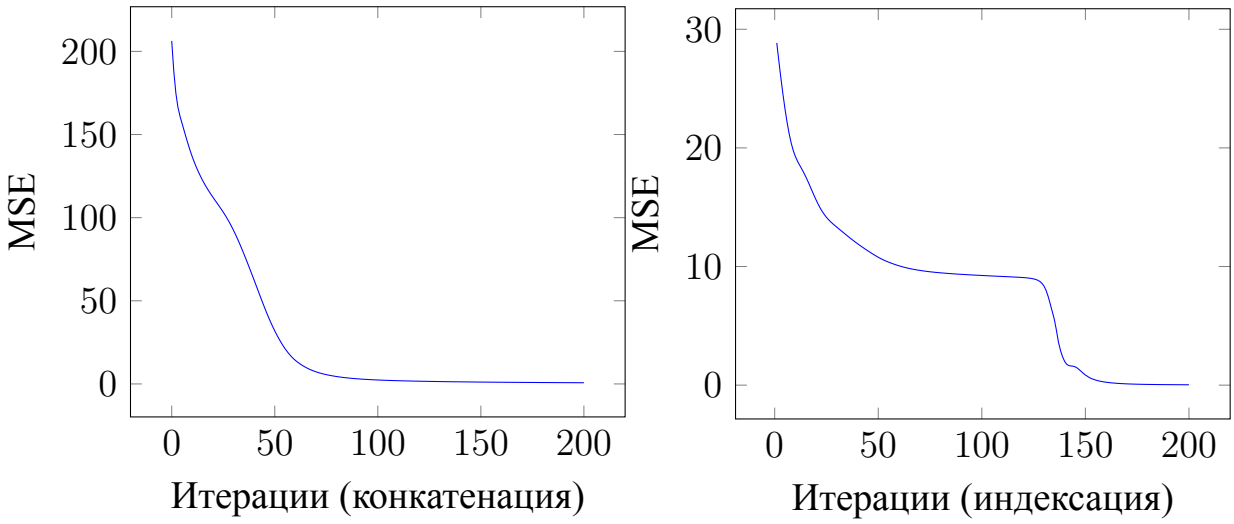

Рисунок 3 – Зависимость MSE от номера итерации

Результаты проведенного эксперимента показывают, что разработанная архитектура нейронных сетей позволяет обучаться полиморфным алгоритмам – в данном случае алгоритм был обучен по выборке одного типа данных, а затем

Таблица 4 – Результаты обучения сети

Тип данных списка	MSE (конкатенация)	MSE (индексация)
Bool	0,0789	0,3346
Maybe Unit	0,0789	0,3346
Maybe Bool	0,0561	0,0565
Pair Bool Bool	0,1880	0,0446
List Bool	0,1237	0,0096

применен к данным другого типа, причем при этом резкого падения точности обнаружено не было.

3.3.4. Обучаемая машина Тьюринга

Рассмотрим применение разработанного подхода к решению более сложных задач. В качестве примера подобной задачи реализуем машину Тьюринга на языке FNN, что позволит показать его полноту.

Код, задающий данный вычислитель, приведен в Приложении 4. В листинге определяются типы данных для бесконечной ленты (`Tape`, представляет собой пару из двух списков), состояний (`State`), алфавита (`Alphabet`) и переходов (`Move`) машины Тьюринга. Там же содержатся функции, осуществляющие переход по ленте (`moveLeft`, `moveRight`, `makeMove`). Сама машина Тьюринга задается функцией `machine`, которая принимает на вход функцию переходов (`transition`), ленту и состояние машины.

Результатом компиляции кода будет расширенная нейронная сеть, которая принимает на вход сеть, соответствующую функции переходов, объект, задающий ленту, и объект, задающий стартовое состояние. Код, определяющий вычислитель, не содержит пропусков, поэтому в процессе обучения модифицироваться не будет. Однако параметры сети, соответствующей функции переходов, могут модифицироваться в процессе обучения. Это позволит находить произвольные алгоритмы для машины Тьюринга, обучаясь по примерам.

В качестве примера рассмотрим вывод алгоритма инверсии битов двоичного числа. В данной задаче алфавит машины Тьюринга состоит из 3-х символов – `Blank`, `Zero`, `One`, машина имеет только одно состояние – `Start`. Выбор этого алгоритма обусловлен тем, что функция переходов, соответствующая точному решению, достаточно проста. Она содержится в листинге под именем `transitionRequired`. Кроме того приведены 2 варианта данной функции с пропусками:

— `transitionSimple` – правила перехода по ленте заданы, требуется выучить, какие символы писать на ленту;
— `transitionDifficult` – требуется выучить и правила переходов по ленте, и то, какие символы писать на ленту.

Поясним процесс обучения функции переходов `transitionFunction`. Обучающая выборка включала в себя все возможные ленты длины 10, состоящие из символов `Zero` и `One`, тестовая выборка состояла из 10 лент длины 100, содержащих случайные битовые последовательности. Процесс обучения состоял из последовательности шагов, на каждом производились следующие действия:

1) Из тестовой выборки брались 10 случайных лент.
2) На вход сети `machine` подавались сеть `transitionFunction`, ленты и стартовое состояние `Start`.
3) Вычислялся MSE результата сети и требуемого ответа.
4) Для результатов сети находились жесткие решения (см. пункт 3.3.2), и считалось, в скольких позициях они отличаются от требуемых.
5) Вычислялись градиенты и делался шаг в направлении уменьшения MSE.

Процесс проверки заключался в выполнении шагов с 1 по 4 на всей тестовой выборке.

Функция переходов `transitionRequired` представляет собой искомое решение задачи. Ее проверка показала, что полученные ответы сети не отличаются от требуемых. Это подтверждает корректность работы данной реализации машины Тьюринга, а также модулей, использованных в Runtime-библиотеке языка FNN.

Функции переходов `transitionSimple` и `transitionDifficult` содержат пропуски, которые требуется заполнить в процессе обучения. Графики зависимостей точности решений от номера итерации градиентного спуска приведены на Рисунке 5. Обученные сети были проверены на тестовой выборке, результаты приведены в Таблице 5.

Таблица 5 – Результаты обучения сети

Функция переходов	MSE	Количество ошибочных ответов
`transitionRequired`	0	0
`transitionSimple`	4,7511	0
`transitionDifficult`	8,8311	0

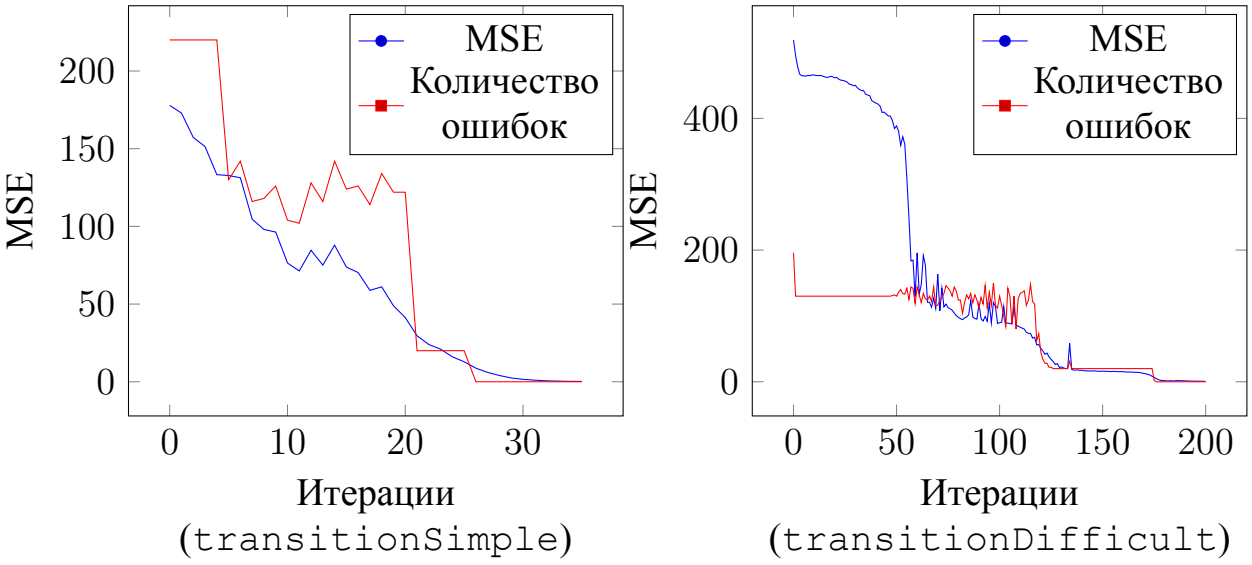

Рисунок 5 – Процесс обучения сети

Результат обучения показывает, что в обоих случаях алгоритм был выведен правильно - на тестовой выборке не было обнаружено отличий полученных жестких ответов от требуемых. В то же время видно, что время поиска алгоритма зависит от числа неизвестных параметров алгоритма. При фиксированных правилах перехода по ленте искомое решение было найдено намного быстрее, чем при поиске его с нуля. Однако искомый алгоритм был выведен корректно в обоих случаях.

Приведенный пример показывает, что с использованием FNN можно реализовать машину Тьюринга, функции переходов которой можно находить в процессе обучения по примерам. Это подтверждает Тьюринг-полноту разработанного языка.

3.3.5. Анализ результатов

Проведенные эксперименты показали, что с использованием разработанного подхода можно обучаться алгоритмам, причем для относительно простых алгоритмов решение находится за достаточно малое число итераций.

Подход позволяет применять жадное предобучение и переиспользовать алгоритмы. Сеть, обучающаяся умножению, использует в себе в себе сеть, вычисляющую сумму. В ходе эксперимента было опробовано два подхода к обучению: в первом обе сеть для умножения использовала предобученную сеть для сложения, во втором подходе они обучались одновременно. Было выявлено, что хоть решение и находилось в обоих случаях, предварительное обучение сети для сложения позволяет значительно ускорить вывод алгоритма умножения.

Графики зависимости точности решения от номера итераций содержат «ступени» - резкое изменение точности за малое количество шагов алгоритма. Причина появления данных участков состоит в следующем. Параметры слоев сети задают описание алгоритма, поэтому в процессе их настройки сеть перебирает алгоритмы решения задачи. Плавное изменение точности происходит, когда сеть подбирает параметры, описывающие одно решение. Ступень появляется в тот момент, когда сеть переходит к другому способу решения задачи, который обладает большей точностью.

Можно сказать, что локальные минимумы, которые сеть обходит в процессе обучения, являются приближенными алгоритмами решения задачи. Глобальный минимум в свою очередь является точным решением.

В процессе проведения экспериментов было также отмечено, что чем меньше пропусков содержит код программы, тем быстрее происходит обучение. В то же время чем больше пропуском содержит код, тем более широкий класс алгоритмов можно находить на основе данного наброска.

Также было подтверждена Тьюринг-полнота разработанного языка, что вместе с возможностью обучения позволяет выводить произвольные алгоритмы. Для этого можно использовать как реализованную на языке обучаемую машину Тьюринга, так и задавать наброски программ непосредственно на языке FNN. Первый подход позволяет находить структуру алгоритма в процессе обучения, но требует больше времени для сходимости. Второй метод требует задать структуру решения, но благодаря этому позволяет вывести алгоритм за меньшее количество итераций.

Выводы по главе 3

В данной главе были приведены определение и детали реализации функционального дифференцируемого языка программирования **FNN**, разработанного на основе проведенного теоретического исследования. Программы на данном языке компилируется в расширенные нейронные сети, при помощи которых можно производить индукцию алгоритмов.

Процесс компиляции программ состоит из ряда этапов, включающих в себя разбор исходного кода программ, вывод типа, компиляцию лямбда-выражений в расширенные нейронные сети и генерацию кода. Результатом компиляции является программа на языке **Python**, использующая разработанную библиотеку времени исполнения. Данная библиотека основана на фреймворке

PyTorch, поддерживающем процедуру автоматического дифференцирования с динамическим графом исполнения. Библиотека времени исполнения содержит определения типов данных (T-деревьев, O-деревьев) и слоев, требуемых для задания расширенной нейронной сети.

С использованием данного подхода был произведен вывод нескольких алгоритмов, обрабатывающих данные произвольной длины. Для этого были написаны наброски алгоритмов, содержащие пропуски, которые были заполнены в процессе обучения по сгенерированным наборам данных, чем была достигнута индукция требуемых алгоритмов. Результаты экспериментов показали, что используя расширенные нейронные сети, построенные по программам на языке **FNN**, можно производить вывод алгоритмов по входным данным за разумное время.

ЗАКЛЮЧЕНИЕ

В данной работе был представлен подход к решению задачи индукции алгоритмов при помощи нейронных сетей глубокого обучения. Предложенное решение состоит в проектировании сетей для конкретных задач на основе программ, написанных на функциональном дифференцируемом языке программирования **FNN**. Данный подход использует соответствие между лямбда-исчислением и нейронными сетями, его преимущество состоит в том, что при помощи нейронных сетей, построенных на основе программ, можно производить индукцию алгоритмов обработки структурированных данных.

В работе вводится определение расширенной нейронной сети, позволяющей изменять архитектуру сети на основе входных данных, определяются понятия T-деревьев и O-деревьев, позволяющих сетям обучаться алгоритмам работы со структурированными данными произвольного размера. Понятие полиморфизма, позволяющее обрабатывать данные разных типов при помощи одного кода, было перенесено на нейронные сети. В работе вводится понятие полиморфных нейронных сетей, которые способны выводить алгоритмы обработки данных произвольных типов.

Теоретические результаты данной работы были проверены практически. Для этого был проведен ряд экспериментов, в которых требовалось вывести ряд алгоритмов, таких как арифметические операции над числами произвольной длины, сравнение чисел и обработка списков произвольного типа. В результате экспериментов было показано, что сети, построенные с использованием разработанного подхода, способны обучаться рекурсивным алгоритмам при помощи метода градиентного спуска. Также на языке была реализована обучаемая машина Тьюринга, что подтверждает полноту разработанного языка и позволяет находить произвольные алгоритмы при помощи обучения по примерам.

В дальнейшем планируется развить как теоретические, так и практические результаты работы. В частности, для расширения возможностей полиморфизма планируется перенести понятие классов типов на нейронные сети. Компилятор языка **FNN** также будет доработан для повышения скорости компиляции и расширения функционала языка.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Коликов Д. А.*, *Потапов А. С.* Индукция алгоритмов с помощью Тьюринг-полных сетей глубокого обучения. — Сборник тезисов докладов конгресса молодых ученых., 2018.

2 *McCulloch W. S.*, *Pitts W.* A logical calculus of the ideas immanent in nervous activity // Bulletin of mathematical biology. — 1990. — Т. 52, № 1/2. — P. 99–115.

3 *Ясницкий Л. Н.* Введение в искусственный интеллект. — Академия, 2008.

4 *Goodfellow I.*, *Bengio Y.*, *Courville A.* Deep Learning. — MIT Press, 2016.

5 *Hochreiter S.*, *Schmidhuber J.* Long short-term memory // Neural computation. — 1997. — Т. 9, № 8. — P. 1735–1780.

6 *Siegelmann H. T.*, *Sontag E. D.* On the computational power of neural nets // Journal of computer and system sciences. — 1995. — Т. 50, № 1. — P. 132–150.

7 *Graves A.*, *Wayne G.*, *Danihelka I.* Neural Turing Machines // ArXiv e-prints. — 2014. — Окт. — arXiv: 1410.5401.

8 *DeepMind G.* Differentiable neural computers [Электронный ресурс]. — 2016. — URL: https://deepmind.com/blog/differentiable-neural-computers/ (дата обращения: 10.04.2018).

9 *Kaiser Ł.*, *Sutskever I.* Neural GPUs Learn Algorithms // ArXiv e-prints. — 2015. — Нояб. — arXiv: 1511.08228 [cs.LG].

10 *Price E.*, *Zaremba W.*, *Sutskever I.* Extensions and Limitations of the Neural GPU // ArXiv e-prints. — 2016. — Нояб. — arXiv: 1611.00736.

11 *Kant N.* Recent Advances in Neural Program Synthesis // ArXiv e-prints. — 2018. — Февр. — arXiv: 1802.02353 [cs.AI].

12 *Siegelmann H. T.* Neural programming language // AAAI. — 1994. — P. 877–882.

13 Adaptive Neural Compilation / R. Bunel [и др.] // ArXiv e-prints. — 2016. — Май. — arXiv: 1605.07969 [cs.AI].

14 *Reed S.*, *de Freitas N.* Neural Programmer-Interpreters // ArXiv e-prints. — 2015. — Нояб. — arXiv: 1511.06279 [cs.LG].

15 Programming with a Differentiable Forth Interpreter / M. Bošnjak [и др.] // ArXiv e-prints. — 2016. — Май. — arXiv: 1605.06640.

16 *Sørensen M. H.*, *Urzyczyn P.* Lectures on the Curry-Howard isomorphism. T. 149. — Elsevier, 2006.

17 *Olah C.* Neural Networks, Types, and Functional Programming [Электронный ресурс]. — 2015. — URL: http://colah.github.io/posts/2015-09-NN-Types-FP/ (дата обращения: 05.04.2018).

18 Automatic differentiation in PyTorch / A. Paszke [и др.]. — 2017.

19 *JetBrains*. Kotlin Programming Language Reference [Электронный ресурс]. — 2018. — URL: https://kotlinlang.org/docs/reference/ (дата обращения: 15.04.2018).

20 *Parr T.* The definitive ANTLR 4 reference. — Pragmatic Bookshelf, 2013.

21 Practical type inference for arbitrary-rank types / S. P. Jones [и др.] // Journal of functional programming. — 2007. — Т. 17, № 1. — P. 1–82.

22 *Cardelli L.* Basic polymorphic typechecking // Sci. Comput. Program. — 1987. — Т. 8, № 2. — P. 147–172.

ПРИЛОЖЕНИЯ

Приложение 1. Пример программы на языке FNN

Листинг 1.1 – Пример программы

```
-- Type Definition
@type Bool = True | False;
-- Pattern Matching
if True x y = x;
if False x y = y;
```

Листинг 1.2 – Результат компиляции

```
from .runtime.modules import ConstantLayer, VariableLayer,
    GuardedLayer
from .runtime.patterns import VarPattern, LitPattern,
    ConstructorPattern
from .runtime.types import TypeSpec, LitSpec, VarSpec

# Defined Types
Bool = TypeSpec(operands=[
    LitSpec(),
    LitSpec(),
])
# Defined Nets
True_net = ConstantLayer(type_spec=Bool, position=0)
False_net = ConstantLayer(type_spec=Bool, position=1)
if_x1_net = GuardedLayer(to_type=VarSpec('x1'), cases=[
    GuardedLayer.Case(
        ConstructorPattern(0, operands=[
            LitPattern(0),
            VarPattern(),
            VarPattern(),
        ]), VariableLayer.Data(0)
    ),
    GuardedLayer.Case(
        ConstructorPattern(0, operands=[
            LitPattern(1),
            VarPattern(),
            VarPattern(),
        ]), VariableLayer.Data(1)
    ),
])
```

Приложение 2. Исходный код для индукции арифметических операций

```
@type N = Z | S N;
@type Pair a b = MkPair a b;

-- Sketch of addition
@type plus = N -> N -> N;
plus Z y = @learn y;
plus (S x) y = @learn (plus x (@learn y));

-- Expected addition algorithm
plusRequired Z b = b;
plusRequired (S x) b = S (plusRequired x b);

-- Sketch of subtraction
@type minus = N -> N -> N;
minus x Z = @learn x;
minus x (S y) = @learn (minus (@learn x) y);

-- Expected subtraction algorithm
minusRequired Z (S y) = Z;
minusRequired x Z = x;
minusRequired (S x) (S y) = S (minusRequired x y);

-- ADT for the result of comparison
@type Cmp = Less | Equal | Greater;

-- Sketch of comparison
@type cmp = N -> N -> Cmp;
cmp x y = @learn x y;
cmp (S x) (S y) = @learn (cmp x y);

-- Expected comparison algorithm
cmpRequired Z Z = Equal;
cmpRequired (S x) Z = Greater;
cmpRequired Z (S y) = Less;
cmpRequired (S x) (S y) = cmpRequired x y;

-- Sketch of multiplication
@type mul = N -> N -> N;
mul Z y = @learn y;
mul (S x) y = plus (@learn (mul x (@learn y))) (@learn y);
```

```
-- Expected multiplication algorithm
mulRequired Z b = Z;
mulRequired (S x) b = plusRequired (mulRequired x b) b;
```

Приложение 3. Исходный код для индукции алгоритмов обработки списков

```
@type List a = Empty | Cons a (List a);
@type N = Z | S N;

@type Unit = MkUnit;
@type Bool = False | True;
@type Pair a b = MkPair a b;
@type Triple a b c = MkTriple a b c;
@type Maybe a = Just a | Nothing;

-- Sketch of concatenation algorithm
@type concat = List a -> List a -> List a;
concat Empty y = @learn y;
concat (Cons x rest) y = @learn x (concat rest y);

-- Reqired concatenation algorithm
concatRequired Empty y = y;
concatRequired (Cons x rest) y = Cons x (concatRequired rest y);

-- Sketch of indexation algorithm
getOrDefault n x Empty = @learn n x;
getOrDefault Z x y = @learn x y;
getOrDefault (S n) x (Cons y rest) = @learn (getOrDefault n (@learn
    x) rest);

-- Required indexation algorithm
getRequired n def Empty = def;
getRequired Z def (Cons x rest) = x;
getRequired (S n) def (Cons x rest) = getRequired n def rest;
```

Приложение 4. Исходный код обучаемой машины Тьюринга

```
-- Types for the tape
@type LList a = LEmpty | LCons (LList a) a;
@type RList a = RCons a (RList a) | REmpty;
@type Tape a = MkTape (LList a) a (RList a);

-- Types for commands of the machine
@type EvalState st = Stop | Continue st;
@type Move = Left | Stay | Right;
@type TransRes s a = MkRes (EvalState s) a Move;

-- Types for states and alphabet
@type State = Start;
@type Alphabet = Blank | Zero | One;

-- Tape movement
moveLeft (MkTape LEmpty x r) = MkTape LEmpty Blank (RCons x r);
moveLeft (MkTape (LCons l x) y r) = MkTape l x (RCons y r);

moveRight (MkTape l x REmpty) = MkTape (LCons l x) Blank REmpty;
moveRight (MkTape l x (RCons y r)) = MkTape (LCons l x) y r;

makeMove tape Stay = tape;
makeMove tape Left = moveLeft tape;
makeMove tape Right = moveRight tape;

-- The Turing machine
@type machine = (s -> a -> TransRes s a) -> Tape a -> s -> Tape a;
machine transition (MkTape l a r) s = @case (transition s a) @of
        (MkRes Stop y m) -> makeMove (MkTape l y r) m,
        (MkRes (Continue st) y m) ->
           machine transition (makeMove (MkTape l y r) m) st;

-- Required transition function
transitionRequired Start Blank = MkRes Stop Blank Stay;
transitionRequired Start Zero = MkRes (Continue Start) One Stay;
transitionRequired Start One = MkRes (Continue Start) Zero Stay;

-- Simple learning task
@type transitionSimple =
  State -> Alphabet -> TransRes State Alphabet;
transitionSimple Start Blank = MkRes Stop (@learn Blank) Stay;
```

```
transitionSimple Start Zero =
                    MkRes (Continue Start) (@learn Zero) Right;
transitionSimple Start One =
                    MkRes (Continue Start) (@learn One) Right;

-- Difficult learning task
@type transitionDifficult =
  State -> Alphabet -> TransRes State Alphabet;
transitionDifficult Start a = @learn{to_depth=2} a;
```