ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

Федеральное государственное образовательное бюджетное учреждение высшего профессионального образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ им. проф. М. А. БОНЧ-БРУЕВИЧА»

Ф.В. Филиппов

МОДЕЛИРОВАНИЕ НЕЙРОННЫХ СЕТЕЙ ГЛУБОКОГО ОБУЧЕНИЯ

УЧЕБНОЕ ПОСОБИЕ

СПбГУТ)))

САНКТ-ПЕТЕРБУРГ
2017

УДК 004.9(004.42)

Рецензенты
кандидат технических наук, доцент кафедры робототехники
и автоматизации производственных систем СПбГЭТУ «ЛЭТИ»
А.В. Шевченко
кандидат технических наук, доцент кафедры конструирования и
производства радиоэлектронных средств
Т.В. Матюхина

*Утверждено редакционно-издательским советом СПбГУТ
в качестве учебного пособия*

Филиппов, Ф.В.

Моделирование нейронных сетей глубокого обучения: учебное пособие / Ф. В. Филиппов; СПбГУТ, – СПб., 2017. – 84 с.

Рассматриваются практические аспекты использования программных пакетов в среде *R* и *RStudio* для моделирования нейронных сетей глубокого обучения.

Пособие предназначено для студентов направления 09.03.02 Информационные системы и технологии и будет полезно при изучении дисциплин «Технологии обработки информации», «Интеллектуальные системы и технологии» и «Интеллектуализация управления инфокоммуникационными системами и сетями».

УДК 004.9(004.42)

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Примерно пять последних лет очень богаты на события, связанные с нейросетевыми технологиями и машинным обучением. Концепция многослойных нейронных сетей стала доступной для практического использования. Это обусловлено заметным ростом производительности компьютеров и появлением эффективных алгоритмов обучения и оптимизации структуры многослойных сетей. Особенно заметный прогресс показали сверточные и рекуррентные сети, подходящие для решения задач в области компьютерного зрения и распознавания речи. Многие крупные компании выпустили в свет свои разработки - *Google* открытую библиотеку *TensorFlow*, ученые из *Microsoft* инструментарий *CNTK* для проектирования и тренировки сетей различного типа.

Появились такие мощные средства для исследования и проектирования нейронных сетей, как сервер H_2O, масштабируемая платформа *MxNet* и открытая библиотека *Keras*. Их важная особенность состоит в том, что они представляют собой дружественный *API,* нацеленный на возможность быстрого экспериментирования с моделями нейронных сетей глубокого обучения.

Естественно, специалисты в области информационных технологий должны быть знакомы с новейшими технологиями обработки данных с помощью перспективных нейросетевых архитектур и правильно оценивать назначение и возможности того или иного архитектурного решения.

Цель настоящего пособия состоит в том, чтобы ознакомить студентов с основами использования современных сред обработки информации для исследования систем искусственного интеллекта.

Язык *R* и среда *RStudio* являются свободно распространяемым программным обеспечением и занимают прочное место среди специалистов в области информационных технологий, занимающихся всесторонним анализом данных, в частности, исследованием нейронных систем различного назначения.

Основная задача пособия заключается в развитии у студентов практических навыков использования эффективных программных пакетов моделирования нейронных систем глубокого обучения в среде *RStudio*.

Пособие включает необходимый теоретический материал для подготовки к практическим занятиям. Для успешного освоения материала данного пособия рекомендуется знакомство с приемами моделирования классических нейронных сетей из [1]. Все сокращения, используемые в пособии расшифрованы в списке сокращений.

1. НЕЙРОННЫЕ СЕТИ ГЛУБОКОГО ОБУЧЕНИЯ

В настоящее время нейронные сети переживают "глубинную революцию", вызванную успешным применением методов *Deep Learning* (глубокого обучения), представляющих собой третье поколение нейронных сетей. В отличие от классических нейронных сетей, новые парадигмы обучения позволили избавиться от ряда проблем, которые сдерживали распространение и успешное применение нейронных сетей на практике.

Теория глубокого обучения дополняет обычные технологии машинного обучения специальными алгоритмами для анализа входной информации на нескольких уровнях представления. Особенность нового подхода заключается в том, что "глубокое обучение" изучает предмет, пока не найдет достаточно информативных уровней представления для учета всех факторов, способных повлиять на характеристики изучаемого предмета.

Таким образом, нейронная сеть на базе такого подхода требует меньше входной информации для обучения, а обученная сеть способна анализировать информацию с гораздо более высокой точностью, чем обычные нейронные сети.

Новая парадигма обучения реализует идею обучения в два этапа. На первом этапе из большого массива неразмеченных данных с помощью автоассоциаторов (путем их послойного обучения без учителя) извлекается информация о внутренней структуре входных данных. Затем, используя эту информацию в многослойной нейросети, ее обучают с учителем (размеченными данными) известными методами. При этом количество неразмеченных данных желательно иметь как можно большим, а размеченных данных может быть намного меньше.

1.1. Автоассоциаторы

Первым автоассоциатором был неокогнитрон Фукушимы. Его задача состоит в том, чтобы получить на выходе как можно более точное отображение входа. При этом, используются два вида автоассоциаторов — синтезирующие и генерирующие. В качестве первых используют автокодеры *AE*, а вторых — ограниченные машины Больцмана *RBM*.

Автокодер — это нейронная сеть с одним скрытым слоем, которая с помощью алгоритма обучения без учителя и метода обратного распространения ошибки устанавливает целевое значение y, равное входному вектору x. Структурная схема автокодера приведена на рис. 1.

Автокодер пытается построить функцию $h(x) = x$, т.е. найти аппроксимацию такой функции, чтобы отклик нейронной сети приблизительно равнялся значению входных признаков. Для того, чтобы решение этой задачи было нетривиальным количество нейронов скрытого слоя должно быть меньше, чем размерность входных данных (как на рисунке).

Это позволяет получить сжатие данных при передаче входного сигнала на выход сети. Например, если входной вектор представляет собой набор уровней яркости изображения 10x10 пикселов (всего 100 признаков), а количество нейронов скрытого слоя 50, сеть вынужденно обучается сжатию изображения. Требование $h(x) = x$ означает, что исходя из уровней активации пятидесяти нейронов скрытого слоя выходной слой должен восстановить 100 пикселей исходного изображения. Такая компрессия возможна, если в данных есть скрытые взаимосвязи, корреляция признаков, и вообще какая-то структура. В таком виде функционирование автокодера очень напоминает метод анализа главных компонент (*PCA*) в том смысле, что понижается размерность входных данных.

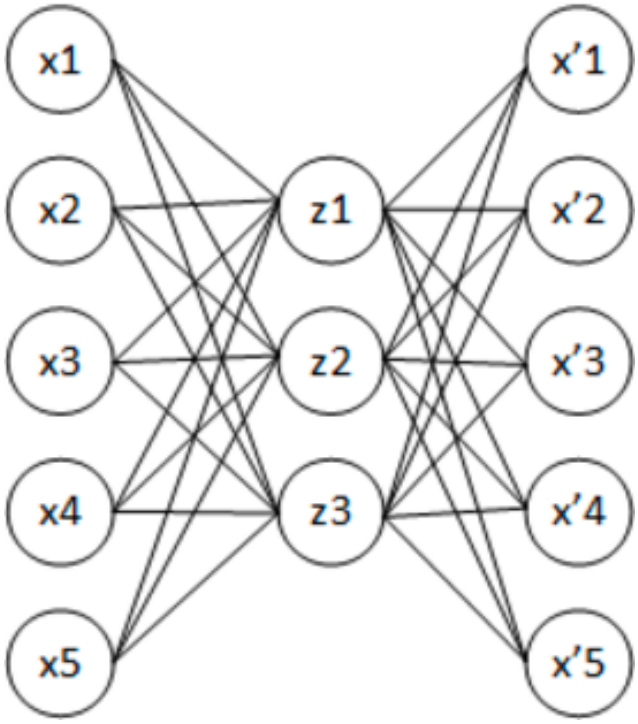

Рис. 1. Структурная схема автокодера

Позже с появлением идеи разрежения получил распространение разреженный автокодер (*sparse autoencoder*). У разреженного автокодера количество скрытых нейронов гораздо больше размерности входа, но они имеют разреженную активацию. Разреженная активация – это когда количество неактивных нейронов в скрытом слое значительно превышает количество активных. Если описывать разреженность неформально, то будем считать нейрон активным, когда значение его функции передачи близко к 1. Если используется сигмоидная функция передачи, то для неактивного нейрона ее значение должно быть близко к 0.

Существует вариант автокодирующего устройства, называемый шумоподавляющий (*denoising*) автокодер. Это автокодер, обучение которого спе-

цифично. При обучении на вход подают случайным образом "испорченные" данные. При этом для сравнения с выходом предъявляют "неиспорченные". Таким способом можно заставить автокодер восстанавливать поврежденные входные данные.

Ограниченная машина Больцмана отличается от обыкновенной отсутствием связей между нейронами одного слоя. На рис. 2 приведена структурная схема RBM.

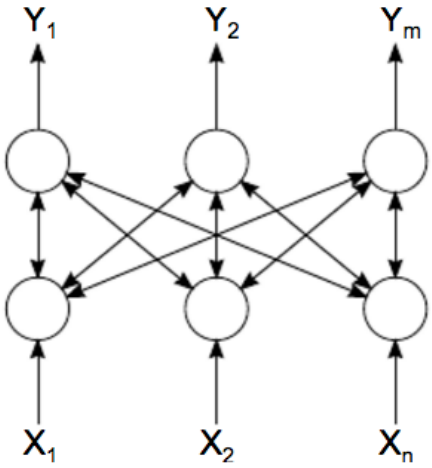

Рис. 2. Структурная схема *RBM*

Особенность модели *RBM* состоит в том в том, что при данном состоянии нейронов одной группы, состояния нейронов другой группы будут независимы друг от друга. У нас есть ряд состояний, которые мы можем наблюдать (видимые нейроны $X_1X_2...X_n$) и ряд состояний, которые скрыты, и мы не можем напрямую увидеть их состояние (скрытые нейроны $Y_1Y_2...Y_m$). Но мы можем сделать вероятностный вывод относительно скрытых состояний, опираясь на состояния, которые мы можем наблюдать. Обучив такую модель мы так же получаем возможность делать выводы относительно видимых состояний, зная скрытые, и тем самым генерировать данные из того вероятностного распределения, на котором обучена модель.

Таким образом, можно сформулировать цель обучения модели: необходимо настроить параметры модели так, чтобы восстановленный вектор из исходного состояния был наиболее близок к оригиналу. Под восстановленным понимается вектор, полученный вероятностным выводом из скрытых состояний, которые в свою очередь получены вероятностным выводом из видимых состояний, т.е. из оригинального вектора.

В качестве процедуры обучения *RBM* используется алгоритм контрастного расхождения *CD*. Этот алгоритм придуман профессором Хинтоном [2], он отличается своей простотой. Главная идея в том, что математические ожидания заменяются вполне определенными значениями. *K*-шаговая процедура реализации алгоритма *CD* представляет собой процесс сэмплирования:

1. состояние видимых нейронов приравнивается к входному образу;
2. выводятся вероятности состояний скрытого слоя;
3. каждому нейрону скрытого слоя ставится в соответствие состояние "1" с вероятностью, равной его текущему состоянию;
4. выводятся вероятности видимого слоя на основании скрытого слоя;
5. если текущая итерация меньше k, то возврат к шагу 2;
6. выводятся вероятности состояний скрытого слоя.

Чем дольше мы делаем сэмплинг, тем точнее будет обучение. В то же время профессор Хинтон утверждает, что даже для $k = 1$ (всего одна итерация сэмплинга) получается вполне хороший результат.

1.2. Накапливающие нейронные сети

Для извлечения из входного набора данных абстракций высокого уровня автоассоциаторы складывают в сеть. На рис. 3 приведена структурная схема накапливающего автокодера *SAE* и нейронной сети *MLP*, которые в совокупности и представляют собой глубокую нейронную сеть с инициализацией весов от *SAE*.

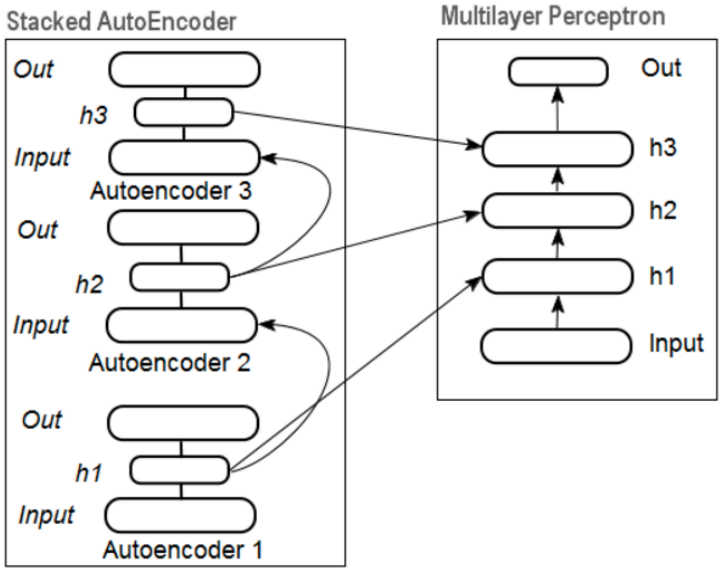

Рис. 3. Глубокая нейронная сеть с инициализацией весов от *SAE*

На рис. 4 приведена схема накапливающей ограниченной сети Больцмана *SRBM* и нейронной сети *MLP*, которые в совокупности представляют собой глубокую нейронную сеть с инициализацией весов от *SRBM*.

Схемы глубоких сетей изображают именно таким образом, подчеркивая, что информация извлекается снизу вверх.

Обучение глубоких сетей проводят в два этапа. На первом этапе послойно обучают без учителя на массиве не размеченных данных автоассоциативную сеть (*SAE* или *SRBM*), после чего полученными после обучения ве-

сами скрытых слоев автоассоциативной сети инициализируют нейроны скрытых слоев обычного многослойного персептрона *MLP*. На рис. 3 и рис. 4 схематично показан этот процесс обучения и переноса. После обучения первого *AE*/*RBM* веса нейронов скрытого слоя становятся входами второго и так далее. Тем самым из данных извлекается все более обобщающая информация о структуре (линия, контур, образ и т.д.).

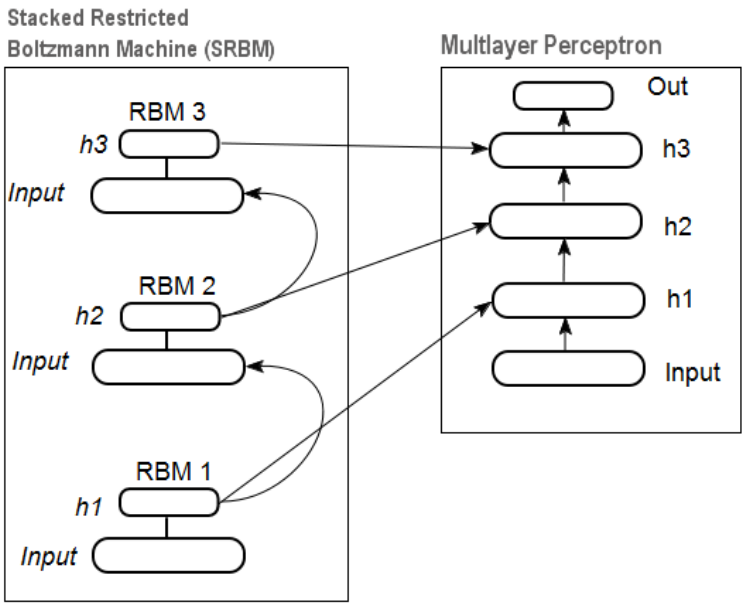

Рис. 4. Глубокая нейронная сеть с инициализацией весов от SRBM

На втором этапе происходит тонкая настройка *MLP* (обучение с учителем) на размеченном наборе данных общеизвестными методами. Практически доказано, что такая инициализация устанавливает веса нейронов скрытых слоев *MLP* в область глобального минимума и последующая тонкая настройка происходит за очень короткое время.

Кроме того для глубоких сетей с количеством слоев более трех профессор Хинтон предложил тонкую настройку производить также в два этапа. На первом обучать только два верхних слоя и только потом обучать всю сеть.

Необходимо отметить, что при обучении без учителя *SRBM* дает менее стабильные результаты, чем *SAE*.

1.3. Сверточные нейронные сети

Архитектура свёрточных нейронных сетей *CNN* делает явное предположение вида «входные данные есть изображения», что позволяет использовать эту специфику. Схематично *CNN* — это последовательность слоев и для ее организации применяется три основных слоя:

1. слой свёртки (convolution);
2. слой пулинга (pool) или субдискретизации (subsampling);
3. полносвязный слой (full connection).

Эти слои используются с целью построения полной архитектуры CNN рис. 5. Кроме того, добавляются блоки линейной ректификации ReLU, которые реализуют функцию активации нейронов $f(x) = max(0,x)$.

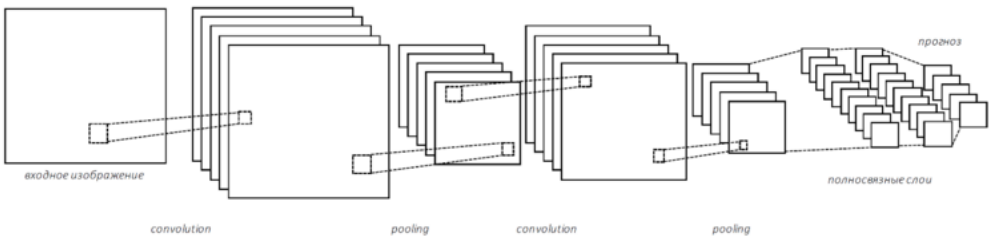

Рис. 5. Классическая сверточная нейронная сеть *LeNet*

Структурная схема, изображенная на рис. 5 разработана Яном Лекуном и применялась на практике для считывания почтовых индексов, цифр и т. п. С 2010 года ведётся проект ILSVRC, в рамках которого различные программные продукты ежегодно соревнуются в классификации и распознавании объектов. Ниже приводятся наиболее популярные архитектуры свёрточных нейронных сетей:

- *AlexNet* сверточная нейронная Алекса Крижевского, Ильи Суцкевера и Джеффа Хинтона. Она на *ILSVRC* 2012 обошла все работы конкурентов (16% ошибок против 26% у архитектуры, занявшей второе место).
- *ZF Net* – свёрточная нейронная сеть Мэтью Зеллера и Роба Фергюса победитель *ILSVRC* 2013, ее архитектура была улучшенной версией *AlexNex*.
- *GoogLeNet* – свёрточная нейронная сеть сотрудников корпорации *Google* победитель *ILSVRC* 2014. Основная заслуга данной архитектуры состоит в разработке и внедрении входного модуля (*Inception Module*), что позволило резко сократить число параметров с 60 до 4 миллионов.
- *ResNet* – остаточная сеть (*Residual Network),* разработанная Каймингом Хе и другими, стала победителем *ILSVRC* 2015. В ней была использована пакетная нормализация и специальные скип-соединения, кроме того, в конце архитектуры отсутствуют полносвязные слои.

С результатами последних соревнований *ILSVRC* 2017 можно ознакомиться в [3]. Следует отметить, что современные нейронные сети представляют из себя громадные архитектуры и требуют больших вычислительных ресурсов. В частности один из последних пакетов *mxnet*, предназначенный для моделирования нейронных сетей глубокого обучения, ориентирован в основном на использование кластеров с мощными *GPU*.

1.4. Рекуррентные нейронные сети

Особенность сетей *RNN* состоит в том, что нейроны получают информацию не только от предыдущего слоя, но и от самих себя с предыдущего прохода. Это означает, что порядок, в котором следует подавать данные при обучении сети становится важным. Большой сложностью сетей *RNN* является проблема исчезающего (или взрывного) градиента, которая заключается в быстрой потере информации с течением времени. Конечно, это влияет лишь на веса связей, а не состояния нейронов, но ведь именно в них накапливается информация.

В классических рекуррентных нейронных сетях Джордана и Элмана главной идеей было обучение своему выходному сигналу на предыдущем шаге [1]. Современные *RNN* стали применять для решении практических задач по распознаванию речи, анализу текстов, видео, машинному переводу и ряда других проблем. Это задачи, при решении которых мы сталкиваемся не просто с отдельными объектами, а с их последовательностями и порядок следования объектов играет существенную роль.

Обычные *RNN* сталкиваются с проблемой долговременной зависимости когда разрыв между актуальной информацией и точкой ее применения может стать достаточно большим и по мере роста этого расстояния, сеть теряет способность связывать информацию.

Чтобы избежать проблемы долговременной зависимости были разработаны *LSTM*, способные запоминать информацию на долгие периоды времени. На рис. 6 дано схематическое представление функционирования одного нейрона долгой краткосрочной памяти.

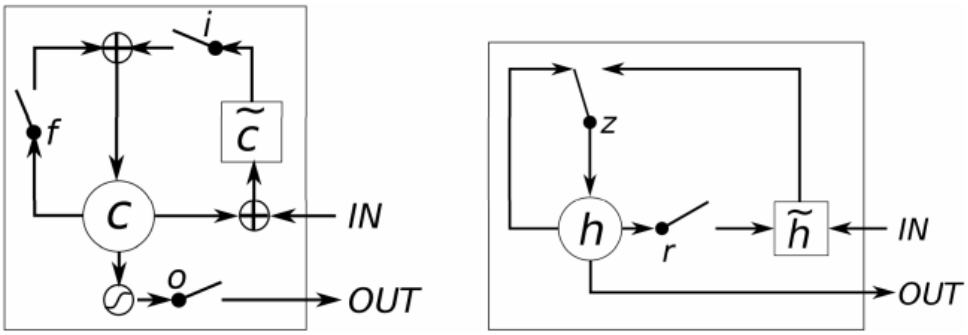

Рис. 6. Схематическое представление нейрона *LSTM* (слева) и *GRU*

В работе нейрона *LSTM* следует понимать три ключевых момента, связанных с обработкой информации. Они реализованы на основе специальных фильтров, которые определяют:

- какую часть хранимой информации следует забыть (фильтр *f-forget*);
- какую часть входной информации следует сохранить (фильтр *i-input*);
- какую часть информации следует отправить на выход (фильтр *o-output*).

Указанные части информации вычисляются с помощью соответствующих фильтров *f-forget*, *i-input* и *o-output* (рис. 6).

Существенным образом от стандартных *LSTM* отличаются управляемые рекуррентные модули *GRU* (рис. 6). В *GRU* фильтры забывания *f-forget* и входа *i-input* объединены в один фильтр обновления. Кроме того, состояние ячейки *c* объединено со скрытым состоянием *h (hidden)*. В результате модель *GRU* проще, чем стандартная *LSTM* и популярность ее неуклонно растет.

1.5. Подготовка входных и целевых данных

При подготовке входных и целевых данных для работы с моделями нейронных сетей глубокого обучения установился определенный порядок работы:

- выбор исходных данных (изучение, анализ, предподготовка, оценка), разбивка на обучающий и тестовый наборы (выборки);
- обучение и выбор модели/моделей на обучающей выборке;
- оценка качества модели/моделей на тестовой выборке и определение лучшей модели из набора по конкретным метрикам;
- передача модели/моделей в работу.

Первый пункт наиболее трудоемок, но и наиболее важен для конечного результата. Получить количественные оценки входного набора для выбора наиболее важных переменных, а еще лучше осуществить автоматический выбор лучших переменных для конкретной модели просто необходимо. Среда *R* предоставляет нам достаточно возможностей для решения перечисленных задач на всех этих этапах.

Реализацию глубоких сетей удобно осуществлять и изучать на языке R, который предоставляет шесть основных пакетов:

1. *deepnet* – простой пакет, реализующий модель глубокой нейросети *SAE* и глубокой нейросети *SRBM* [4].
2. *darh* – очень развитый и широкий пакет моделирования для глубокой нейросети *SRBM/SAE* [5]. Этот пакет позволяет создать и настроить модель любой сложности, он построен на базе оригинальных программ Хинтона.
3. *H2O* – очень серьезный пакет, предназначен для обучения моделей глубоких сетей (и не только) на "больших наборах данных" (>1 Гб) записанных в *csv*-файлах [6].
4. *mxnet* – гибкая и мощная среда реализующая алгоритмы глубокого обучения и имеющая интеграцию с *R*. Позволяет максимально эффективно задействовать аппаратные средства (она работает и на *CPU*, и на *GPU*, на одной рабочей станции и их кластере) [7].

5. *tensorflow* – пакет обеспечивает доступ к полному *API TensorFlow* из среды *R* [8].
6. *Keras* –высокоуровневый *API* разработанный с упором на возможность быстрого экспериментирования с моделями нейронных сетей [32].

2. ПАКЕТ *DEEPNET*

В пакете реализован ряд архитектур и алгоритмов нейронной сети глубокого обучения, включая *RBM, DBN* и *SAE*. Это относительно небольшой, но достаточно мощный пакет с разнообразными архитектурами. Он позволяет обучать сеть передачи данных с помощью функции *nn.train()* и инициализировать веса сети глубокого доверия с помощью функции *dbn.dnn.train()*. Функция *nn.train()* внутренне использует функцию *rbm.train()* для обучения ограниченной машины Больцмана, которую также можно использовать индивидуально. Кроме того, пакет *deepnet* позволяет моделировать накапливающие автокодеры с помощью функции *sae.dnn.train()*.

Ниже перечислен полный состав пакета и указано краткое назначение функций:

- *nn.train* – обучение однослойной и многослойной нейронной сети методом *backprop*;
- *nn.test* – тестирование обученной нейронной сети с возвращением значения ошибки;
- *nn.predict* – предсказание новых выборок для обученной нейронной сети;
- *dbn.dnn.train* – обучение сети с архитектурой *DBN*;
- *rbm.train* – обучение сети *RBM*;
- *rbm.down* – генерирование видимого вектора для скрытых состояний сети *RBM*;
- *rbm.up* – извлечение скрытых состояний *RBM* по известным видимым состояниям;
- *sae.dnn.train* – обучение нейронной сети с архитектурой *stacked autoencoder*.

Перейдем к описанию перечисленных функций пакета.

2.1. Функция *nn.train()*

Функция обучает один или несколько скрытых слоев нейронной сети методом обратного распространения ошибки.

Вызов функции (указаны значения аргументов, задаваемые по умолчанию):

```
nn.train(x, y, initW = NULL, initB = NULL, hidden = c(10), activationfun = "sigm",
```

```
learningrate = 0.8, momentum = 0.5, learningrate_scale = 1, output = "sigm",
numepochs = 3, batchsize = 100, hidden_dropout = 0, visible_dropout = 0)
```

Аргументы функции:

x – матрица входных данных (образцов);
y – вектор или матрица (выходных) целевых значений;
hidden – вектор с числом нейронов в каждом скрытом слое;
actvationfun – функция активации нейронов скрытого слоя (*sigm, tanh*);
l*earningrate* – скорость обучения для градиентного спуска;
momentum – импульс для градиентного спуска;
learningrate_scale – коэффициент изменения скорости обучения (скорость обучения будет умножаться на этот коэффициент после каждой итерации);
numepochs – число итераций для обучения;
batchsize – размер минимального количества образцов;
output – функции активации нейронов выходного слоя (*sigm, linear* или *softmax*);
hidden_dropout – удаляемая часть для скрытых слоев;
visible_dropout – удаляемая часть для входного слоя.

Пример 1

```
Var1 <- c(rnorm(50, 1, 0.5), rnorm(50, -0.6, 0.2))
Var2 <- c(rnorm(50, -0.8, 0.2), rnorm(50, 2, 1))
x <- matrix(c(Var1, Var2), nrow = 100, ncol = 2)
y <- c(rep(1, 50), rep(0, 50))
nn <- nn.train(x, y, hidden = c(5))
```

2.2. Функция *nn.test()*

Функция тестирует новые образцы на обученной нейронной сети и возвращает процент ошибок классификации.

Вызов функции:

```
nn.test(nn, x, y, t = 0.5)
```

Аргументы функции:

nn – нейронная сеть обученная с помощью функции *nn.train()*;
x – новые входные образцы для предсказания выхода;
y – новые выходные метки;
t – порог для классификации: если значение *nn.predict* >= *t* то метка = 1, в противном случае метка = 0.

Для лучшего понимания работы функции вызовем и проанализируем ее содержимое:

```
> nn.test
  function (nn, x, y, t = 0.5)
  {
    y_p <- nn.predict(nn, x)
    m <- nrow(x)
    y_p[y_p >= t] <- 1
    y_p[y_p < t] <- 0
    error_count <- sum(abs(y_p - y))/2
    error_count/m
  }
```

В вектор *y_p* помещаются значения предсказанные с помощью функции *nn.predict(nn,x)*, которые затем классифицируются по результатам сравнения с порогом. В переменой *err_count* сначала подсчитывается число несовпадений, которое затем делится на длину вектора для определения процента ошибок.

Пример 2

```
Var1 <- c(rnorm(50, 1, 0.5), rnorm(50, -0.6, 0.2))
Var2 <- c(rnorm(50, -0.8, 0.2), rnorm(50, 2, 1))
x <- matrix(c(Var1, Var2), nrow = 100, ncol = 2)
y <- c(rep(1, 50), rep(0, 50))
nn <- nn.train(x, y, hidden = c(5))
test_Var1 <- c(rnorm(50, 1, 0.5), rnorm(50, -0.6, 0.2))
test_Var2 <- c(rnorm(50, -0.8, 0.2), rnorm(50, 2, 1))
test_x <- matrix(c(test_Var1, test_Var2), nrow = 100, ncol = 2)
err <- nn.test(nn, test_x, y)
```

2.3. Функция *nn.predict()*

Функция служит для предсказания новых выходных значений для обученной нейронной сети. Она возвращает необработанное выходное значение нейронной сети.

Вызов функции:

```
nn.predict(nn, x)
```

Аргументы функции:

nn – нейронная сеть обученная с помощью функции *nn.train()*;
x – новые входные образцы для предсказания выхода.

Пример 5

```
# обучаем нейронную сеть nn
Var1 <- c(rnorm(50, 1, 0.5), rnorm(50, -0.6, 0.2))
Var2 <- c(rnorm(50, -0.8, 0.2), rnorm(50, 2, 1))
x <- matrix(c(Var1, Var2), nrow = 100, ncol = 2)
y <- c(rep(1, 50), rep(0, 50))
nn <- nn.train(x, y, hidden = c(10),activationfun = "tanh")
## предсказываем выходные значения
test_Var1 <- c(rnorm(50, 1, 0.5), rnorm(50, -0.6, 0.2))
test_Var2 <- c(rnorm(50, -0.8, 0.2), rnorm(50, 2, 1))
test_x <- matrix(c(test_Var1, test_Var2), nrow = 100, ncol = 2)
yy <- nn.predict(nn, test_x)
```

2.4. Функция *dbn.dnn.train()*

Функция служит для обучения нейронной сети глубокого обучения с весами, инициализированными *DBN*.

Вызов функции (указаны значения аргументов, задаваемые по умолчанию):

```
dbn.dnn.train(x, y, hidden = c(1), activationfun = "sigm", learningrate = 0.8,
    momentum = 0.5, learningrate_scale = 1, output = "sigm", numepochs = 3,
    batchsize = 100, hidden_dropout = 0, visible_dropout = 0, cd = 1)
```

Аргументы функции:

x – матрица входных данных (образцов);
y – вектор или матрица целевых значений;
hidden – вектор с числом нейронов в каждом скрытом слое;
actvationfun – функция активации нейронов скрытого слоя (*sigm, tanh*);
l*earningrate* – скорость обучения для градиентного спуска;
momentum – импульс для градиентного спуска;
learningrate_scale – коэффициент изменения скорости обучения (скорость обучения будет умножаться на этот коэффициент после каждой итерации);
numepochs – число итераций для обучения;
batchsize – размер минимального количества образцов;
output – функции активации нейронов выходного слоя (*sigm, linear, tanh*);
hidden_dropout – удаляемая часть для скрытых слоев;
visible_dropout – удаляемая часть для входного слоя;
cd – число итераций для образца Гиббса алгоритма *CD*.

Рассмотрим пример использования функции для обучения распознаванию различных пар случайных чисел, образованных векторами *Var1* и *Var2*.

Пары значений в диапазонах: 1±0.5, -0.8±0.2 должны отождествляться с целевым значением 1, а пары значений в диапазонах: -0.6±0.2, 2±1 должны отождествляться с целевым значением 0. Для обучения используем по 50 образцов входных данных. Указанные пары значений сформированы из векторов *Var1* и *Var2* в матрице *x*, а целевые значения в векторе *y*:

Пример 3

```
Var1 <- c(rnorm(50, 1, 0.5), rnorm(50, -0.6, 0.2))
Var2 <- c(rnorm(50, -0.8, 0.2), rnorm(50, 2, 1))
x <- matrix(c(Var1, Var2), nrow = 100, ncol = 2)
y <- c(rep(1, 50), rep(0, 50))
dnn <- dbn.dnn.train(x, y, hidden = c(10))
```

Проверим качество обучения:

Пример 4

```
test_Var1 <- c(rnorm(50, 1, 0.5), rnorm(50, -0.6, 0.2))
test_Var2 <- c(rnorm(50, -0.8, 0.2), rnorm(50, 2, 1))
test_x <- matrix(c(test_Var1, test_Var2), nrow = 100, ncol = 2)
error <- nn.test(dnn, test_x, y)
```

2.5. Функция *rbm.train()*

Функция служит для обучения *RBM* – ограниченной машины Больцмана. *RBM* – это «мелкие» двухслойные нейронные сети, которые составляют строительные блоки сетей глубокого обучения (рис. 7). Первый уровень *RBM* называется видимым или входным *v* (*visible*), а второй – скрытым *h* (*hidden*).

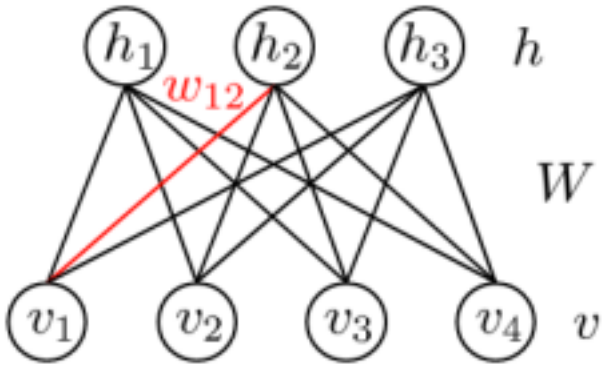

Рис. 7. Модель ограниченной машины Больцмана (*RBM*)

Цель обучения модели *RBM*: необходимо настроить параметры модели *W* так, чтобы восстановленный вектор *h* из исходного состояния *v* был наиболее близок к оригиналу. Под восстановленным понимается вектор, по-

лученный вероятностным выводом из скрытых состояний, которые в свою очередь получены вероятностным выводом из обозреваемых состояний, т.е. из оригинального вектора.

Вызов функции:

```
rbm.train(x, hidden, numepochs = 3, batchsize = 100, learningrate = 0.8,
        learningrate_scale = 1, momentum = 0.5, visible_type = "bin",
        hidden_type = "bin", cd = 1)
```

Аргументы функции:

x – матрица входных данных (образцов);
hidden – вектор с числом нейронов в каждом скрытом слое;
numepochs – число итераций для обучения;
batchsize – размер минимального количества образцов;
l*earningrate* – скорость обучения для градиентного спуска;
learningrate_scale – коэффициент изменения скорости обучения;
momentum – импульс для градиентного спуска;
visible_type – функция активации нейронов входного слоя (*sigm*);
hidden_type – функция активации нейронов скрытого слоя (*sigm*);
cd – число итераций для образца Гиббса алгоритма *CD*.

Пример 6

```
Var1 <- c(rep(1, 50), rep(0, 50))
Var2 <- c(rep(0, 50), rep(1, 50))
x3 <- matrix(c(Var1, Var2), nrow = 100, ncol = 2)
r1 <- rbm.train(x3, 10, numepochs = 20, cd = 10)
```

2.6. Функция *rbm.up()*

Функция формирует состояния скрытых нейронов по состояниям видимых нейронов.

Вызов функции:

```
rbm.up(rbm, v)
```

Аргументы функции:

rbm – объект *RBM* обученный с помощью функции *train.rbm()*;
v – состояния видимых нейронов.

Пример 7

```
Var1 <- c(rep(1, 50), rep(0, 50))
```

```
Var2 <- c(rep(0, 50), rep(1, 50))
 x <- matrix(c(Var1, Var2), nrow = 100, ncol = 2)
r1 <- rbm.train(x3, 3, numepochs = 20, cd = 10)
v <- c(0.2, 0.8)
h <- rbm.up(r1, v)
```

2.7. Функция *rbm.down()*

Функция формирует состояния видимых нейронов по состояниям скрытых нейронов.

Вызов функции:

```
rbm.up(rbm, h)
```

Аргументы функции:

rbm – объект *RBM* обученный с помощью функции *train.rbm*;
h – состояния скрытых нейронов.

Пример 8

```
Var1 <- c(rep(1, 50), rep(0, 50))
Var2 <- c(rep(0, 50), rep(1, 50))
x3 <- matrix(c(Var1, Var2), nrow = 100, ncol = 2)
r1 <- rbm.train(x3, 3, numepochs = 20, cd = 10)
h <- c(0.2, 0.8, 0.1)
v <- rbm.down(r1, h)
```

2.8. Функция *sae.dnn.train()*

Функция служит для обучения нейронной сети глубокого обучения с весами инициализированными накапливающим автокодером.

Вызов функции:

```
sae.dnn.train(x, y, hidden = c(1), activationfun = "sigm", learningrate = 0.8,
        momentum = 0.5, learningrate_scale = 1, output = "sigm",
        sae_output = "linear", numepochs = 3, batchsize = 100,
        hidden_dropout = 0, visible_dropout = 0)
```

Аргументы функции:

x – матрица входных данных (образцов);
y – вектор или матрица целевых значений;
hidden – вектор с числом нейронов в каждом скрытом слое;
actvationfun – функция активации нейронов скрытого слоя (*sigm, tanh*);

l*earningrate* – скорость обучения для градиентного спуска;

momentum – импульс для градиентного спуска;

learningrate_scale – коэффициент изменения скорости обучения;

numepochs – число итераций для обучения;

batchsize – размер минимального количества образцов;

output – функция активации нейронов выходного слоя (*sigm, linear, tanh*);

sae_output – функция активации выходного слоя автокодера (*sigm, linear, softmax*);

hidden_dropout – удаляемая часть для скрытых слоев;

visible_dropout – удаляемая часть для входного слоя;

Пример 9

```
Var1 <- c(rnorm(50, 1, 0.5), rnorm(50, -0.6, 0.2))
Var2 <- c(rnorm(50, -0.8, 0.2), rnorm(50, 2, 1))
x <- matrix(c(Var1, Var2), nrow = 100, ncol = 2)
y <- c(rep(1, 50), rep(0, 50))
dnn <- sae.dnn.train(x, y, hidden = c(5, 5))
## проверка dnn
test_Var1 <- c(rnorm(50, 1, 0.5), rnorm(50, -0.6, 0.2))
test_Var2 <- c(rnorm(50, -0.8, 0.2), rnorm(50, 2, 1))
test_x <- matrix(c(test_Var1, test_Var2), nrow = 100, ncol = 2)
nn.test(dnn, test_x, y)
```

Приведем еще один пример использования этой функции для обучения распознаванию цифр классического набора MNIST:

Пример 10

```
library(chron)
library(data.table)
mnist.train <- as.matrix(fread("mnist_train.csv", header=F))
x  <-  mnist.train[, 1:784]/255
y  <-  model.matrix(~as.factor(mnist.train[, 785])-1)
nn <-  dbn.dnn.train(x, y, hidden=c(64), output="softmax",
                     batchsize=128,  numepochs=100,  learningrate = 0.1)
```

3. ПАКЕТ *DARCH*

Пакет *darch* предназначен для создания многослойных нейронных сетей (глубокие архитектуры) и их обучению по методу, представленному публикациями [9,10]. Этот метод включает предварительную подготовку с использованием процедуры *contrastive divergence* (контрастного расхождения) [11] и дальнейшей тонкой настройкой с использованием общеизвестных алгоритмов обучения, таких как *backpropagation* (обратное распространение) или *conjugate gradients* (сопряженные градиенты). Кроме того, тонкая настройка может быть усовершенствована с помощью методов *maxout* и *dropout*.

Пакет *darch* предоставляет широкий круг функций, позволяющих не просто создать и обучить модель, но буквально по кирпичикам сложить и настроить ее под любые предпочтения. Сеть глубокого обучения можно строить из *n RBM* в виде *MLP* с количеством слоев *n-1*. Послойное предобучение *RBM* производится на неразмеченных данных без учителя, а дальнейшее тонкое обучение нейросети производится с учителем на размеченных данных, при этом есть доступ ко всем внутренним параметрам.

3.1. Функция *darch()*

Функция *darch()*, одноименная с пакетом, снабжена огромным числом аргументов и служит для формирования нейронной сети глубокого обучения с дополнительным предварительным обучением и одним из множества алгоритмов тонкой настройки.

Вызов функции (указаны значения аргументов, задаваемые по умолчанию):

```
darch(x, y, layers = 10, ..., autosave = F,
    autosave.epochs = round(darch.numEpochs/20),
    autosave.dir = "./darch.autosave", autosave.trim = F, bp.learnRate = 1,
    bp.learnRateScale = 1, bootstrap = F, bootstrap.unique = T,
    bootstrap.num = 0, cg.length = 2, cg.switchLayers = 1, darch = NULL,
    darch.batchSize = 1, darch.dither = F, darch.dropout = 0,
    darch.dropout.dropConnect = F, darch.dropout.momentMatching = 0,
    darch.dropout.oneMaskPerEpoch = F, darch.elu.alpha = 1,
    darch.errorFunction = if (darch.isClass) crossEntropyError else mseError,
    darch.finalMomentum = 0.9, darch.fineTuneFunction = backpropagation,
    darch.initialMomentum = 0.5, darch.isClass = T,
    darch.maxout.poolSize = 2, darch.maxout.unitFunction = linearUnit,
    darch.momentumRampLength = 1, darch.nesterovMomentum = T,
    darch.numEpochs = 100, darch.returnBestModel = T,
    darch.returnBestModel.validationErrorFactor = 1 - exp(-1),
    darch.stopClassErr = -Inf, darch.stopErr = -Inf,
    darch.stopValidClassErr = -Inf, darch.stopValidErr = -Inf,
    darch.trainLayers = T, darch.unitFunction = sigmoidUnit,
```

```
darch.weightDecay = 0,
darch.weightUpdateFunction = weightDecayWeightUpdate, dataSet = NULL,
dataSetValid = NULL,
generateWeightsFunction = generateWeightsGlorotUniform, gputools = F,
gputools.deviceId = 0, logLevel = NULL, normalizeWeights = F,
normalizeWeightsBound = 15, paramsList = list(),
preProc.factorToNumeric = F, preProc.factorToNumeric.targets = F,
preProc.fullRank = T, preProc.fullRank.targets = F,
preProc.orderedToFactor.targets = T, preProc.params = F,
preProc.targets = F, rbm.allData = F, rbm.batchSize = 1,
rbm.consecutive = T, rbm.errorFunction = mseError,
rbm.finalMomentum = 0.9, rbm.initialMomentum = 0.5, rbm.lastLayer = 0,
rbm.learnRate = 1, rbm.learnRateScale = 1, rbm.momentumRampLength = 1,
rbm.numCD = 1, rbm.numEpochs = 0, rbm.unitFunction = sigmoidUnitRbm,
rbm.updateFunction = rbmUpdate, rbm.weightDecay = 2e-04, retainData = F,
rprop.decFact = 0.5, rprop.incFact = 1.2, rprop.initDelta = 1/80,
rprop.maxDelta = 50, rprop.method = "iRprop+", rprop.minDelta = 1e-06,
seed = NULL, shuffleTrainData = T, weights.max = 0.1,
weights.mean = 0, weights.min = -0.1, weights.sd = 0.01,
xValid = NULL, yValid = NULL)
```

Аргументы функции:

x – матрица или фрейм входных данных;

y – матрица или фрейм целевых значений, соответственно;

layers – вектор, определяющий количество нейронов в каждом скрытом слое;

autosave – логическое указание на то, следует ли автоматически сохранить экземпляр сети *DArch* в файл во время точной настройки;

autosave.epochs – указывает число эпох (итераций) после которого происходит автоматическое сохранение экземпляра;

autosave.dir – директорий для автосохранения файлов, имена файлов будут формироваться как, например, *Autosave_010.net* для экземпляра *DArch* после 10 эпох;

autosave.trim – нужно ли обрезать сеть перед ее сохранением (обрезание приведет к удалению набора данных и весов слоев, что приведет к созданию сети, которая больше не пригодна для прогнозирования или обучения);

bp.learnRate – скорость обучения для *backpropagation*, либо одна для всех слоев, либо вектор при использовании разных скоростей обучения для каждого слоя;

bp.learnRateScale – коэффициент изменения скорости обучения (скорость обучения будет умножаться на этот коэффициент после каждой итерации);

bootstrap – логическое указание на то, следует ли использовать начальную загрузку для создания данных обучения и проверки;

bootstrap.unique – логическое указание на то, следует ли брать только уникальные образцы для обучения или взять все образцы (это игнорируется, если *bootstrap.num* > 0);

bootstrap.num – указывает количество учебных образцов для начальной загрузки;

cg.length – numbers of line search;

cg.switchLayers – указывает, когда тренировать всю сеть вместо двух верхних слоев;

darch – существующий экземпляр *DArch*, для которого обучение должно быть возобновлено (при предварительном обучении предыдущие результаты обучения теряются);

darch.batchSize – размер партии, то есть количество обучающих образцов, которые используются до обновления веса при тонкой настройке;

darch.dither – нужно ли применять сглаживание к числовым столбцам в исходных данных обучения;

darch.dropout – вектор скорости отсева нейронов для каждого отдельного слоя (при включении *darch.dropout.dropConnect* этот вектор нуждается в дополнительном элементе (один элемент на матрицу весов между двумя слоями);

darch.dropout.dropConnect – использовать *DropConnect* вместо отсева для скрытых слоев (будет использовать *darch.dropout* в качестве скоростей отсева связей);

darch.dropout.momentMatching – количество итераций, которое должно быть выполнено до момента отсева, 0 для отключения согласования момента;

darch.dropout.oneMaskPerEpoch – нужно создавать новую маску для каждой партии (по умолчанию) или для каждой эпохи;

darch.elu.alpha – альфа-параметр для экспоненциальной линейной функции, смотри описание функции *exponentialLinearUnit()*;

darch.errorFunction – функция ошибки во время точной настройки: возможные функции ошибки включают *mseError()*, *rmseError()* и *crossEntropyError()*;

darch.finalMomentum – окончательный импульс при тонкой настройке;

darch.fineTuneFunction – функция точной настройки: возможные значения включают *backpropagation()*, *rpropagation()*, *minimizeClassifier()* и *minimizeAutoencoder()* (без учителя);

darch.initialMomentum – начальный импульс при тонкой настройке;

darch.isClass – должен ли выход во время точной настройки трактоваться в качестве метки класса и выводится скорость классификации;

darch.maxout.poolSize – размер пула для модулей *maxout* при использовании функции активации *maxout*, смотри функцию *maxoutUnit()*;

darch.maxout.unitFunction – функция внутреннего блока, используемая модулем maxout, смотри функцию *darch.unitFunction()*;

darch.momentumRampLength – указывает как скоро, по сравнению с общим количеством прошедших эпох, импульс должен достигнуть *darch.finalMomentum*. Значение 1 указывает на то, что *darch.finalMomentum*

должен быть достигнут в заключительную эпоху, значение 0.5 указывает, что *darch.finalMomentum* должен быть достигнут после завершения половины тренировки. Обратите внимание, что это приведет к несоответствиям в изменениях значения импульса, если обучение возобновится с теми же параметрами для *darch.initialMomentum* и *darch.finalMomentum*. Установите аргумент в 0, чтобы избежать этой проблемы при возобновлении обучения;

darch.nesterovMomentum – использовать Нестеровский ускоренный импульс на основе градиентного спуска для алгоритмов тонкой настройки [12];

darch.numEpochs – количество эпох тонкой настройки;

darch.returnBestModel – следует ли возвращать лучшую модель в конце обучения, вместо последней;

darch.returnBestModel.validationErrorFactor – определяет насколько высоко должна быть оценена ошибка проверки, по сравнению с ошибкой обучения при оценке моделей с данными проверки. Это значение от 0 до 1 (по умолчанию $1 - e^{-1}$). Коэффициент ошибки обучения и коэффициент ошибки проверки всегда будут добавляться к 1, поэтому, если задать здесь 1, ошибка обучения будет проигнорирована, а если задать 0, ошибка проверки будет проигнорирована;

darch.stopClassErr – когда ошибка классификации ниже или равна этому значению, обучение останавливается;

darch.stopErr – когда значение функции ошибки меньше или равно этому значению, обучение прекращается;

darch.stopValidClassErr – когда ошибка классификации данных ниже или равна этого значения, обучение останавливается;

darch.stopValidErr – когда значение функции ошибки в данных проверки ниже или равна этому значению, обучение прекращается;

darch.trainLayers – или *TRUE* для обучения всех слоев или маска, содержащая *TRUE* для слоев, которые необходимо обучить, и *FALSE* для слоев, которые не должны быть обучены (нет записи для входного слоя);

darch.unitFunction – послойная функция или вектор функций активации длины (число слоев – 1). Обратите внимание, что первая запись означает функцию активации на выход слоя 2. Слой 1 не имеет функции активации, так как входные значения используются напрямую. Возможные функции активации: *linearUnit()*, *sigmoidUnit()*, *tanhUnit()*, *rectifiedLinearUnit()*, *softplusUnit()*, *softmaxUnit()* и *maxoutUnit()*;

darch.weightDecay – коэффициент распада веса по умолчанию равен 0. Все веса будут умножаться на (1 – *darch.weightDecay*) до каждого обновления веса;

darch.weightUpdateFunction – функция обновления веса или вектор функций обновления веса, очень похожие на *darch.unitFunction*. Возможные функции обновления веса включают *weightDecayWeightUpdate* и *maxoutWeightUpdate*. Обратите внимание, что *maxoutWeightUpdate* должен использоваться после функции активации *maxout()*;

dataSet – экземпляр *DataSet*, переданный из функции *darch.DataSet()*, может быть указан вручную;

dataSetValid – экземпляр *DataSet*, содержащий данные проверки;

generateWeightsFunction – функция генерации веса или вектор функций генерации веса для слоев длины (количество уровней – 1). Возможные функции генерации веса: *generateWeightsUniform()*, *generateWeightsNormal()*, *generateWeightsGlorotNormal()*, *generateWeightsGlorotUnifo()*, *generateWeightsHeNormal*() и *generateWeightsHeUniform()*;

gputools – следует ли использовать *GPU* для умножения матриц, если они доступны;

gputools.deviceId – целое число, указывающее, какое устройство *GPU* следует использовать для умножения матрицы, смотри функцию *chooseGpu()*;

logLevel – использует установленный в настоящий момент уровень журнала по умолчанию, который является *futile.logger :: flog.info*, если он не был изменен. Другие доступные уровни включают, от наименьших до самых подробных: *FATAL*, *ERROR*, *WARN*, *DEBUG* и *TRACE*;

normalizeWeights – следует ли нормализовать вес (норма *L2* = *normalizeWeightsBound*);

normalizeWeightsBound – верхняя граница нормы *L2* входящих весовых векторов используется только в том случае, если аргумент *normalizeWeights* имеет значение *TRUE*;

paramsList – список параметров, может включать любые параметры, перечисленные выше, применяется для удобства или для использования в скриптах;

preProc.factorToNumeric – следует ли преобразовывать все факторы в числовые значения;

preProc.factorToNumeric.targets – должны ли все факторы быть преобразованы в числовые значения в целевых данных;

preProc.fullRank – использовать кодировку полного ранга, смотри [13];

preProc.fullRank.targets – использовать кодировку полного ранга для целевых данных [13];

preProc.orderedToFactor.targets – нужно ли преобразовывать упорядоченные факторы в целевых данных в неупорядоченные факторы (упорядоченные факторы преобразуются в числовые значения с помощью функции *dummyVars()* [14] и больше не могут использоваться для задач классификации);

preProc.params – список параметров для перехода к функции *preProcess* для входных данных или *FALSE*, чтобы отключить предварительную обработку входных данных;

preProc.targets – необходимо ли центрировать и масштабировать целевые данные, в отличие от *preProc.params*, это просто включение или выключение предварительной обработки целевых данных;

rbm.allData – следует ли использовать данные обучения и проверки для предварительной подготовки;

rbm.batchSize – размер партии, то есть количество обучающих образцов для предварительной подготовки;

rbm.consecutive – следует ли обучать каждую *RBM* по очереди при каждой попытке *rbm.numEpochs* или поочередно тренировать каждую *RBM* целиком (*FALSE*);

rbm.errorFunction – функция ошибки во время предварительной подготовки. Она используется только для оценки ошибки *RBM* и не влияет на само обучение. Возможные функции ошибки включают *mseError()* и *rmseError()*;

rbm.finalMomentum – окончательный импульс во время предварительной подготовки;

rbm.initialMomentum – начальный импульс во время предварительной подготовки;

rbm.lastLayer – число указывающее, на каком слое остановить предварительную подготовку. Возможные значения включают 0, что означает, что все слои обучены, положительные целые числа, что означает прекратить обучение после *RBM* сформировавшей видимый слой *rbm.lastLayer*, отрицательные целые числа, что означает прекратить обучение на *RBM* уровня *rbm.lastLayer*, начиная с верхней *RBM*;

rbm.learnRate – скорость обучения во время предварительной подготовки;

rbm.learnRateScale – скорость обучения *rbm.learnRate* будет умножаться на этот коэффициент после каждой эпохи (итерации);

rbm.momentumRampLength – указывает как скоро, по сравнению с общим количеством эпох *rbm.numEpochs*, импульс должен достигнуть rbm.finalMomentum. Значение 1 указывает на то, что *rbm.finalMomentum* должен быть достигнут в заключительную эпоху, значение 0.5 указывает, что *rbm.finalMomentum* должен быть достигнут после завершения половины тренировки;

rbm.numCD – количество полных шагов, для которых выполняется алгоритм *CD*. Увеличение этого значения значительно замедлит обучение;

rbm.numEpochs – количество предварительных тренировок. Примечание: при указани здесь значения, отличного от 0, а также при использовании существующего экземпляра *DArch* в параметре *darch*, веса сети будут полностью сброшены. Предварительная подготовка по существу является формой расширенной инициализации веса, и нет смысла выполнять предварительную подготовку в ранее обученной сети;

rbm.unitFunction – функция активации нейрона во время предварительной подготовки. Возможные функции: *sigmoidUnitRbm, tanhUnitRbm* и *linearUnitRbm*;

rbm.updateFunction – функция обновления во время предварительной подготовки. В настоящее время darch предоставляет только rbmUpdate;

rbm.weightDecay – коэффициент распада веса по умолчанию равен *2e-04*. Все веса будут умножаться на (1 – *rbm.weightDecay*) до каждого обновления веса;

retainData – логическое указание на то, следует ли хранить данные обучения в экземпляре *DArch* после обучения или при сохранении на диске;

rprop.decFact – снижающий коэффициент для обучения;

rprop.incFact – повышаающий коэффициент для обучения;
rprop.initDelta – значение инициализации для обновления;
rprop.maxDelta – верхняя граница для размера шага;
rprop.method – метод обучения;
rprop.minDelta – нижняя граница для размера шага;
seed – позволяет специфицировать исходное значение, которое будет установлено через *set.seed*. Используется в контексте функции *darchBench()*;
shuffleTrainData – логическое указание на то, следует ли перетасовывать данные обучения перед каждой эпохой;
weights.max – параметр *max* для функции *runif*;
weights.mean – среднее значение mean для функции *rnorm*;
weights.min – параметр *min* для функции *runif*;
weights.sd – параметр *sd* в функцию *rnorm*;
xValid – матрица или *data.frame* входных данных для проверки;
yValid – матрица или *data.frame* целевых данных для проверки;
data – data.frame, содержащий набор данных, если *x* – формула.

Пример 11

```
data(iris)
model <- darch(Species ~ ., iris)
print(model)
predictions <- predict(model, newdata = iris, type = "class")
cat(paste("Incorrect classifications:", sum(predictions != iris[,5])))
```

Пример 12

```
trainData <- matrix(c(0,0,0,1,1,0,1,1), ncol = 2, byrow = TRUE)
trainTargets <- matrix(c(0,1,1,0), nrow = 4)
model2 <- darch(trainData, trainTargets, layers = c(2, 10, 1),
darch.numEpochs = 500, darch.stopClassErr = 0, retainData = T)
e <- darchTest(model2)
cat(paste0("Incorrect classifications on all examples: ", e[3], " (", e[2], "%)\n"))
plot(model2)
```

Много примеров использования функции *darch()* можно найти в [15].

3.2. Функция *backpropagation()*

Функция реализует алгоритм обратного распространения ошибки для сети глубокой архитектуры. Единственными специфическими для пользователя параметрами для алгоритма обратного распространения, являются *bp.learnRate* и bp.learnRateScale, они могут быть переданы в функцию *darch()* при включении *backpropagation* в качестве функции точной настройки. Аргумент *bp.learnRate* определяет скорость обучения *backpropagation* и может

быть задан как одиночный скаляр или вектор, определяющий скорости обучения на каждом уровне. Аргумент *bp.learnRateScale* – это единственный скаляр, определяющий коэффициент масштабирования для скоростей обучения, который будет применяться после каждой эпохи.

Алгоритм *backpropagation* поддерживает выпадение (*dropout*) и использует функцию обновления веса, определенную с помощью параметра *darch.weightUpdateFunction* для функции *darch()*.

Вызов функции (указаны значения аргументов, задаваемые по умолчанию):

```
backpropagation(darch, trainData, targetData,
    bp.learnRate = getParameter(".bp.learnRate", rep(1, times =length(darch@layers))),
    bp.learnRateScale = getParameter(".bp.learnRateScale"),
    nesterovMomentum = getParameter(".darch.nesterovMomentum"),
    dropout = getParameter(".darch.dropout", rep(0, times = length(darch@layers)
    + 1), darch), dropConnect = getParameter(".darch.dropout.dropConnect"),
    matMult = getParameter(".matMult"), debugMode = getParameter(".debug", F),
    ...)
```

Аргументы функции:

darch – экземпляр класса *DArch*;
trainData – данные для обучения (входные данные);
targetData – целевые данные (выходы);
bp.learnRate – скорости обучения для *backpropagation*, либо одна для всех уровней, либо вектор разных скоростей обучения для каждого слоя;
bp.learnRateScale – скорость обучения умножается на это значение после каждой эпохи;
nesterovMomentum – смотри параметр *darch.nesterovMomentum* функции *darch()*;
dropout – смотри параметр *darch.dropout* функции *darch()*;
dropConnect – смотри параметр *darch.dropout.dropConnect* функции *darch()*;
matMult – функция умножения матрицы, внутренний параметр;
debugMode – включен ли режим отладки, внутренний параметр;
... – дополнительные параметры.

Пример 13

```
data(iris)
model <- darch(Species ~ ., iris, darch.fineTuneFunction = "backpropagation")
```

Другими функциями тонкой настройки могут служить: *rpropagation()*, *minimizeAutoencoder()* и *minimizeClassifier()*.

3.3. Функция *rpropagation()*

Функция обрабатывает глубокую архитектуру с помощью алгоритма упругого обратного распространения. Она может использовать четыре разных метода обучения.

Алгоритм *Rprop* поддерживает выпадение и использует функцию обновления веса, определенную параметром *darch.weightUpdateFunction* для функции *darch()*. Возможные методы обучения (параметр *rprop.method*) следующие:

Rprop+: *Rprop* с возвратом веса;
Rprop-: *Rprop* без потери веса;
iRprop+: улучшенный *Rprop* с возвратом веса;
iRprop-: улучшенный *Rprop* без потери веса.

Подробные сведения об отказоустойчивом алгоритме обратного распространения приведены в [21,22].

Вызов функции:

```
rpropagation(darch, trainData, targetData,
    rprop.method = getParameter(".rprop.method"),
    rprop.decFact = getParameter(".rprop.decFact"),
    rprop.incFact = getParameter(".rprop.incFact"),
    rprop.initDelta = getParameter(".rprop.initDelta"),
    rprop.minDelta = getParameter(".rprop.minDelta"),
    rprop.maxDelta = getParameter(".rprop.maxDelta"),
    nesterovMomentum = getParameter(".darch.nesterovMomentum"),
    dropout = getParameter(".darch.dropout"),
    dropConnect = getParameter(".darch.dropout.dropConnect"),
    errorFunction = getParameter(".darch.errorFunction"),
    matMult = getParameter(".matMult"), debugMode = getParameter(".debug", F,
    darch), ...)
```

Аргументы функции:

darch – глубокая архитектура для обучения;
trainData – данные для обучения (входные данные);
targetData – целевые данные (выходы);
rprop.method – метод обучения, по умолчанию *iRprop+*;
rprop.decFact – понижающий коэффициент для обучения, значение по умолчанию – 0.6;
rprop.incFact – повышающий коэффициент для обучения, значение по умолчанию – 1.2;
rprop.initDelta – значение инициализации для обновления, по умолчанию 0.0125;

rprop.minDelta – нижняя граница для размера шага, по умолчанию 0.000001;

rprop.maxDelta – верхняя граница для размера шага, по умолчанию 50;

nesterovMomentum – аналогичен параметру *darch.nesterovMomentum* в *darch()*;

dropout – аналогичен параметру *darch.dropout* в *darch()*;

dropConnect – аналогичен параметру *darch.dropout.dropConnect* в *darch()*;

errorFunction – аналогичен параметру *darch.errorFunction* в *darch()*;

matMult – функция умножения матрицы, внутренний параметр;

debugMode – включен ли режим отладки, внутренний параметр;

. . . – дополнительные параметры.

Пример 14

```
data(iris)
 model <- darch(Species ~ ., iris, darch.fineTuneFunction = "rpropagation",
  preProc.params = list(method = c("center", "scale")),
  darch.unitFunction = c("softplusUnit", "softmaxUnit"),
  rprop.method = "iRprop+", rprop.decFact = .5, rprop.incFact = 1.2,
  rprop.initDelta = 1/100, rprop.minDelta = 1/1000000, rprop.maxDelta = 50)
```

3.4. Функция *minimizeAutoencoder()*

Функция обучает автокодер с помощью метода сопряженного градиента. Она поддерживает выпадение, но не использует функцию обновления веса, определенную с помощью параметра *darch.weightUpdate-Function* для *darch()*, так что распад веса, импульс и т. д. не поддерживаются.

В результате формируется обученный *DArch* объект.

Вызов функции:

```
minimizeAutoencoder(darch, trainData, targetData,
    cg.length = getParameter(".cg.length"),
    dropout = getParameter(".darch.dropout"),
    dropConnect = getParameter(".darch.dropout.dropConnect"),
    matMult = getParameter(".matMult"), debugMode = getParameter(".debug"),
    ...)
```

Аргументы функции:

darch – экземпляр класса *DArch*;

trainData – матрица данных для обучения (входные данные);

targetData – метки для обучающих данных (выходы);

cg.length – количество строк поиска;

dropout – смотри параметр *darch.dropout* функции *darch()*;

dropConnect – смотри параметр *darch.dropout.dropConnect* функции *darch()*;

matMult – функция умножения матрицы, внутренний параметр;

debugMode – включен ли режим отладки, внутренний параметр;

... – дополнительные параметры.

Пример 15

```
data(iris)
  model <- darch(Species ~ ., iris, c(6,10,2,10,6), darch.isClass = F,
   preProc.params = list(method=c("center", "scale")),
   darch.numEpochs = 20, darch.batchSize = 6, darch.unitFunction = tanhUnit
   darch.fineTuneFunction = "minimizeAutoencoder")
```

3.5. Функция *minimizeClassifier()*

Функция обучает классификатор с помощью метода сопряженного градиента. Она реализует тонкую настройку для сети классификации с обратным распространением. Параметр cg.switchLayers предназначен для переключения между двумя типами обучения. Верхние два слоя могут обучаться в одиночку, пока эпоха не будет равна epochSwitch. После этого будет проведена подготовка всей сети.

Функция поддерживает выпадение, но не использует функцию обновления веса, определенную с помощью параметра *darch.weightUpdateFunction* для *darch()*, так что распад, импульс и т. д. не поддерживаются.

В результате формируется обученный *DArch* объект.

Вызов функции:

```
minimizeClassifier(darch, trainData, targetData,
    cg.length = getParameter(".cg.length"),
    cg.switchLayers = getParameter(".cg.length"),
    dropout = getParameter(".darch.dropout"),
    dropConnect = getParameter(".darch.dropout.dropConnect"),
    matMult = getParameter(".matMult"), debugMode = getParameter(".debug"),
    ...)
```

Аргументы функции:

darch – экземпляр класса *DArch*;

trainData – матрица данных для обучения (входные данные);

targetData – метки для обучающих данных (выходы);

cg.length – количество строк поиска;

cg.switchLayers – указывает, когда обучается вся сеть вместо двух верхних слоев;

dropout – смотри параметр *darch.dropout* функции *darch()*;

dropConnect – смотри параметр *darch.dropout.dropConnect* функции *darch()*;

matMult – функция умножения матрицы, внутренний параметр;

debugMode – включен ли режим отладки, внутренний параметр;

... – дополнительные параметры.

Пример 16

```
data(iris)
   model <- darch(Species ~ ., iris,
    preProc.params = list(method = c("center", "scale")),
    darch.unitFunction = c("sigmoidUnit", "softmaxUnit"),
    darch.fineTuneFunction = "minimizeClassifier",
    cg.length = 3, cg.switchLayers = 5)
```

3.6. Функция *crossEntropyError()*

Функция вычисляет ошибку кросс-энтропии на основе исходных и оценочных параметров. Она может использоваться для параметра *darch.errorFunction* функции *darch()*, но является только допустимой функцией ошибки, если используется в сочетании с функцией активации *softmaxUnit()*.

Вызов функции:

```
crossEntropyError(original, estimate)
```

Аргументы функции:

original – оригинальная матрица данных;
estimate – расчетная матрица данных.

Пример 17

```
data(iris)
model <- darch(Species ~ ., iris, darch.errorFunction = "crossEntropyError")
```

3.7. Функция *mseError()*

Функция вычисляет среднеквадратическую ошибку *MSE* на основе исходных и оценочных параметров. Эта функция является допустимым значением для обоих параметров *darch()*: *rbm.errorFunction* и *darch.errorFunction*.

Вызов функции:

```
mseError(original, estimate)
```

Аргументы функции:

original – оригинальная матрица данных;
estimate – расчетная матрица данных.

В результате работы формируется список с именем функции ошибки в первой записи и значением ошибки во второй записи.

Пример 18

```
data(iris)
model <- darch(Species ~ ., iris, rbm.errorFunction = "mseError",
   darch.errorFunction = "mseError")
```

3.8. Функция *rmseError()*

Функция вычисляет квадратный корень из среднеквадратической ошибки *RMSE* на основе исходных и оценочных параметров. Эта функция является допустимым значением для обоих параметров *darch()*: *rbm.errorFunction* и *darch.errorFunction*.

Вызов функции:

```
rmseError(original, estimate)
```

Аргументы функции:

original – оригинальная матрица данных;
estimate – расчетная матрица данных.

В результате работы формируется список с именем функции ошибки в первой записи и значением ошибки во второй записи.

Пример 19

```
data(iris)
model <- darch(Species ~ ., iris, rbm.errorFunction = "rmseError",
   darch.errorFunction = "rmseError")
```

3.9. Функция *darchBench()*

Функция *darchBench()* – это обычная функция *benchmark(),* которая применяется для функции *darch()* пользователями, которые не могут или не хотят использовать стандартный пакет *rbenchmark* для оценки временных затрат [16]. Для работы этой функции необходимо, чтобы был загружен пакет *foreach*.

Вызов функции:

```
darchBench(..., bench.times = 1, bench.save = F,
    bench.dir = "./darch.benchmark", bench.continue = T, bench.delete = F,
    bench.seeds = NULL, output.capture = bench.save, logLevel = NULL)
```

Аргументы функции:

... – параметры функции *darch()*;

bench.times – количество выполняемых проверок;

bench.save – следует ли сохранять результаты в каталоге;

bench.dir – относительный или абсолютный путь, включающий каталог, где сохраняются результаты проверок, если значение *bench.save* истинно;

bench.continue – следует ли добавлять результаты к уже существующим результатам в каталоге, указанном в файле *bench.dir*;

bench.delete – удалять ли содержимое *bench.dir*, если *bench.continue* – *FALSE* (при задании этого параметра TRUE будут удалены ВСЕ файлы в данном каталоге);

bench.seeds – вектор предустановки, по одному значению для каждого теста (будет передан в *darch*);

output.capture – собирать ли *R* выход в *.Rout* файлы в данном каталоге. Это единственный способ получить доступ к выходу *R*, поскольку цикл *foreach* ничего не печатает на консоли. Этот параметр будет проигнорирован, если для *bench.save* установлено значение *FALSE*;

logLevel – futile.logger log level. Использует установленный в настоящий момент уровень журнала по умолчанию, который является *futile.logger :: flog.info*, если он не был изменен. Другие доступные уровни включают в себя, от наименьших до самых подробных: *FATAL, ERROR, WARN, DEBUG* и *TRACE*.

В качестве выходных значений выдает список экземпляров *DArch* и результаты каждого обращения к *darch*. Другие функции интерфейса *darch*: *darchTest, plot.DArch, pred.DArch* и *print.DArch*.

Пример 20

```
data(iris)
modelList <- darchBench(Species ~ ., iris, c(0, 50, 0),
    preProc.params = list(method = c("center", "scale")),
    darch.unitFunction = c("sigmoidUnit", "softmaxUnit"),
    darch.numEpochs = 30, bench.times = 10, bench.save = T)
```

3.10. Функция *darchModelInfo()*

Эта функция создает описание модели, позволяющее обучать экземпляры *DArch* с помощью функции *train*. Более подробно с назначением и возможностями использования функции можно ознакомиться в [17].

Вызов функции:

```
darchModelInfo(params = NULL, grid = NULL)
```

Аргументы функции:

params – data.frame параметров или *NULL* для использования значения по умолчанию (*bp.learnRate*);

grid – функция, которая выдает файл *data.frame*, содержащий сетку комбинаций параметров или *NULL*, для использования значения по умолчанию.

Пример 21

```
data(iris)
   tc <- trainControl(method = "boot", number = 5, allowParallel = F,
     verboseIter = T)
   parameters <- data.frame(parameter = c("layers", "bp.learnRate",
"darch.unitFunction"),
     class = c("character", "numeric", "character"),
     label = c("Network structure", "Learning rate", "unitFunction"))
   grid <- function(x, y, len = NULL, search = "grid")
   {
     df <- expand.grid(layers = c("c(0,20,0)","c(0,10,10,0)","c(0,10,5,5,0)"),
      bp.learnRate = c(1,2,5,10))
     df[["darch.unitFunction"]] <- rep(c("c(tanhUnit, softmaxUnit)",
      "c(tanhUnit, tanhUnit, softmaxUnit)",
      "c(tanhUnit, tanhUnit, tanhUnit, softmaxUnit)"), 4)
df }

   caretModel <- train(Species ~ ., data = iris, tuneLength = 12, trControl = tc,
     method = darchModelInfo(parameters, grid), preProc = c("center", "scale"),
     darch.numEpochs = 15, darch.batchSize = 6, testing = T, ...)
```

3.11. Функция *darchTest()*

Функция пересылает данные в реальном времени через глубокую нейронную сеть и оценивает точность классификации с использованием указанных меток.

Вызов функции:

```
darchTest(darch, newdata = NULL, targets = T)
```

Аргументы функции:

darch – экземпляр объекта *DArch*;
newdata – новые данные для использования, *NULL* – используются данные обучения;
target – этикетки для данных, *NULL* – используются обучающие метки (возможно только при *newdata = NULL*).

Это удобная функция, аналогичная прогнозу *predict.DArch()*. Она оценивает эффективность классификации и возвращает список индикаторов точности (необработанная ошибка сети, процент неправильных классификаций и абсолютное число неправильных классификаций).

Пример 22

```
data(iris)
model <- darch(Species ~ ., iris, retainData = T)
classificationStats <- darchTest(model)
```

3.12. Функция *generateWeightsNormal()*

Эта функция реализует стандартную процедуру для генерации случайных весов для экземпляров сети. Она использует функцию *rnorm()*.

Вызов функции:

```
generateWeightsNormal(numUnits1, numUnits2,
    weights.mean = getParameter(".weights.mean", 0, ...),
    weights.sd = getParameter(".weights.sd", 0.01, ...), ...)
```

Аргументы функции:

numUnits1 – количество нейронов в нижнем слое;

numUnits2 – количество нейронов в верхнем слое;
weights.mean – среднее значение для функции *rnorm*;
weights.sd – sd параметр для функции *rnorm*;
. . . – дополнительные параметры, используемые для формирования весов.

В результате работы функции формируется матрица весов.

Пример 23

```
data(iris)
model <- darch(Species ~ ., iris, generateWeightsFunction = "generateWeightsNormal",
   weights.mean = .1, weights.sd = .05)
```

3.13. Функция *generateWeightsUniform()*

Эта функция используется для генерации случайных весов и смещений с использованием функции *runif()*.

Вызов функции:

```
generateWeightsUniform(numUnits1, numUnits2,
    weights.min = getParameter(".weights.min", -0.1, ...),
    weights.max = getParameter(".weights.max", 0.1, ...), ...)
```

Аргументы функции:

numUnits1 – количество нейронов в нижнем слое;
numUnits2 – количество нейронов в верхнем слое;
weights.min – min значение для функции *runif()*;
weights.max – max значение для функции *runif()*;
. . . – дополнительные параметры, используемые для формирования весов.

В результате работы функции формируется матрица весов.

Пример 24

```
data(iris)
model <- darch(Species ~ ., iris, generateWeightsFunction = "generateWeightsUniform",
   weights.min = -.1, weights.max = .5)
```

3.14. Функция *generateWeightsGlorotNormal()*

Эта функция используется для генерации случайных весов и смещений с использованием *Glorot* нормальной инициализации весов [18].

Вызов функции:

```
generateWeightsGlorotNormal(numUnits1, numUnits2,
    weights.mean = getParameter(".weights.mean", 0, ...), ...)
```

Аргументы функции:

numUnits1 – количество нейронов в нижнем слое;
numUnits2 – количество нейронов в верхнем слое;
weights.mean – средний параметр для функции *rnorm*;
. . . – дополнительные параметры, используемые для формирования весов, передаваемые в *generateWeightsNormal*.

В результате работы функции формируется матрица весов.

Пример 25

```
data(iris)
model <- darch(Species ~ ., iris, generateWeightsFunction =
"generateWeightsGlorotNormal",    weights.mean = .1)
```

Другие функции генерации весов: *generateWeightsGlorotUniform, generateWeightsHe-Normal, generateWeightsHeUniform, generateWeightsNormal* и *generateWeights-Uniform*.

3.15. Функция *generateWeightsGlorotUniform()*

Эта функция используется для генерации случайных весов и смещений с использованием *Glorot* равномерной инициализации весов [18].

Вызов функции:

```
generateWeightsGlorotUniform(numUnits1, numUnits2, ...)
```

Аргументы функции:

numUnits1 – количество нейронов в нижнем слое;
numUnits2 – количество нейронов в верхнем слое;
. . . – дополнительные параметры, используемые для формирования весов, передаваемые в *generateWeightsUniform*.

В результате работы функции формируется матрица весов.

Пример 26

```
data(iris)
model <- darch(Species ~ ., iris, generateWeightsFunction =
"generateWeightsGlorotUniform")
```

3.16. Функция *generateWeightsHeNormal()*

Эта функция используется для генерации случайных весов и смещений с использованием *He* нормальной инициализации весов [19].

Вызов функции:

```
generateWeightsHeNormal(numUnits1, numUnits2,
    weights.mean = getParameter(".weights.mean", 0, ...), ...)
```

Аргументы функции:

numUnits1 – количество нейронов в нижнем слое;
numUnits2 – количество нейронов в верхнем слое;
weights.mean – средний параметр для функции *rnorm*;
. . . – дополнительные параметры, используемые для формирования весов, передаваемые в *generateWeightsNormal*.

В результате работы функции формируется матрица весов.

Пример 27

```
data(iris)
model <- darch(Species ~ ., iris, generateWeightsFunction =
"generateWeightsHeNormal",    weights.mean = .1)
```

3.17. Функция *generateWeightsHeUniform()*

Эта функция используется для генерации случайных весов и смещений с использованием *He* равномерной инициализации весов [19].

Вызов функции:

```
generateWeightsHeUniform(numUnits1, numUnits2, ...)
```

Аргументы функции:

numUnits1 – количество нейронов в нижнем слое;
numUnits2 – количество нейронов в верхнем слое;
. . . – дополнительные параметры, используемые для формирования весов, передаваемые в *generateWeightsUniform*.

В результате работы функции формируется матрица весов.

Пример 28

```
data(iris)
model <- darch(Species ~ ., iris, generateWeightsFunction =
"generateWeightsHeUniform")
```

3.18. Функция *linearUnit()*

Функция вычисляет активацию нейронов и возвращает список, в котором первая запись представляет собой линейную активацию нейронов, а вторая запись является производной от передаточной функции.

Вызов функции:

```
linearUnit(input, ...)
```

Аргументы функции:

input – вход для функции активации;
... – дополнительные параметры.

Пример 29

```
data(iris)
model <- darch(Species ~ ., iris, darch.unitFunction = "linearUnit")
```

Другие возможные функции активации модуля *darch*: exponential*LinearUnit, maxoutUnit, rectifiedLinearUnit, sigmoidUnit, softmaxUnit, softplusUnit* и *tanhUnit*.

3.19. Функция exponentialLinearUnit()

Функция вычисляет активацию нейронов и возвращает список, в котором первая запись представляет собой экспоненциальную линейную активацию (*ELU*) нейронов, а вторая запись является производной от передаточной функции.

Вызов функции:

```
exponentialLinearUnit(input, alpha = getParameter(".darch.elu.alpha", 1, ...), ...)
```

Аргументы функции:

input – вход для функции активации;
alpha – ELU гиперпараметр.
... – дополнительные параметры.

Пример 30

```
data(iris)
model <- darch(Species ~ ., iris, darch.unitFunction = "exponentialLinearUnit",
   darch.elu.alpha = 2)
```

3.20. Функция *maxoutUnit()*

Функция вычисляет активацию нейронов и возвращает список, в котором первая запись представляет собой результат передаточной функции *maxout*, а вторая запись является производной от передаточной функции.

Вызов функции:

```
maxoutUnit(input, ..., poolSize = getParameter(".darch.maxout.poolSize", 2,
    ...), unitFunc = getParameter(".darch.maxout.unitFunction", linearUnit,
    ...), dropoutMask = vector())
```

Аргументы функции:

input – вход для функции активации;
... – дополнительные, передаваемые во внутреннюю функцию блока;
poolSize – размер каждого *maxout* региона;
unitFunc – функция внутреннего блока для *maxout*;
dropoutMask – вектор, содержащий маску *dropout*.

Процедура *maxout* устанавливает активацию всех нейронов, причем

один с самой высокой активацией в пуле 0. Если процедура используется без *maxoutWeightUpdate*, она становится алгоритмом local-winner-take-all, так как единственная разница между ними заключается в том, что результирующие используются для *maxout*.

Пример 31

```
data(iris)
  # LWTA:
  model <- darch(Species ~ ., iris, c(0, 50, 0),
   darch.unitFunction = c("maxoutUnit", "softmaxUnit"),
   darch.maxout.poolSize = 5, darch.maxout.unitFunction = "sigmoidUnit")
 # Maxout:
  model <- darch(Species ~ ., iris, c(0, 50, 0),
   darch.unitFunction = c("maxoutUnit", "softmaxUnit"),
   darch.maxout.poolSize = 5, darch.maxout.unitFunction = "sigmoidUnit",
   darch.weightUpdateFunction = c("weightDecayWeightUpdate",
"maxoutWeightUpdate"))
```

3.21. Функция *rectifiedLinearUnit()*

Функция вычисляет активацию нейронов и возвращает список, в котором первая запись представляет собой выпрямленную линейную активацию нейронов, а вторая запись является производной от передаточной функции.

Вызов функции:

```
rectifiedLinearUnit(input, ...)
```

Аргументы функции:

input – вход для функции активации;
... – дополнительные параметры.

Пример 32

```
data(iris)
model <- darch(Species ~ ., iris, darch.unitFunction = "rectifiedLinearUnit")
```

3.22. Функция *sigmoidUnit()*

Функция вычисляет активацию нейронов и возвращает список, в котором первая запись представляет собой результат вычисления сигмоидной функции, а вторая запись является производной от передаточной функции.

Вызов функции:

```
sigmoidUnit(input, ...)
```

Аргументы функции:

input – вход для функции активации;
... – дополнительные параметры.

Пример 33

```
data(iris)
model <- darch(Species ~ ., iris, darch.unitFunction = "sigmoidUnit")
```

3.23. Функция *softmaxUnit()*

Функция вычисляет активацию нейронов и возвращает список, в котором первая запись представляет собой результат вычисления функции *softmax*, а вторая запись является производной от передаточной функции.

Вызов функции:

```
softmaxUnit(input, ...)
```

Аргументы функции:

input – вход для функции активации;
... – дополнительные параметры.

Пример 34

```
data(iris)
model <- darch(Species ~ ., iris,
  darch.unitFunction = c("sigmoidUnit", "softmaxUnit"))
```

Подробную информацию о функции *softmax()* можно получить в [20].

3.24. Функция *softplusUnit()*

Функция вычисляет активацию нейронов и возвращает список, в котором первая запись представляет собой результат вычисления функции *softmax*, а вторая запись является производной от передаточной функции. Функция *softplus()* – это сглаженная версия выпрямленной линейной активации.

Вызов функции:

```
softplusUnit(input, ...)
```

Аргументы функции:

input – вход для функции активации;
... – дополнительные параметры.

Пример 35

```
data(iris)
data(iris)
model <- darch(Species ~ ., iris, darch.unitFunction = "softplusUnit")
```

3.25. Функция *tanhUnit()*

Функция вычисляет активацию нейронов и возвращает список, в котором первая запись представляет собой результат активации с помощью функции *tanh*, а вторая запись является производной от передаточной функции.

Вызов функции:

```
tanhUnit(input, ...)
```

Аргументы функции:

input – вход для функции активации;
... – дополнительные параметры.

Пример 36

```
data(iris)
model <- darch(Species ~ ., iris, darch.unitFunction = "tanhUnit")
```

3.26. Функция maxoutWeightUpdate()

Функция обновляет веса на *maxout* слоях, изменяя вес только активных нейронов. Кроме того, все веса в пуле должны быть одинаковыми. Существует еще другая функция обновления весов *weightDecayWeightUpdate()*.

Вызов функции:

```
maxoutWeightUpdate(darch, layerIndex, weightsInc, biasesInc, ...,
  weightDecay = getParameter(".darch.weightDecay", 0, darch),
  poolSize = getParameter(".darch.maxout.poolSize", 2, darch))
```

Аргументы функции:

darch – DArch фрагмент;
layerIndex – индекс слоя внутри сети;
weightsInc – матрица, содержащая запланированные весовые обновления из алгоритма точной настройки;
biasesInc – обновления весов смещения;
... – дополнительные параметры;
weightDecay – соответствует параметру *darch.weightDecay* функции *darch.default*, перед каждым обновлением веса умножаются на (1 – *weightDecay*);
poolSize – размер пулов maxout, смотри параметр *darch.maxout.poolSize* функции *darch()*.

Пример 37

```
data(iris)
model <- darch(Species ~ ., iris, c(0, 50, 0),
  darch.unitFunction = c("maxoutUnit", "softmaxUnit"),
  darch.maxout.poolSize = 5, darch.maxout.unitFunction = "sigmoidUnit",
  darch.weightUpdateFunction = c("weightDecayWeightUpdate",
      "maxoutWeightUpdate"))
```

3.27. Функция *weightDecayWeightUpdate()*

Функция обновляет веса, используя *weightDecay*. Она умножает веса на значение (1 – *weightDecay*) перед применением запланированных изменений веса.

Вызов функции:

```
weightDecayWeightUpdate(darch, layerIndex, weightsInc, biasesInc, ...,
  weightDecay = getParameter(".darch.weightDecay", 0, darch),
  debug = getParameter(".debug", F, darch))
```

Аргументы функции:

darch – DArch фрагмент;
layerIndex – индекс слоя внутри сети;

weightsInc – матрица, содержащая запланированные весовые обновления из алгоритма точной настройки;

biasesInc – обновления весов смещения;

... – дополнительные параметры;

weightDecay – соответствует параметру *darch.weightDecay* функции *darch.default*, перед каждым обновлением веса умножаются на (1 – *weightDeay*);

debug – внутренний флаг отладки.

Пример 38

```
data(iris)
model <- darch(Species ~ ., iris, c(0, 50, 0),
   darch.weightUpdateFunction = "weightDecayWeightUpdate")
```

3.28. Функция *plot.DArch()*

Функция строит различные графики в зависимости от параметра типа *type*.

Вызов функции:

```
plot(x, y = "raw", ..., type = y)
```

Аргументы функции:

x – экземпляр объекта *DArch*;

y – тип графика

... – дополнительные параметры, передаваемые функции;

type – определяет какой тип графика построить: *raw, class, time, momentum* или *net*.

Ниже поясняются типы графиков:

1. *raw* – среднеквадратичная ошибка (*MSE*) – это значение по умолчанию;
2. *class* – ошибка классификации;
3. *time* – график времени, необходимого для каждой эпохи;
4. *momentum* – график изменения темпа обучения;
5. *net* – вызывает *plotnet* для построения сети.

Пример 39

```
data(iris)
model <- darch(Species ~ ., iris)
plot(model)
plot(model, "class")
```

```
plot(model, "time")
plot(model, "momentum")
plot(model, "net")
```

3.29. Функция *predict.DArch()*

Функция передает данные через глубокую нейронную сеть. В результате формируется вектор или матрица выходов сети, в зависимости от параметра типа *type*.

Вызов функции:

```
predict(object, ..., newdata = NULL, type = "raw",
    inputLayer = 1, outputLayer = 0)
```

Аргументы функции:

object – экземпляр объекта *DArch*;

... – дополнительные параметры, если *newdata* – *NULL*, будет использоваться первый неименованный параметр вместо *newdata*;

newdata – новые данные для прогнозирования, *NULL* – вернуть последний сетевой выход;

type – тип вывода, один из: *raw, bin, class* или *character*. Тип *raw* возвращает выходной слой. Тип *bin* возвращает 1 для каждого вывода слоя больше 0.5 и 0 в противном случае, а тип *class* возвращает 1 для вывода с наивысшей активацией, в противном случае 0. Кроме того, при использовании *class* метки возвращаются, когда они доступны. Тип *character* совпадает с *class*, за исключением использования символьных векторов вместо факторов;

inputLayer – указывает номер слоя (> 0). В этот слой будут загружены данные из *newdata*. Обратите внимание, что номера рассчитываются с входного слоя, т. е. для сети с тремя слоями, 1 будет указывать входной слой;

outputLayer – указывает номер слоя (если > 0) или смещение (если <= 0) относительно последнего слоя. Возвращается выход данного слоя. Обратите внимание, что номера рассчитываются с входного слоя, т. е. для сети с тремя слоями, 1 будет указывать входной слой.

Пример 40

```
data(iris)
model <- darch(Species ~ ., iris, retainData = T)
predict(model)
```

3.30. Функция *print.DArch()*

Функция выводит подробные сведения о экземпляре *DArch*. Распечатанные данные включают параметры *DArch* и сводку статистики обучения.

Вызов функции:

```
print(x, …)
```

Аргументы функции:

x – экземпляр объекта *DArch*;
... – дополнительные параметры;

Пример 41

```
data(iris)
  model <- darch(Species ~ ., iris)
  print(model)
```

3.31. Функция *provide.MNIST()*

Функция загрузит сжатый набор данных *MNIST* и сохранит его в *.RData* файлах с помощью *readMNIST*. В результате булевское значение укажет на успех или неудачу.

Вызов функции:

```
provideMNIST(folder = "data/", download = F)
```

Аргументы функции:

folder – имя папки, включая конечную косую черту;
download – логическое указание, разрешено ли скачивание.

Пример 42

```
provideMNIST("mnist/", download = T)
```

4. ИНТЕРФЕЙСНЫЕ ПАКЕТЫ

Машинное обучение и, в частности, моделирование глубоких нейронных сетей широко используется исследовательскими организациями для разработки практических систем различного назначения. Именно поэтому, разработано большое число библиотек программного обеспечения для решения задач моделирования нейронных систем с большим диапазоном возможностей. Большой набор предоставляемых функций обеспечивает возможность строить сети практически любой сложности, а гибкая архитектура позволяет развернуть вычисления на один или несколько центральных или графических процессоров.

Реализация подобных подходов к моделированию осуществляется на базе гибких масштабируемых платформ, включающих в свой состав большое число многоядерных вычислителей. Ряд подобных платформ предоставляют полный доступ к их использованию на основе *API*.

В настоящем разделе рассмотрим три пакета, позволяющие из среды R воспользоваться возможностями различных платформ для моделирования нейронных систем глубокого обучения.

4.1. Пакет *H2O*

Этот пакет служит для запуска сервера *H2O* через *REST API* из среды *R* [6]. Пакет позволяет запускать базовые команды *H2O* с помощью команд *R*. Чтобы использовать пакет, необходимо сначала запустить *H2O*.

4.1.1. Запуск сервера

Для запуска *H2O* на локальном компьютере, необходимо вызвать *h2o.init()* без каких-либо аргументов, и *H2O* будет автоматически запущен на *localhost*: 54321, где *IP* – 127.0.0.1, а номер порта – 54321. Чтобы *H2O* работал в кластере, нужно указать *IP* и порт удаленной машины в качестве аргументов при вызове функции *h2o.init ()*. Ниже приведен пример запуска кластера:

```
> localH2O = h2o.init(ip = "localhost", port = 54321, startH2O = TRUE)

H2O is not running yet, starting it now...

java version "1.6.0_65"
Java(TM) SE Runtime Environment (build 1.6.0_65-b14-462-11M4609)
Java HotSpot(TM) 64-Bit Server VM (build 20.65-b04-462, mixed mode)

Starting H2O JVM and connecting: ..... Connection successful!

R is connected to the H2O cluster:
```

```
H2O cluster uptime:         7 seconds 908 milliseconds
H2O cluster version:        3.10.4.6
H2O cluster version age:    2 months and 4 days
H2O cluster name:           H2O_started_from_R_felixfilippov_kos278
H2O cluster total nodes:    1
H2O cluster total memory:   0.12 GB
H2O cluster total cores:    2
H2O cluster allowed cores:  2
H2O cluster healthy:        TRUE
H2O Connection ip:          localhost
H2O Connection port:        54321
H2O Connection proxy:       NA
H2O Internal Security:      FALSE
R Version:                  R version 3.2.1 (2015-06-18)

Note:  As started, H2O is limited to the CRAN default of 2 CPUs.
    Shut down and restart H2O as shown below to use all your CPUs.
        > h2o.shutdown()
        > h2o.init(nthreads = -1)
        >
```

H2O поддерживает ряд стандартных статистических моделей, таких как *GLM*, K-средних и *Random Forest*. Например, чтобы запустить *GLM*, достаточно вызвать функцию *h2o.glm()* с анализируемыми данными и параметрами *H2O* (переменная ответа, распределение ошибок и т. д.) в качестве аргументов. Операция будет выполняться на сервере, связанном с объектом данных, где выполняется *H2O*, а не внутри среды *R*.

Обратите внимание, что фактические данные не хранятся в рабочей области *R* и никакая фактическая работа не выполняется в среде *R*. Среда *R* сохраняет только именованные объекты, которые однозначно идентифицируют набор данных, модель и т. д. на сервере. Когда пользователь делает запрос, *R* запрашивает сервер через *REST API*, который в результате работы возвращает файл *JSON* с соответствующей информацией, которую *R* выводит на консоль.

Таким образом *R* для *H2O* – это математический движок с открытым исходным кодом для больших данных, который вычисляет алгоритмы параллельного распределенного машинного обучения в частности, такие как нейронные сети глубокого обучения в различных кластерных средах.

При использовании пакета *H2O* все вычисления выполняются (в высоко оптимизированном *Java*-коде) в кластере *H2O* и инициируются вызовами *REST* из среды *R*.

4.1.2. Функция h2o.deeplearning()

Функция создает модель глубокой нейронной сети использующую ядра центральных процессоров кластера. Строит многослойную искусственную

нейронную сеть прямого распространения и возвращает фрейм *H2OFrame*, который включает характеристики скрытых слоев.

Вызов функции (указаны значения по умолчанию, когда представлен перечень возможных значений – по умолчанию задается первое значение):

```
h2o.deeplearning(x, y, training_frame, model_id = NULL,
        validation_frame = NULL, nfolds = 0,
        keep_cross_validation_predictions = FALSE,
        keep_cross_validation_fold_assignment = FALSE, fold_assignment = c("AUTO",
        "Random", "Modulo", "Stratified"), fold_column = NULL,
        ignore_const_cols = TRUE, score_each_iteration = FALSE,
        weights_column = NULL, offset_column = NULL, balance_classes = FALSE,
        class_sampling_factors = NULL, max_after_balance_size = 5,
        max_hit_ratio_k = 0, checkpoint = NULL, pretrained_autoencoder = NULL,
        overwrite_with_best_model = TRUE, use_all_factor_levels = TRUE,
        standardize = TRUE, activation = c("Tanh", "TanhWithDropout", "Rectifier",
        "RectifierWithDropout", "Maxout", "MaxoutWithDropout"), hidden = c(200,
        200), epochs = 10, train_samples_per_iteration = -2,
        target_ratio_comm_to_comp = 0.05, seed = -1, adaptive_rate = TRUE,
        rho = 0.99, epsilon = 1e-08, rate = 0.005, rate_annealing = 1e-06,
        rate_decay = 1, momentum_start = 0, momentum_ramp = 1e+06,
        momentum_stable = 0, nesterov_accelerated_gradient = TRUE,
        input_dropout_ratio = 0, hidden_dropout_ratios = NULL, l1 = 0, l2 = 0,
        max_w2 = 3.4028235e+38, initial_weight_distribution = c("UniformAdaptive",
        "Uniform", "Normal"), initial_weight_scale = 1, initial_weights = NULL,
        initial_biases = NULL, loss = c("Automatic", "CrossEntropy", "Quadratic",
        "Huber", "Absolute", "Quantile"), distribution = c("AUTO", "bernoulli",
        "multinomial", "gaussian", "poisson", "gamma", "tweedie", "laplace",
        "quantile", "huber"), quantile_alpha = 0.5, tweedie_power = 1.5,
        huber_alpha = 0.9, score_interval = 5, score_training_samples = 10000,
        score_validation_samples = 0, score_duty_cycle = 0.1,
        classification_stop = 0, regression_stop = 1e-06, stopping_rounds = 5,
        stopping_metric = c("AUTO", "deviance", "logloss", "MSE", "RMSE", "MAE",
        "RMSLE", "AUC", "lift_top_group", "misclassification",
        "mean_per_class_error"), stopping_tolerance = 0, max_runtime_secs = 0,
        score_validation_sampling = c("Uniform", "Stratified"),
        diagnostics = TRUE, fast_mode = TRUE, force_load_balance = TRUE,
        variable_importances = FALSE, replicate_training_data = TRUE,
        single_node_mode = FALSE, shuffle_training_data = FALSE,
        missing_values_handling = c("MeanImputation", "Skip"), quiet_mode = FALSE,
        autoencoder = FALSE, sparse = FALSE, col_major = FALSE,
        average_activation = 0, sparsity_beta = 0,
        max_categorical_features = 2147483647, reproducible = FALSE,
        export_weights_and_biases = FALSE, mini_batch_size = 1,
        categorical_encoding = c("AUTO", "Enum", "OneHotInternal", "OneHotExplicit",
        "Binary", "Eigen"), elastic_averaging = FALSE,
        elastic_averaging_moving_rate = 0.9,
        elastic_averaging_regularization = 0.001)
```

Аргументы функции:

x – вектор, содержащий имена или индексы предикторных переменных для использования при построении модели. Если *x* отсутствует, то используются все столбцы, кроме *y*;

y – имя выходной переменной в модели. Если данные не содержат заголовок, это столбец с индексом увеличивающимся слева направо. Выход должен быть целым или категориальным;

training_frame – идентификатор фрейма данных обучения (не требуется, чтобы разрешить первоначальную проверку параметров модели);

model_id – идентификатор назначения для этой модели, если не указан, то генерируется автоматически;

validation_frame – идентификатор фрейма данных проверки;

nfolds – количество подмножеств наборов данных для *N*-кратной перекрестной проверки (0 для отключения или > = 2);

keep_cross_validation_predictions – сохранять ли прогнозы перекрестных проверок моделей;

keep_cross_validation_fold_assignment – сохранять ли распределение подмножеств перекрестных проверок;

fold_assignment – схема разбиения перекрестной проверки, если *fold_column* не указан. Параметр *Stratified* будет стратифицировать подмножества на основе выходной переменной для классификации. Должен быть одним из: *AUTO, Random, Modulo, Stratified*;

fold_column – столбец с индексом указателей перекрестной проверки на одно наблюдение;

ignore_const_cols – игнорировать ли постоянные столбцы;

score_each_iteration – нужно ли вести счет для каждой итерации обучения модели;

weights_column – столбец с весами наблюдения. Присвоение некоторому наблюдению веса 0 эквиваленто исключению его из набора данных. Присвоение наблюдению относительного веса 2 эквивалентено повторению этой строки дважды. Отрицательные веса не допускаются;

offset_column – столбец смещения, он будет добавлен к комбинации столбцов перед применением функции ссылки;

balance_classes – баланс классов данных тренировок с использованием избыточной/неполной выборки (для несбалансированных данных);

class_sampling_factors – коэффициенты избыточной/неполной выборки для каждого класса (в лексикографическом порядке). Если не указано, коэффициенты выборки будут автоматически вычисляться для получения баланса классов во время обучения. Требуется *balance_classes*;

max_after_balance_size – максимальный относительный размер данных обучения после балансировки класса (может быть меньше 1,0). Требуется *balance_classes*;

max_hit_ratio_k – максимальное число (до *K*) прогнозов, используемых для вычисления отношения успехов (только для мультиклассов, 0 для отключения);

checkpoint – контрольная точка модели для возобновления обучения;

pretrained_autoencoder – предварительная модель автокодера для инициализации этой модели;

overwrite_with_best_model – переопределить ли конечную модель на лучшую модель, найденную во время обучения;

use_all_factor_levels – использовать ли все уровни факторов категориальных переменных. В противном случае первый факторный уровень будет опущен (без потери точности). Полезно для важных переменных и автоподключено для автокодера;

standardize – если включено, данные стандартизуются автоматически. Если отключено, пользователь должен предоставить правильно масштабированные входные данные;

activation – функция активации, должна быть одной из: *Tanh, TanhWithDropout, Rectifier, RectifierWithDropout, Maxout* или *MaxoutWithDropout*;

hidden – размеры скрытых слоев (например, [100, 100]);

epochs – сколько раз набор данных должен быть итерирован (потоковый), может быть дробным;

train_samples_per_iteration – количество учебных образцов (в глобальном масштабе) на каждую итерацию *MapReduce*. Особые значения: 0: одна эпоха, -1: все доступные данные (например, реплицированные данные обучения), -2: автоматически;

target_ratio_comm_to_comp – целевое соотношение затрат на связь для вычислений. Только для работы с несколькими узлами и *train_samples_per_iteration* = -2 (автонастройка);

seed – начальная установка для случайных чисел (влияет на определенные части алгоритма, которые являются стохастическими, и они могут или не могут быть включены по умолчанию). Примечание: воспроизводится только при работе с одним потоком (*thread*). По умолчанию -1 (случайное число, основанное на времени);

adaptive_rate – адаптивная ли скорость обучения;

rho – коэффициент затухания времени адаптивного обучения (сходство с предыдущими обновлениями);

epsilon – адаптивный коэффициент сглаживания коэффициента обучения (чтобы избежать деления на ноль и обеспечить прогресс);

rate – скорость обучения (выше => меньше стабильности, ниже => более медленная сходимость);

rate_annealing – отклик на скорость обучения: *rate*/(1 + *rate_annealing* * *samples*);

rate_decay – коэффициент затухания скорости обучения между слоями (*n*-й слой: *rate* * *rate_decay* ^ (*n* - 1);

momentum_start – начальный импульс в начале тренировки (попробуйте 0.5);

momentum_ramp – количество учебных образцов, при котором импульс увеличивается;

momentum_stable – конечный момент после окончания образцов (попробуйте 0.99);

nesterov_accelerated_gradient – использовать ли градиент ускорения Нестерова (рекомендуется);

input_dropout_ratio – коэффициент отсечки (dropout) входного слоя (может улучшить обобщение, попробуйте 0,1 или 0,2);

hidden_dropout_ratios – скрытые коэффициенты отсечки (можно улучшить обобщение), укажите одно значение для скрытого слоя;

l1 – L1 регуляризация (может добавить стабильность и улучшить обобщение, приводит к тому, что многие веса становятся равными 0);

l2 – L2 регуляризация (может добавить стабильность и улучшить обобщение, приводит к тому, что многие веса становятся близкими к 0);

max_w2 – ограничение на квадрат суммы входящих весов на единицу (например, для *rectifier*);

initial_weight_distribution – начальное распределение весов. Должно быть одним из: *UniformAdaptive, Uniform* или *Normal*;

initial_weight_scale – Uniform: -значение… +значение, *Normal*: стандартное отклонение (*stddev*);

initial_weights – список идентификаторов *H2OFrame* для инициализации весовой матрицы данной модели;

initial_biases – список идентификаторов *H2OFrame* для инициализации вектора смещений данной модели;

loss – функция потерь, может быть одной из: *Automatic, CrossEntropy, Quadratic, Huber, Absolute* или *Quantile*. По умолчанию *Automatic*;

distribution – функция распределения, может быть одной из: *AUTO, bernoulli, multinomial, gaussian, poisson, gamma, tweedie, laplace, quantile* или *huber*. По умолчанию *AUTO*;

quantile_alpha – желаемый квантиль для *Quantile*-регрессии должен быть от 0 до 1;

tweedie_power – мощность для *Tweedie*-регрессии должна составлять от 1 до 2;

huber_alpha – желаемый квантиль для *Huber/M*-регрессии (порог между квадратичной и линейной потерями должен быть между 0 и 1);

score_interval – кратчайший интервал времени (в секундах) между модельным скорингом;

score_training_samples – количество образцов обучающих наборов для подсчета очков (0 для всех);

score_validation_samples – количество образцов утвержденных наборов для подсчета очков (0 для всех);

score_duty_cycle – максимальная доля рабочего цикла для подсчета очков (ниже: больше тренировок, выше: больше очков);

classification_stop – критерий остановки для ошибки классификации по данным обучения (-1 для отключения);

regression_stop – критерий остановки для ошибки регрессии (*MSE*) по данным обучения (-1 для отключения);

stopping_rounds – ранняя остановка основана на схождении stop_metric. Остановка, если простое изменение средней длины *k stop_metric* не улучшится для *k : = stopping_rounds* считает события (0 для отключения);

stopping_metric – метрика, используемая для ранней остановки Должно быть одно из: *AUTO* (*logloss* для классификации, *deviance* для регрессии), *deviance, logloss, MSE,* RMSE, MAE, RMSLE, AUC, lift_top_group, misclassification или mean_per_class_error. По умолчанию используется *AUTO*;

stopping_tolerance – относительный допуск для критерия остановки на метрической основе (остановка, если относительное улучшение не так сильно);

max_runtime_secs – максимально допустимое время выполнения в секундах для обучения модели. Для отключения используется 0;

score_validation_sampling – метод, используемый для выборки набора данных проверки для подсчета очков. Должен быть одним из: *Uniform* или *Stratified*;

diagnostics – включить ли диагностику скрытых слоев;

fast_mode – включить ли быстрый режим (малая аппроксимация при обратном распространении);

force_load_balance – настроить ли дополнительную балансировку нагрузки, чтобы увеличить скорость обучения для небольших наборов данных (чтобы все ядра были заняты);

variable_importances – вычислить ли значения важности для входных переменных (метод *Gedeon*) – может быть медленным для больших сетей;

replicate_training_data – продублировать ли весь набор данных обучения каждого узела для более быстрого обучения на небольших наборах данных;

single_node_mode – осуществлять ли запуск на одном узле для точной настройки параметров модели;

shuffle_training_data – включить ли перетасовку данных обучения (рекомендуется, если данные обучения дублируются);

missing_values_handling – обработка отсутствующих значений. Должен быть один из: *MeanImputation* или *Skip*;

quiet_mode – включить ли тихий режим для вывода на стандартный вывод;

autoencoder – формировать ли модель автокодера;

sparse – редкая обработка данных (более эффективная для данных с большим количеством нулевых значений);

col_major – использовать ли основной столбец матрицы весов для входного слоя. Может ускорить распространение вперед, но может замедлить *backpropagation*;

average_activation – средняя активация для разреженного автокодера;

sparsity_beta – осуществлять ли регуляризацию разреженности;

max_categorical_features – максимальное количество категориальных характеристик, выполняемых с помощью хэширования;

reproducible – усиливать ли воспроизводимость при малых данных (при медленной – будет использоваться только 1 поток);

export_weights_and_biases – экспортировать ли веса и смещения нейронной сети в *H2OFrame*;

mini_batch_size – минимальный размер обучающей выборки (меньший ведет к лучшей подгонке, больший позволяет лучше ускоряться и обобщаться);

categorical_encoding – схема кодирования для категориальных объектов, может быть одной из: *AUTO, Enum, OneHotInternal, OneHotExplicit, Binary* или *Eigen*;

elastic_averaging – применять ли упругое усреднение между вычислительными узлами. Может улучшить сходимость распределенной модели;

elastic_averaging_moving_rate – применять ли упругое усреднение скорости перемещения (только если разрешено упругое усреднение);

elastic_averaging_regularization – значение упругого усреднения регуляризации (только если разрешено упругое усреднение).

Пример 43

```
library(h2o)
h2o.init()
iris.hex <- as.h2o(iris)
iris.dl <- h2o.deeplearning(x = 1:4, y = 5, training_frame = iris.hex)
```

По сути аналогичная функция *h2o.deepwater()* применяется для построения модели *DNN* с использованием нескольких GPU. Она создает глубокую нейронную сеть на *H2OFrame*, содержащую различные источники данных. Подробно функция описана в [6].

4.1.3. Функция *h2o.predict()*

Функция получает прогнозы для различных объектов модели *H2O*.

Вызов функции:

```
h2o.predict(object, newdata, ...)
```

Аргументы функции:

object – объект *H2OModel*, который представляет модель, используемую для получения прогноза;

newdata – объект *H2OFrame*, в котором указаны переменные, для предсказания

... – дополнительные аргументы для передачи.

4.1.4. Метод *plot.H2OModel*

Этот метод отправляет тип модели H2O для выбора правильной истории подсчета очков. Аргументы *timestep* и *metric* ограничены тем, что доступно в истории подсчета для определенного типа модели. Метод возвращает график подсчета очков.

Вызов метода:

```
plot(x, timestep = "AUTO", metric = "AUTO", ...)
```

Аргументы:

x – объект *H2OModel*, для которого требуется график истории подсчета очков;

timestep – единица измерения по оси *x*;

metric – единица измерения по оси *y*;

. . . – дополнительные аргументы для передачи.

Пример 44

```
if (requireNamespace("mlbench", quietly=TRUE)) {
        ibrary(h2o)
        h2o.init()
        df <- as.h2o(mlbench::mlbench.friedman1(10000,1))
        rng <- h2o.runif(df, seed=1234)
        train <- df[rng<0.8,]
        valid <- df[rng>=0.8,]
        gbm <- h2o.gbm(x = 1:10, y = "y", training_frame = train, validation_frame = valid,
        ntrees=500, learn_rate=0.01, score_each_iteration = TRUE)
        plot(gbm)
        plot(gbm, timestep = "duration", metric = "deviance")
        plot(gbm, timestep = "number_of_trees", metric = "deviance")
        plot(gbm, timestep = "number_of_trees", metric = "rmse")
        plot(gbm, timestep = "number_of_trees", metric = "mae")
}
```

4.1.5. Функция *h2o.anomaly()*

Обнаружение аномалий в наборе данных *h2o* с использованием модели глубокого обучения с автокодированием.

Вызов функции:

```
h2o.anomaly(object, data, per_feature = FALSE)
```

Аргументы функции:

object – объект *H2OAutoEncoderModel*, который представляет модель, используемую для обнаружения аномалий;

data – объект *H2OFrame*;

per_feature – указывает, следует ли возвращать квадратичную ошибку реконструкции для каждой функции.

В качестве модели может использоваться *H2OAutoEncoderModel*, построенная с помощью функции *h2o.deeplearning()*. В результате работы функция возвращает объект *H2OFrame*, содержащий реконструкцию *MSE* или квадратичную ошибку для каждой функции.

Пример 45

```
prosPath = system.file("extdata", "prostate.csv", package = "h2o")
prostate.hex = h2o.importFile(path = prosPath)
prostate.dl = h2o.deeplearning(x = 3:9, training_frame = prostate.hex,
                               autoencoder = TRUE, hidden = c(10, 10), epochs = 5)
prostate.anon = h2o.anomaly(prostate.dl, prostate.hex)
head(prostate.anon)
prostate.anon.per.feature = h2o.anomaly(prostate.dl, prostate.hex, per_feature=TRUE)
head(prostate.anon.per.feature)
```

4.2. Пакет *Mxnet*

Пакет обеспечивает интерфейс между средой *R* и гибкой, масштабируемой платформой, которая поддерживает самые современные модели *DNN*, включая сверточные нейронные сети *CNN* и сети с долгой краткосрочной памятью *LSTM*.

4.2.1. Установка пакета

Установка пакета зависит от требуемой конфигурации и подробно описана на сайте [23]. Возможный выбор конфигурации для установки пакета приведен на рис.8.

Рис. 8. Выбор конфигурации для установки пакета *MXNET*

Так, для выбранной конфигурации (*MacOS, R, CPU*) сразу предлагается версия установки:

```
cran <- getOption("repos")
cran["dmlc"] <- https://s3-us-west-2.amazonaws.com/apache-mxnet/R/CRAN/
options(repos = cran)
install.packages("mxnet")
```

Для проверки инсталляции предлагается запустить короткую программу *MxNet R*, чтобы создать матрицу 2x3 состоящую из единиц, умножить каждый элемент в матрице на 2, а затем прибавить 1. Результат будет матрицей 2x3 со всеми элементами, равными 3:

```
library(mxnet)
a <- mx.nd.ones(c(2,3), ctx = mx.cpu())
b <- a*2 + 1
b
```

4.2.2. Символические выражения

Для разработки моделей нейронных сетей используются процедуры пакета символического выражения *mx.symbol* представляющего конфигурацию *API MxNet*.

Каждый элемент *mx.symbol.operator* представляет символическое выражение с несколькими выходами. Они являются операторами, такими как простые операции с матрицами (например, «+») или слой нейронной сети (например, слой свертки). Оператор может принимать несколько входных переменных, производить несколько выходных переменных и иметь внутренние переменные состояния. Переменная может быть либо свободной, которую мы можем связать со значением позже, либо выходом другого символа.

Ниже перечислены базовые операторы для формирования слоев нейронной сети глубокого обучения:

FullyConnected – выполняет линейное преобразование: $Y = XW^T + b$;
Convolution – выполняет операцию фильтрации входов;
Activation – выполняет заданную функцию активации;
BatchNorm – нормализует заданные значения;
Pooling – выполняет операцию пулинга;
SoftmaxOutput – вычисляет градиент потери кросс-энтропии по отношению к выходу;
Softmax – применяет функцию softmax.

Подробную информацию о всех дополнительных возможностях и специальных операторах можно найти в [28].

4.2.3. Построение сверточной нейронной сети

В качестве примера использования пакета рассмотрим решение классической задачи по распознаванию образов в *R* с использованием глубокой сверточной нейронной сети, подобной рис. 5. Подробное описание решения этой задачи можно найти на сайте [24], поэтому мы не будем останавливаться на подготовке обучающего и тестового набора данных, а опишем процедуры создания модели, ее обучения и тестирования.

Сначала создадим символьную модель CNN, последовательно определяя архитектуру ее слоев:

Пример 46

```
data <- mx.symbol.Variable('data')
# Первый слой
conv_1 <- mx.symbol.Convolution(data = data, kernel = c(5, 5), num_filter = 20)
tanh_1 <- mx.symbol.Activation(data = conv_1, act_type = "tanh")
pool_1 <- mx.symbol.Pooling(data = tanh_1, pool_type = "max", kernel = c(2, 2),
```

```
            stride = c(2, 2))
# Второй слой
conv_2 <- mx.symbol.Convolution(data = pool_1, kernel = c(5, 5), num_filter = 50)
tanh_2 <- mx.symbol.Activation(data = conv_2, act_type = "tanh")
pool_2 <- mx.symbol.Pooling(data=tanh_2, pool_type = "max", kernel = c(2, 2),
            stride = c(2, 2))
# Третий слой
flatten <- mx.symbol.Flatten(data = pool_2)
fc_1 <- mx.symbol.FullyConnected(data = flatten, num_hidden = 500)
tanh_3 <- mx.symbol.Activation(data = fc_1, act_type = "tanh")
# Четвертый слой
fc_2 <- mx.symbol.FullyConnected(data = tanh_3, num_hidden = 40)
# Пятый (выходной) слой
NN_model <- mx.symbol.SoftmaxOutput(data = fc_2)
```

В первом слое выполняется операция фильтрации с помощью 20 фильтров размера 5x5, реализуется функция активации *tanh* и осуществляется *max*-пулинг 2x2 с шагом 2 по вертикали и горизонтали.

Операции второго слоя аналогичны, за исключением того, что количество фильтров увеличено до 50.

Третий и четвертый слои являются полносвязными и содержит 500 и 40 скрытых нейронов, соответственно.

Наконец, выходной слой реализует немного нестандартный пример, а именно функцию активации *Softmax* для минимизации перекрестной энтропии. Эта модель актуальна при задаче классификации, когда необходимо получить на выходе нейросети вероятности принадлежности входного образа одному из не пересекающихся классов. Очевидно, что суммарный выход сети по всем нейронам выходного слоя должен равняться единице (так же как и для выходных образов обучающей выборки).

Для ***обучения полученной модели*** *NN_model* воспользуемся соответствующей функцией пакета *mxnet*:

Пример 47

```
model <- mx.model.FeedForward.create(
        NN_model,
         X = train_array,
         y = train_y,
         ctx = mx.cpu(),
         num.round = 480,
         array.batch.size = 40,
         learning.rate = 0.01,
         momentum = 0.9,
         eval.metric = mx.metric.accuracy,
         epoch.end.callback = mx.callback.log.train.metric(100)
)
```

Для ***тестирования обученной модели*** *model* используем стандартную функцию *predict()*, формирующую выходные значения *predicted* для тестовых входов *test_array*:

Пример 48

```
# Предсказание выхода
predicted <- predict(model, test_array)
# Присвоение меток
predicted_labels <- max.col(t(predicted)) - 1
# Вычисление точности предсказания
sum(diag(table(test[, 1], predicted_labels)))/40
```

После присвоения меток *predict_labels* полученным выходным значениям и подсчета среднего значения точности предсказания по всем тестовым входам получим точность предсказания обученной *CNN*. Так, в [24] для классического набора лиц *Olivetti* [25] подобная сеть обеспечила точность распознавания 0,975.

Приобретение практических навыков использования пакета *mxnet* для работы с моделями *CNN* на первых этапах требует тщательного изучения хороших примеров. Одним из таких может служить материал сайта [27], в котором на практических примерах рассматриваются особенности создания и обучения модели, а также вопросы использования итераторов и других дополнительных возможностей. В частности, исследуются приемы для работы с изображениями на базе функций пакета *imager*.

4.3. Пакет *tensorflow*

Пакет обеспечивает доступ к полному *API TensorFlow* изнутри среды *R*. *TensorFlow* – это библиотека программного обеспечения с открытым исходным кодом для численных расчетов с использованием графов потока данных. Узлы в графе представляют собой математические операции, а ребра графа – многомерные массивы данных (тензоры), передаваемые между ними. *API TensorFlow* состоит из набора модулей *Python*, которые позволяют создавать и вычислять графы потока данных.

Гибкая архитектура позволяет развернуть вычисления на один или несколько центральных или графических процессоров с помощью единого *API*. *TensorFlow* был первоначально разработан исследователями и инженерами, работающими в команде *Google* в исследовательской организации *Google Machine Intelligence* для целей машинного обучения и моделирования глубоких нейронных сетей.

TensorFlow использует высокоэффективный сервер для выполнения своих вычислений. Соединение с этим сервером называется сессией. Обычное использование *TensorFlow* – сначала создать всю программу (граф вычислений), а затем запустить ее в сессии. Однако вместо этого можно ис-

пользовать удобный класс *InteractiveSession*, позволяющий сделать *TensorFlow* более удобным для его изучения за счет интерактивных сессий. Использование этого класса позволяет чередовать операции, которые создают программу (граф) вычислений с теми, которые запускают ее на исполнение. Это особенно удобно при интерактивной работе в консоли *R*:

```
library(tensorflow)
sess <- tf$InteractiveSession()
```

Ниже в данном разделе приводится описание некоторых функций пакета и рассматриваются примеры его применения для построения нейронных сетей.

4.3.1. Установка пакета

Для установки пакета используется функция *install_tensorflow()*:

```
install_tensorflow(method = c("auto", "virtualenv", "conda", "system"),
    version = "latest", gpu = FALSE, package_url = NULL, conda = "auto")
```

Аргументы функции:

method – способ установки. По умолчанию *auto* – автоматически находит метод, который будет работать в локальной среде. Измените значение по умолчанию, чтобы принудительно установить определенный метод. Примечание: метод *virtualenv* недоступен в *Windows*, а метод *system* доступен только в *Windows*;

version – версия *TensorFlow* для установки (должна быть либо «последней», либо полной спецификацией *major.minor.patch*, например «1.1.0»);

gpu – установка *GPU* версии *TensorFlow*;

package_url – URL-адрес пакета *TensorFlow* для установки (если не указано, это определяется автоматически). Примечание: если этот параметр указан, параметры *version* и *gpu* игнорируются;

conda – путь к исполняемому *conda* (или «авто», чтобы найти *conda* с помощью *PATH* и других обычных мест расположения).

Вместе с *TensorFlow* полезно установить дополнительные пакеты *Python* с помощью функции *install_tensorflow_extras()*:

```
install_tensorflow_extras(packages, conda = "auto")
```

Для этой функции требуется версия *TensorFlow*, ранее установленная с помощью функции *install_tensorflow()*. Для значений аргументов *virtualenv* и *conda* указанные пакеты будут установлены в среду *r-tensorflow*. Для системных установок в *Windows* указанные пакеты будут установлены в библиотеку системных пакетов.

Дополнительную документацию по пакету *tensorflow* можно найти на сайте [27].

4.3.2. Пример нахождения апроксимирующей прямой

Приведем простой пример использования библиотеки *tensorflow* для нахождения аппроксимирующей прямой для случайных данных сгенерированных в плоскости x, y.

Пример 49

```
library(tensorflow)

# Создаем 100 фоновых x, y точек данных, y = x * 0,1 + 0,3
x_data <- runif(100, min=0, max=1)
y_data <- x_data * 0.1 + 0.3

# Попытаемся найти значения для W и b, вычисляющие y_data = W * x_data + b
W <- tf$Variable(tf$random_uniform(shape(1L), -1.0, 1.0))
b <- tf$Variable(tf$zeros(shape(1L)))
y <- W * x_data + b

# Минимизируем среднеквадратическую ошибку
loss <- tf$reduce_mean((y - y_data) ^ 2)
optimizer <- tf$train$GradientDescentOptimizer(0.5)
train <- optimizer$minimize(loss)


# Открываем сессию и запускаем граф вычислений
sess = tf$Session()
sess$run(tf$global_variables_initializer())

#  Отыскиваем оптимальную линию (лучше всего подходит W: 0,1, b: 0,3)
for (step in 1:201) {
  sess$run(train)
  if (step %% 20 == 0)
    cat(step, "-", sess$run(W), sess$run(b), "\n")
 }
```

Первая часть кода из примера строит граф потока данных. *TensorFlow* фактически не запускает никаких вычислений до тех пор, пока не будет создан сеанс *sess* и не будет вызвана функция запуска *sess$run*.

4.3.3. Построение классификатора

Рассмотрим основные строительные блоки модели *TensorFlow* при построении классификатора *MNIST*.

Подготовка данных

Прежде чем мы создадим нашу модель, загрузим набор данных *MNIST* и запустим сеанс *TensorFlow*:

Пример 50

```
library(tensorflow)
input_dataset <- tf$examples$tutorials$mnist$input_data
mnist <- input_dataset$read_data_sets("MNIST-data", one_hot = TRUE)
 sess <- tf$InteractiveSession()
```

Здесь *mnist* – это простой класс, в котором хранятся наборы данных для обучения, проверки и тестирования как матрицы *R*. Он также предоставляет функцию для итерации мини-наборов данных, которые мы будем использовать ниже.

Для эффективного вычисления в *R* мы обычно вызываем базовые функции *R*, которые выполняют дорогостоящие операции, такие как матричное умножение вне *R*, используя высокоэффективный код, реализованный на другом языке. Однако много накладных расходов может быть затрачено при переключении на R для каждой операции. Эти накладные расходы особенно вредны, если требуется запускать вычисления на графических процессорах или распределенными способами, где может быть высокая стоимость передачи данных. *TensorFlow* позволяет нам описывать граф взаимодействующих операций, полностью работающих вне *R* и роль *R*-кода заключается в построении этого внешнего графа вычислений.

Сначала мы исследуем код *mnist_softmax.R* из [29], который является базовой реализацией модели *TensorFlow*, а затем покажем некоторые способы повышения точности классификации.

Теперь построим *модель регрессии softmax* с одним линейным слоем, а затем распространим ее на случай регрессии *softmax* с многослойной сверточной сетью. Начнем строить граф вычислений, создавая узлы для входных изображений и целевых (выходных) классов:

Пример 51

```
x <- tf$placeholder(tf$float32, shape(NULL, 784L))
y_ <- tf$placeholder(tf$float32, shape(NULL, 10L))
```

Здесь переменные *x* и *y_* не являются конкретными значениями, каждая из них является контейнером, который будет заполняться значениями когда *TensorFlow* начнет выполнить вычисление. Входные *x* изображения цифр будут состоять из *2d*-тензора чисел с плавающей запятой. Здесь ему присваивается размерность (*NULL*, 784), где 784 – размерность одного 28x28 пиксель-

ного *MNIST*-изображения, а *NULL* указывает, что первое измерение, соответствующее размеру партии, может быть любого размера. Целевые классы выхода *y_* также будут состоять из 2d-тензора, где каждая строка является одним 10-мерным вектором, указывающим, какой разряд вектора (от нуля до девяти) соответствует соответствующему изображению *MNIST*.

Аргумент *shape* для контейнера необязателен, но позволяет *TensorFlow* автоматически обнаруживать ошибки, возникающие из-за несогласованных тензорных форм.

Теперь определим веса *W* и смещения *b*. В *TensorFlow* для этого используются переменные. Переменные – это значения, которые используются и модифицируются в графе вычислений *TensorFlow*. В приложениях машинного обучения, как правило, параметры модели являются переменными:

Пример 52

```
W <- tf$Variable(tf$zeros(shape(784L, 10L)))
b <- tf$Variable(tf$zeros(shape(10L)))
```

Начальное значение для каждого параметра передается в вызове *tf$ Variable*. В данном случае *W* и *b* инициализируются как тензоры, полные нулей. *W* – матрица 784x10 (потому что у нас есть 784 входных функции и 10 выходов), а *b* – 10-мерный вектор (потому что у нас 10 классов).

Прежде чем переменные могут использоваться в сеансе, они должны быть инициализированы с использованием этого сеанса. Это можно сделать для всех переменных одновременно:

Пример 53

```
sess$run(tf$global_variables_initializer())
```

Теперь можно реализовать регрессионную модель – она займет всего одну строку - умножаем векторизованные входные изображения *x* на весовую матрицу *W*, добавляем смещение *b* и вычисляем вероятности *softmax*, которые присваиваются каждому классу:

Пример 54

```
y <- tf$nn$softmax(tf$matmul(x,W) + b)
```

Теперь просто определить функцию потерь. Потеря показывает, насколько плохим было предсказание модели на одном примере. Здесь функция потерь – это кросс-энтропия между целью и предсказанием модели:

Пример 55

```
cross_entropy <- tf$reduce_mean(-tf$reduce_sum(y_ * tf$log(y), reduction_indices=1L))
```

Когда мы определили модель и функцию потерь можно начать обучение модели. Поскольку определен весь граф вычислений, можно использовать автоматическое дифференцирование, чтобы найти градиенты потерь по каждой из переменных. TensorFlow имеет множество встроенных алгоритмов оптимизации [30]. В этом примере для достижения кросс-энтропии будем использовать крутой градиентный спуск с шагом 0,5:

Пример 56

```
optimizer <- tf$train$GradientDescentOptimizer(0.5)
train_step <- optimizer$minimize(cross_entropy)
```

Здесь добавлены новые операции к графу вычислений. Это операция вычисления градиентов и вычисление шагов обновления и обновление параметров.

При запуске операции *train_step* параметры будут обновляться на основе градиентного спуска. Таким образом, обучение модели будет реализовано путем многократного запуска *train_step*:

Пример 57

```
for (i in 1:1000) {
  batches <- mnist$train$next_batch(100L)
  batch_xs <- batches[[1]]
  batch_ys <- batches[[2]]
  sess$run(train_step,
        feed_dict = dict(x = batch_xs, y_ = batch_ys))
 }
```

На каждой итерации обучения мы загружаем 100 примеров. Затем запускаем операцию *train_step*, используя *feed_dict*, чтобы заменить тензоры контейнеры *x* и *y_* примерами обучения. Обратите внимание, что можно заменить любой тензор на графе вычислений с помощью *feed_dict* – он не ограничивается только заполнителями.

Чтобы узнать, насколько хорошо наша модель сначала выясним, где предсказана правильная метка:

Пример 58

```
correct_prediction <- tf$equal(tf$argmax(y, 1L), tf$argmax(y_, 1L))
```

Здесь использована функция t*f$argmax*, которая дает индекс записи максимального значения в тензоре вдоль некоторой оси. Например, *tf$argmax(y, 1L)* – это метка, которую модель считает наиболее вероятной для каждого входа, тогда как *tf$rgmax (y_, 1L)* является истинной меткой. Мы

можем использовать *tf$equal*, чтобы проверить, соответствует ли наше предсказание истине.

Это дает нам список логических элементов. Чтобы определить, какая фракция верна, мы переводим логические значения в числа с плавающей запятой, а затем вычисляем среднее значение. Например, *c*(*TRUE, FALSE, TRUE, TRUE*) станет *c*(1,0,1,1), который даст среднее значение 0,75:

Пример 59

```
accuracy <- tf$reduce_mean(tf$cast(correct_prediction, tf$float32))
```

Наконец, можно оценить точность распознавания на тестовых данных:

Пример 60

```
accuracy$eval(feed_dict=dict(x = mnist$test$images, y_ = mnist$test$labels))
## [1] 0.9165
```

Точность не превышает 92%, что является очень плохим результатом для *MNIST*. Для повышения точности заменим простую регрессионную модель на модель сверточной нейронной сети *CNN*.

Инициализация весов и смещений

Чтобы создать эту модель, нам нужно будет создать множество весов и смещений. Обычно необходимо инициализировать вес с небольшим количеством шума для нарушения симметрии и предотвратить градиенты 0. Поскольку мы используем нейроны *ReLU*, рекомендуется также инициализировать их с небольшим положительным исходным смещением, чтобы избежать «мертвых нейронов». Вместо того, чтобы повторять это несколько раз в процессе построения модели, создадим две удобные функции:

Пример 61

```
weight_variable <- function(shape) {
  initial <- tf$truncated_normal(shape, stddev=0.1)
  tf$Variable(initial)
 }
  bias_variable <- function(shape) {
  initial <- tf$constant(0.1, shape=shape)
  tf$Variable(initial)
 }
```

Фильтрация и пулинг

TensorFlow обеспечивает нам большую гибкость в задании параметров для операций свертки (фильтрации) и пулинга (объединения). В этом примере выберем версию *vanilla*: шаг фильтрации 1 и нулевой *padding*, так что вы-

ход имеет тот же размер, что и вход; пулинг - простой максимум по блокам 2x2. Чтобы упростить код, также реализуем эти операции как функции:

Пример 62

```
conv2d <- function(x, W) {
  tf$nn$conv2d(x, W, strides=c(1L, 1L, 1L, 1L), padding='SAME')
}
 max_pool_2x2 <- function(x) {
  tf$nn$max_pool(
    x,
    ksize=c(1L, 2L, 2L, 1L),
    strides=c(1L, 2L, 2L, 1L),
    padding='SAME')
 }
```

Первый сверточный слой

Реализуем первый слой, состоящий из свертки, за которым следует максимальный пулинг. Свертка будет вычислять 32 функции для каждого ядра 5x5. Ее тензор будет иметь форму (5, 5, 1, 32). Первые два измерения - размер ядра фильтра, далее следует количество входных каналов и количество выходных. Кроме того будет вектор смещения с компонентом для каждого выходного канала:

Пример 63

```
W_conv1 <- weight_variable(shape(5L, 5L, 1L, 32L))
b_conv1 <- bias_variable(shape(32L))
```

Чтобы применить слой для обработки входа, мы сначала переформатируем *x* на *4d*-тензор со вторым и третьим размерами, соответствующими ширине и высоте изображения, и четвертым значением, соответствующим количеству цветных каналов:

Пример 64

```
x_image <- tf$reshape(x, shape(-1L, 28L, 28L, 1L))
```

Теперь фильтруем *x_image* с помощью весового тензора *W_conv1*, добавляем смещение *b_conv1*, применяем функцию *ReLU* и, наконец, применяем операцию max-пулинга:

Пример 65

```
h_conv1 <- tf$nn$relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 <- max_pool_2x2(h_conv1)
```

После выполнения операции *max*-пулинга на первом слое карты признаков будут иметь размер 14x14.

Второй сверточный слой

Чтобы построить глубокую сеть, формируем несколько сверточных слоев. Второй слой будет иметь 64 функции для каждого ядра 5x5:

Пример 66

```
W_conv2 <- weight_variable(shape = shape(5L, 5L, 32L, 64L))
b_conv2 <- bias_variable(shape = shape(64L))
h_conv2 <- tf$nn$relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 <- max_pool_2x2(h_conv2)
```

После выполнения операции *max*-пулинга на втором слое карты признаков будут иметь размер 7x7.

Полносвязный слой

Теперь, когда размер изображения уменьшеньшился до 7x7, добавляем полносвязный слой с 1024 нейронами, чтобы осуществить обработку всего изображения. Для этого преобразуем тензор предыдущего слоя в вектор, умножаем на весовую матрицу, добавляем смещение и применяем функцию активации *ReLU*:

Пример 67

```
W_fc1 <- weight_variable(shape(7L * 7L * 64L, 1024L))
b_fc1 <- bias_variable(shape(1024L))
h_pool2_flat <- tf$reshape(h_pool2, shape(-1L, 7L * 7L * 64L))
h_fc1 <- tf$nn$relu(tf$matmul(h_pool2_flat, W_fc1) + b_fc1)
```

Отсев (dropout) нейронов

Чтобы уменьшить переобучение, применим операцию отсева (исключения) нейронов до выходного слоя. Для этого создадим контейнер (местозаполнитель) для записи вероятности того, что выход нейрона сохраняется во время отсева. Это позволит включить отсев во время обучения и отключить его во время тестирования. *TensorFlow* выполняя операцию *tfnndropout()* автоматически обрабатывает масштабирование выходов нейронов в дополнение к их маскировке, поэтому отсев просто работает без какого-либо дополнительного масштабирования.

Пример 68

```
keep_prob <- tf$placeholder(tf$float32)
h_fc1_drop <- tf$nn$dropout(h_fc1, keep_prob)
```

Подробную информацию об операции отсева можно получить в [31].

Выходной слой

В выходном слое используем операцию *softmax*, как и для *CNN* в примерах 46-48:

Пример 69

```
W_fc2 <- weight_variable(shape(1024L, 10L))
b_fc2 <- bias_variable(shape(10L))
y_conv <- tf$nn$softmax(tf$matmul(h_fc1_drop, W_fc2) + b_fc2)
```

Обучение и оценка модели

Для обучения и оценки точности модели используем код, который почти идентичен коду из примеров * - *+5 для простой однослойной сети. Различия будут заключаются в следующем:

- заменим оптимизатор градиентного спуска *GradientDescentOptimizer* на более сложный оптимизатор *AdamOptimizer*;
- включим дополнительные параметры *keep_prob* в *feed_dict*, чтобы контролировать скорость отсева;
- добавим регистрацию на каждую 100-ю итерацию в процессе обучения.

Пример 70

```
cross_entropy <- tf$reduce_mean(
        -tf$reduce_sum(y_ * tf$log(y_conv), reduction_indices=1L))
train_step <- tf$train$AdamOptimizer(1e-4)$minimize(cross_entropy)
correct_prediction <- tf$equal(tf$argmax(y_conv, 1L), tf$argmax(y_, 1L))
accuracy <- tf$reduce_mean(tf$cast(correct_prediction, tf$float32))

sess$run(tf$global_variables_initializer())

 for (i in 1:20000) {
  batch <- mnist$train$next_batch(50L)
  if (i %% 100 == 0) {
    train_accuracy <- accuracy$eval(feed_dict = dict(
       x = batch[[1]], y_ = batch[[2]], keep_prob = 1.0))
    cat(sprintf("step %d, training accuracy %g\n", i, train_accuracy))
  }
  train_step$run(feed_dict = dict(
    x = batch[[1]], y_ = batch[[2]], keep_prob = 0.5))
 }
 test_accuracy <- accuracy$eval(feed_dict = dict(
     x = mnist$test$images, y_ = mnist$test$labels, keep_prob = 1.0))
 cat(sprintf("test accuracy %g", test_accuracy))
```

Оценка точности распознавания *MNIST* на данном тестовом наборе после запуска этого кода должна составлять примерно 99,2%. Таким образом мы создали, обучили и оценили точность работы довольно сложной модели глубокого обучения на базе пакета *TensorFlow.*

4.4. Пакет *Keras*

Keras — открытая библиотека, написанная на языке *Python*. Она содержит многочисленные реализации широко применяемых строительных блоков нейронных сетей, таких как слои, оптимизаторы, целевые и передаточные функции, и множество других инструментов.

С практической точки зрения *Keras* - это высокоуровневый *API* разработанный с упором на возможность быстрого экспериментирования с моделями нейронных сетей. Он имеет следующие ключевые особенности:

- обеспечивает дружественный *API*, который позволяет просто и быстро создавать модели глубокого обучения;
- позволяет выполнять одинаковый код на *CPU* или *GPU*;
- способен работать поверх нескольких *back-end*, включая *TensorFlow*, *CNTK* или *Theano*;
- поддерживает произвольные сетевые архитектуры: модели с несколькими входами и выходами, совместное использование слоев, совместное использование моделей и т. д.

Интерфейс *R* для *Keras* обеспечивает программный пакет *keras* [33].

4.4.1. Установка пакета

Интерфейс *R* для *Keras* по умолчанию использует серверную систему *TensorFlow*. Чтобы установить как основную библиотеку *Keras*, так и серверную систему *TensorFlow* можно использовать функцию *install_keras():*

Пример 71

```
install_keras(method = c("virtualenv", "conda"), conda = "auto",
              tensorflow = "default", extra_packages = NULL)
```

Аргументы:

method - способ установки (*virtualenv* или *conda*);

conda - путь к исполняемому *conda* (или *auto*, чтобы найти *conda* с помощью *PATH* и других обычных мест установки);

tensorflow - версия *TensorFlow* для установки, *default* для установки процессора последней версии, добавка *gpu* для установки версии *GPU* последней версии;

extra_packages - дополнительные пакеты *PyPI* для установки вместе с *Keras* и *TensorFlow*.

Далее будут рассмотрены особенности использования интерфейса *R* для *Keras* на примерах построения классификатора *MNIST*, сверточной сети подобной *VGG* и модели накапливающей *LSTM* для классификации последовательностей. Подробную информацию обо всех функциях, используемых в представленных ниже примерах можно найти на сайте [33].

4.4.2. Построение классификатора *MNIST*

Набор данных *MNIST* включен в *Keras* и может быть загружен с помощью функции *dataset_mnist()*. Загрузим этот набор и создадим переменные для наших обучающих и тестовых данных:

Пример 72

```
library(keras)
mnist <- dataset_mnist()
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y
```

Входные данные *x* представляют собой массив значений оттенков серого для элементов матрицы 28x28. Чтобы подготовить данные для обучения, преобразуем матрицу 28x28 в вектор длиной 784. Значения оттенков серого из целых чисел от 0 до 255 преобразуем в вещественные значения с плавающей запятой в диапазоне от 0 до 1, поделив их на 255:

Пример 73

```
dim(x_train) <- c(nrow(x_train), 784)
dim(x_test) <- c(nrow(x_test), 784)
# rescale
x_train <- x_train / 255
x_test <- x_test / 255
```

Данные *y* представляют собой целочисленный вектор со значениями от 0 до 9. Чтобы подготовить эти данные для обучения, кодируем векторы в двоичные матрицы классов с использованием *Keras* функции *to_categorical()*:

Пример 74

```
y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)
```

Функция *to_categorical()* выполняет так называемое прямое (*on-hot*) кодирование.

Определение модели

Основная структура данных *Keras* - это модель, определяющая способ организации и взаимодействия слоев нейронной сети. Простейшим типом модели является последовательная модель, линейная совокупность слоев. Начнем создавать последовательную модель путем постепенного добавления слоев с использованием оператора «добавления» (*pipe*), обозначаемого как %>%:

Пример 75

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = 'softmax')
```

Аргумент *input_shape* в первом слоя определяет форму входных данных (числовой вектор длиной 784, представляющий изображение в градациях серого). Выходной слой выводит числовой вектор длиной 10 (вероятности для каждой цифры) с использованием функции активации *softmax*. Для получения детальной послойной информации о модели model можно использовать функцию *summary()*.

Теперь можно скомпилировать модель с соответствующей функцией потерь, типом оптимизатора и метрикой:

Пример 76

```
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)
```

Обучение и оценка

Для обучения модели используется функция *fit()*. В качестве параметров обучения зададим 30 эпох, размер партии – 128 входных изображений:

Пример 77

```
history <- model %>% fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
)
```

Объект *history*, возвращаемый методом *fit()*, включает показатели потерь и точности, которые можно отобразить графически (рис. 9):

Пример 78

```
plot(history)
```

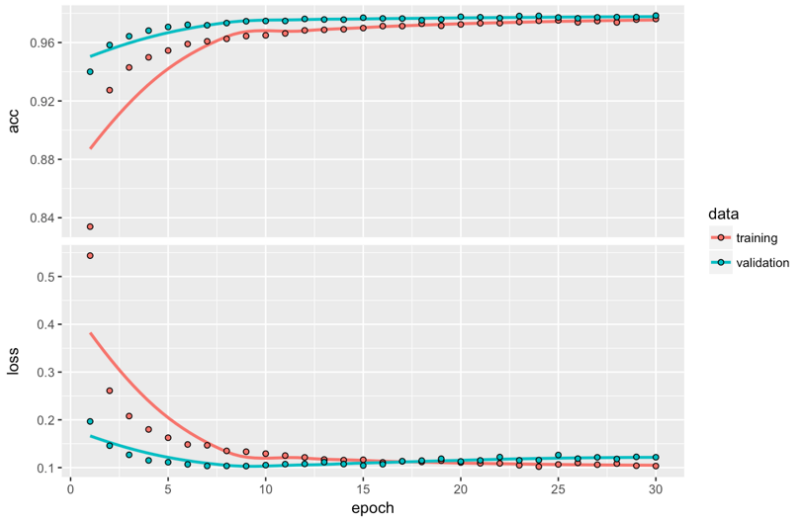

Рис. 9. График зависимости точности и потерь от числа итераций

Оценим производительность модели по данным теста:

```
loss_and_metrics <- model %>% evaluate(x_test, y_test)
```

Создание прогнозов по новым данным:

```
classes <- model %>% predict_classes(x_test)
```

Keras предоставляет словарь для создания моделей глубокого обучения, которые просты, элегантны и интуитивно понятны.

4.4.3. Сверточная сеть *VGG*

В качестве обучающего *x_train, y_train* и проверочного *x_train, y_train* наборов сгенерируем фиктивные данные с помощью генератора случайных чисел:

Пример 79

```
library(keras
x_train <- array(runif(100 * 100 * 100 * 3), dim = c(100, 100, 100, 3))

y_train <- runif(100, min = 0, max = 9) %>%
   round() %>%
```

```
  matrix(nrow = 100, ncol = 1) %>%
   to_categorical(num_classes = 10)

x_test <- array(runif(20 * 100 * 100 * 3), dim = c(20, 100, 100, 3))

y_test <- runif(20, min = 0, max = 9) %>%
   round() %>%
  matrix(nrow = 20, ncol = 1) %>%
   to_categorical(num_classes = 10)
```

Создадим модель сверточной сети для обработки изображений размером 100x100, с обработкой по трем цветовым каналам. Для обработки используем 32 фильтра размером 5x5:

Пример 80

```
model <- keras_model_sequential()
model %>%
  layer_conv_2d(filters = 32, kernel_size = c(5,5), activation = 'relu',
                input_shape = c(100,100,3)) %>%
  layer_conv_2d (filters = 32, kernel_size = c(5,5), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_conv_2d (filters = 64, kernel_size = c(5,5), activation = 'relu') %>%
  layer_conv_2d  (filters = 64, kernel_size = c(5,5), activation = 'relu') %>%
  layer_max_pooling_2d (pool_size = c(2,2)) %>%
  layer_dropout (rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = 'relu') %>%
  layer_dropout (rate = 0.25) %>%
  layer_dense (units = 10, activation = 'softmax')
```

Откомпилируем полученную модель:

Пример 81

```
model %>%
compile(
   loss = 'categorical_crossentropy',
   optimizer = optimizer_sgd(lr = 0.01, decay = 1e-6,
   momentum = 0.9, nesterov = TRUE)
  )
```

Обучим откомпилированную модель, используя 10 итераций и подавая обучающие данные отдельными партиями размером 32. Оценим точность ее работы:

Пример 82

```
model %>% fit(x_train, y_train, batch_size = 32, epochs = 10)
score <- model %>% evaluate(x_test, y_test, batch_size = 32)
```

4.4.4. Модель накапливающей *LSTM*

Данная модель используется для классификации последовательностей. В ней мы образуем три слоя LSTM друг над другом, делая модель способной изучать временные представления достаточно высокого уровня.

Пример 83

```
# задаем размерности входных данных
data_dim <- 16
timesteps <- 8
num_classes <- 10

# формируем LSTM модель
model <- keras_model_sequential()
model %>%
  layer_lstm(units = 32, return_sequences = TRUE, input_shape = c(timesteps,
data_dim)) %>%
   layer_lstm(units = 32, return_sequences = TRUE) %>%
   layer_lstm(units = 32) %>% # возвращаем один вектор размерности 32
   layer_dense(units = 10, activation = 'softmax') %>%

# компилируем модель
   compile(
    loss = 'categorical_crossentropy',
    optimizer = 'rmsprop',
    metrics = c('accuracy')
  )

   # генерируем фиктивные данные
x_train <- array(runif(1000 * timesteps * data_dim), dim = c(1000, timesteps,
data_dim))
y_train <- matrix(runif(1000 * num_classes), nrow = 1000, ncol = num_classes)  #
generate dummy validation data
x_val <- array(runif(100 * timesteps * data_dim), dim = c(100, timesteps, data_dim))
y_val <- matrix(runif(100 * num_classes), nrow = 100, ncol = num_classes)

# обучаем полученную модель на фиктивных данных
model %>% fit(    x_train, y_train, batch_size = 64, epochs = 5, validation_data =
list(x_val, y_val)
 )
```

Первые два слоя *LSTM* возвращают свои полные выходные последовательности, а последний возвращает только последний шаг в своей выходной последовательности, тем самым снижая временную размерность, т. е. преобразует входную последовательность в один вектор.

СОКРАЩЕНИЯ

AE – autoencoder – автокодер
API – application programming interface – интерфейс программирования приложения (интерфейс прикладного программирования)
AUC – area under ROC curve – площадь под *ROC*-кривой
CD – contrastive divergence – контрастное расхождение
CG – conjugate gradient – сопряженный градиент
CNN – convolutional neural network – сверточная нейронная сеть
CNTK – cognitive toolkit – когнитивный инструментарий
CPU – central processing unit – центральный процессор
CUDA – compute unified device architecture — программно-аппаратная архитектура параллельных вычислений
DBN – deep belief network – глубокая сеть доверия
DNN – deep neural network – глубокая нейронная сеть
ELU – exponential linear unit – экспоненциальный линейный элемент
FCNN – fully connected neural network – полносвязная нейронная сеть
GPU – graphics processing unit – графический процессор
GRU – gated recurrent unit – управляемый рекуррентный модуль
ILSVRC – imagenet large scale visual recognition challenge – кампания по широкомасштабному распознаванию образов
LSTM – long short-term memory – долгая краткосрочная память
MAE – mean absolute error – средняя абсолютная ошибка
MLP – multilayer percptron – многослойный персептрон
MSE – mean squared error – среднеквадратическая ошибка
PCA – principal component analysis – метод главных компонент
PyPI - python package index — каталог пакетов *Python*
RBM – restricted Boltzmann machine – ограниченная машина Больцмана
ReLU – rectified linear unit – блок линейной ректификации
REST – representational state transfer — передача состояния представления
RMSE – root mean squared error – корень среднеквадратичной ошибки
RMSLE – root mean squared log error – логарифмический корень среднеквадратичной ошибки
RNN – recurrent neural network – рекуррентная нейронная сеть
ROC – receiver operating characteristic – рабочая характеристика приёмника (кривая ошибок)
SAE – stacked autoencoder – накапливающий автокодер
SRBM – stacked restricted Boltzmann machine – накапливающая ограниченная машина Больцмана
Theano — библиотека численного вычисления в *Python*
VGG – visual geometry group – группа визуальной геометрии (предложила *VGG*-модель *CNN*, Оксфордский университет)

ИСПОЛЬЗОВАННЫЕ ИСТОЧНИКИ

1. Филиппов Ф.В. Моделирование нейронных сетей на R: учебное пособие, СПбГУТ, – СПб., 2016. – 86 с.
2. Hinton G. Training Products of Experts by Minimizing Contrastive Divergence. Neural Computation, 14, 1771–1800, 2002.
3. http://image-net.org/challenges/LSVRC/2017/results
4. Package ‘deepnet’ Feb. 19, 2015. https://cran.r-project.org/web/packages/deepnet/deepnet.pdf
5. Package ‘darch’ July 20, 2016. https://cran.r-project.org/web/packages/darch/darch.pdf
6. Package ‘h2o’ July 1, 2017. https://cran.r-project.org/web/packages/h2o/h2o.pdf
7. MXNet Documentation Release 0.0.8 Aug 07, 2017. https://media.readthedocs.org/pdf/mxnet-test/latest/mxnet-test.pdf
8. Package ‘tensorflow’ July 27, 2017. https://cran.r-project.org/web/packages/tensorflow/tensorflow.pdf
9. Hinton, G. E., S. Osindero, Y. W. Teh, A fast learning algorithm for deep belief nets, Neural Computation 18(7), S. 1527-1554, DOI: 10.1162/neco.2006.18.7.1527 2006.
10. Hinton, G. E., R. R. Salakhutdinov, Reducing the dimensionality of data with neural networks, Science 313(5786), S. 504-507, DOI: 10.1126/science.1127647, 2006.
11. Hinton G.E., 2002, DOI:10.1162/089976602760128018
12. https://cs231n.github.io/neural-networks-3/#sgdConvolutionalNeural Networks for Visual Recognition.
13. https://www.rdocumentation.org/packages/caret/versions/6.0-73/topics/preProcess Pre-Processing of Predictors
14. https://rdrr.io/rforge/caret/man/dummyVars.html
15. https://github.com/maddin79/darch/tree/v0.12.0/examples
16. Филиппов Ф.В. Программирование на языке R: практикум. СПбГУТ, – СПб., 2017.
17. https://topepo.github.io/caret/using-your-own-model-in-train.html
18. Glorot X., Bengio Y. Understanding the difficulty of training deep feed-forward neural networks. International conference on artificial intelligence and statistics, 2010, pp. 249-256.
19. He K., Zhang X., Ren S., Sun J. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification https://arxiv.org/abs/150 2.01852
20. http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-12.html
21. Riedmiller M., Braun H.. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In Proceedings of the IEEE International Conference on Neural Networks, pp 586-591. IEEE Press, 1993.
22. Igel C., Huesken M. Improving the Rprop Learning Algorithm, Proceedings

of the Second International Symposium on Neural Computation, NC 2000, ICSC Academic Press, Canada/Switzerland, pp. 115-121., 2000.
23. http://mxnet.io/get_started/install.html
24. http://firsttimeprogrammer.blogspot.ru/2016/08/image-recognition-tutorial-in-r-using.html
25. http://scikit-learn.org/stable/datasets/olivetti_faces.html
26. Deep learning с использованием языка R и библиотеки mxnet. https://statist-bhfz.github.io/mxnet_usage.html
27. https://tensorflow.rstudio.com
28. http://mxnet.io/tutorials/index.html
29. https://github.com/rstudio/tensorflow/blob/master/inst/examples/mnist/mnist_softmax.R
30. https://www.tensorflow.org/api_docs/python/train.html#optimizers
31. Srivastava N., Hinton G., Krizhevsky A., Sutskever I., Salakhutdinov R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Journal of Machine Learning Research 15, 2014, p. 1929-1958
32. https://keras.io
33. https://rstudio.github.io/keras/index.html

Филиппов Феликс Васильевич

Моделирование нейронных сетей глубокого обучения

Учебное пособие

Редактор

План 2017 г., п.

Подписано к печати
Объем усл.-печ. л. Тираж экз. Заказ

Издательство СПбГУТ. 191186 СПб., наб. р. Мойки, 61