Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

К. С. Курочка, К. А. Панарин

НЕЙРОСЕТЕВАЯ ОБРАБОТКА ДАННЫХ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ
для студентов специальностей
1-40 05 01 «Информационные системы и технологии
(по направлениям)» и 1-40 80 04 «Информатика
и технологии программирования»
дневной и заочной форм обучения

Электронный аналог печатного издания

Гомель 2021

УДК 004.032.26(075.8)
ББК 32.818.1я73
К93

Рекомендовано к изданию научно-методическим советом факультета автоматизированных и информационных систем ГГТУ им. П. О. Сухого
(*протокол № 10 от 01.06.2020 г.*)

Учебно-методическое пособие выполнено в ходе реализации проекта MaCICT (Modernisation of Master Curriculum in Information Computer Technologies) в рамках Erasmus+ Программы Европейского Союза. В этом проекте представлены лучшие практики вузов-партнеров из Евросоюза для модернизации учебного плана и дисциплин с целью внедрения в учебный процесс изучения гибких навыков, позволяющих повысить конкурентоспособность на рынке труда выпускников ИТ-специальностей.

Рецензент: зав. каф. «Автоматизированные системы обработки информации» ГГУ им. Ф. Скорины канд. техн. наук, доц. *В. Д. Левчук*

Курочка, К. С.

К93 Нейросетевая обработка данных : учеб.-метод. пособие для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» днев. и заоч. форм обучения / К. С. Курочка, К. А. Панарин. – Гомель : ГГТУ им. П. О. Сухого, 2021. – 260 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

ISBN 978-985-535-480-3.

Содержит методологию проектирования нейронных сетей, выбора способа их обучения и технологий применения в системах обработки данных по дисциплинам «Введение в нейронные сети» (главы 1, 2), «Многослойные нейронные сети и распознавание образов» (главы 3, 4) и «Нейросетевая обработка данных» (главы 5, 6).

Для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)», 1-40 80 04 «Информатика и технологии программирования» дневной и заочной форм обучения.

УДК 004.032.26(075.8)
ББК 32.818.1я73

ISBN 978-985-535-480-3

Оглавление

ПРЕДИСЛОВИЕ

Пособие предназначено для студентов первой ступени получения высшего образования по специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» и второй ступени получения высшего образования по специальности 1-40 80 04 «Информатика и технологии программирования».

В данной работе последовательно излагается материал, посвященный нейросетевому моделированию и обработке данных, приводятся сведения об используемом математическом аппарате, рассматривается ряд нейросетевых топологий и методик их проектирования и обучения, приведены примеры применения нейросетевых моделей на решении задач классификации и распознавания образов.

Благодаря широте охвата материала, который преподается нескольким курсам первой и второй ступени получения высшего образования, формат учебно-методического пособия позволяет студентам изучать данные дисциплины последовательно и с пониманием того, как одна дисциплина является логическим продолжением другой, что дает возможность в конечном итоге сформировать у студентов целостное понимание нейросетевой обработки данных как мультидисциплинарной науки, состоящей из математического базиса, информационно-алгоритмического иструментария и практико ориентированной проблематикой.

В первой главе изложены этапы развития теории нейронных сетей, приводятся классы решаемых с помощью нейронных сетей задач, описывается персептрон Розенблата, методы его обучения и примеры практического использования.

Во второй главе рассмотрены простейшие нейронные сети с обратными и латеральными связями, стохастические нейронные сети и методики их обучения.

Третья глава посвящена многослойным нейронным сетям прямого распространения. Приведены сведения о методике глубокого обучения и оценки качества обучения, проанализированы вопросы регуляризации и оптимизации при обучении глубоких моделей.

Четвертая глава данной работы содержит описание современных программных фреймворков для построения нейронных сетей глубокого обучения, технологию их использования для решения ряда практических задач.

Пятая глава описывает современные нейронные сети глубокого обучения, такие как сверточные, рекуррентные, реккурсивные, генеративно-состязательные. Даны архитектуры, методики обучения и классы решаемых задач.

В шестой главе приведены практические аспекты применения нейросетевого анализа данных. Рассмотрены примеры использования нейронных сетей для анализа данных, изложены концепции глубокого обучения с подкреплением и нечетких нейронных сетей.

Представленное учебно-методическое пособие может быть полезно студентам технических специальностей при изучении методов интеллектуального анализа данных.

ГЛАВА 1. ОСНОВНЫЕ ЭТАПЫ РАЗВИТИЯ ТЕОРИИ НЕЙРОННЫХ СЕТЕЙ

1.1. Биологические и искусственные нейронные сети

1.1.1. Машинное обучение

Иерархия методов машинного обучения представлена на рис. 1.1.

Рис. 1.1. Иерархия методов машинного обучения

Контролируемое обучение использует маркированные наборы данных, которые состоят из входных данных и ожидаемых результатов.

Когда вы обучаете искусственный интеллект (ИИ) с помощью контролируемого обучения, то подаете на вход данные и указываете, каким должен быть результат [1].

Если результат, который выдал ИИ отличается от ожидаемого, то он должен исправить свои вычисления. Процесс повторяется многократно над массивом данных, до тех пор, пока ИИ допускает ошибки.

Примером контролируемого обучения может служить искусственный интеллект, предсказывающий погоду. Он обучается предсказывать погоду, используя данные за длительный период времени. Входными данными служат давление, влажность и скорость ветра, а в результате мы должны получить температуру.

Неконтролируемое обучение – это задача, которая состоит в обучении ИИ с использованием неструктурированных данных.

Когда вы тренируете искусственный интеллект с помощью неконтролируемого обучения, то позволяете ИИ осуществить логическую классификацию данных.

Пример искусственного интеллекта, использующего неконтролируемое машинное обучение, – робот-предсказатель поведения клиентов интернет-магазина, который обучается, не используя заранее известные входные и выходные данные. Вместо этого он должен самостоятельно классифицировать входные данные. Алгоритму нужно определить и сообщить вам, какой тип пользователей какие продукты предпочитает.

Обучение с подкреплением – алгоритм выбирает действие в ответ на каждую точку данных.

Это наиболее распространенный подход в робототехнике, где набор показаний датчиков в один момент времени представляет собой точку данных, а алгоритму необходимо выбрать следующее действие робота.

Кроме того, он естественным образом подходит для приложений из Интернета вещей. Алгоритм обучения также вскоре получает сигнал, оповещающий об успехе, который дает понять, насколько удачно было принято решение.

На основе этого алгоритм изменяет свою стратегию для достижения лучшего результата.

1.1.2. Задачи, решаемые с помощью искусственных нейронных сетей

Аппроксимация и интерполирование данных. Рассмотрим общую постановку задачи.

Имеется выборка экспериментальных данных, представляющая собой множество примеров (опытов). Каждый пример – это вектор значений входных переменных и соответствующих им значений выходов.

Требуется с использованием одной из архитектур нейронных сетей получить нейросетевую модель, позволяющую с нулевой или наименьшей из возможных погрешностью описать многосвязную зависимость выходов от входов в пределах изменения независимых переменных модели, соответствующих диапазонам их изменения в экспериментальной выборке.

Прогнозирование временных рядов. Представим общую постановку задачи.

Дана выборка с результатами наблюдений изменения одной или нескольких целевых характеристик во времени. Требуется получить нейросетевую модель, позволяющую спрогнозировать значения целевых характеристик в один или несколько заданных моментов времени вперед. Помимо динамики целевых переменных для обучения нейронных сетей зачастую используются дополнительные переменные, характеризующие внешние условия, влияющие на эту динамику.

Во многих случаях в нейросетевые модели добавляются обратные связи, позволяющие подавать в качестве входных переменных значения, полученные на выходе в результате прохождения сигналов по нейронной сети на предыдущем такте расчета.

Специально для прогнозирования временных рядов исследователями были предложены архитектуры нейронных сетей с обратными связями на основе многослойных перцептронов, такие, как искусственные нейронные сети Элмана и Джордана.

Распознавание и ассоциация образов. Задачи распознавания и ассоциации образов сводятся к выделению некоторого эталонного сигнала, хранящегося в памяти нейронной сети из зашумленного входного образа.

Приведем общую постановку таких задач.

Дано множество эталонных образов либо пар соответствия входных и выходных эталонных образов в виде векторов бинарных и (или) аналоговых сигналов.

Нужно получить нейросетевую модель, позволяющую:

– удалить из входного сигнала шум (ошибки) и получить на выходе чистый (эталонный) сигнал;

– сопоставить зашумленный входной образ с соответствующим ему эталонным выходным образом.

Классификация образов. Задача классификации ставится следующим образом.

Имеется множество векторов входных образов, переменные которых представлены в форме аналоговых и (или) бинарных сигналов. Для каждого из входных образов известен класс соответствия. Требуется получить нейросетевую модель, позволяющую выполнить одно из следующих действий:

– однозначно указать класс образов, к которому может быть отнесен входной вектор;

– ограничить круг возможных классов, к которым может быть отнесен зашумленный входной образ;

– сделать заключение о невозможности отнесения входного образа к одному из известных сети классов.

Кластеризация данных. Представим постановку задачи кластеризации.

Дана экспериментальная выборка, представляющая собой множество векторов переменных, характеризующих известные свойства объектов, событий, процессов или явлений. Для каждого из примеров выборки могут быть известны дополнительные данные или описание, относящиеся лишь к самим отдельно взятым примерам, но не участвующие непосредственно в обучении нейронной сети.

Требуется получить нейронную сеть, которая на протяжении всего своего жизненного цикла будет выполнять одну или несколько функций из списка:

– относить подаваемые на ее входы вектора к одному из уже существующих кластеров, наиболее близкому к входному вектору;

– создавать новый кластер, если не обнаружено достаточно близкого соответствия существующим кластерам;

– удалять кластеры, оказавшиеся неиспользованными в результате последней эпохи обучения.

1.1.3. Исторические аспекты развития теории нейронных сетей

Термин «нейронная сеть» появился в середине XX в. Первые работы, в которых были получены основные результаты в данном направлении, были проделаны МакКаллоком и Питтсом (*MacKallok–Pitts*). В 1943 г. ими была разработана компьютерная модель нейронной сети на основе математических алгоритмов и теории деятельности головного мозга [2]. Они выдвинули предположение, что нейроны можно упрощенно рассматривать как устройства, оперирующие двоичными числами, и назвали эту модель «пороговой логикой». Подобно своему биологическому прототипу, нейроны МакКаллока–Питтса были способны обучаться путем подстройки параметров, описывающих синаптическую проводимость.

В 1949 г. Д. Хебб (*D. Hebb*) предлагает первый алгоритм обучения. Он высказал идеи о характере соединения нейронов мозга и их взаимодействии и первым предположил, что обучение заключается,

в первую очередь, в изменениях силы синаптических связей. Теория Хебба считается типичным случаем самообучения, при котором испытуемая система спонтанно обучается выполнять поставленную задачу без вмешательства со стороны экспериментатора.

В 1958 г. Ф. Розенблатт изобретает однослойный перцептрон. Перцептрон обретает популярность – его используют для распознавания образов, прогнозирования погоды и т. д.

В 1969 г. М. Минский публикует формальное доказательство ограниченности перцептрона и показывает, что он неспособен решать некоторые задачи, связанные с инвариантностью представлений, не способен реализовать функцию «Исключающее ИЛИ». Интерес к нейронным сетям резко спадает.

1974 г. – Пол Дж. Вербос и А. И. Галушкин одновременно изобретают алгоритм обратного распространения ошибки для обучения многослойных перцептронов. Изобретение не привлекло особого внимания.

1975 г. – Фукусима представляет когнитрон – самоорганизующуюся сеть, предназначенную для инвариантного распознавания образов, но это достигается только при помощи запоминания практически всех состояний образа.

1982 г. – после периода забвения интерес к нейросетям вновь возрастает. Дж. Хопфилд показал, что нейронная сеть с обратными связями может представлять собой систему, минимизирующую энергию (так называемая сеть Хопфилда).

Кохоненом представлены модели сети, обучающейся без учителя (нейронная сеть Кохонена), решающей задачи кластеризации, визуализации данных (самоорганизующаяся карта Кохонена) и другие задачи предварительного анализа данных.

1986 г. – Дэвидом И. Румельхартом, Дж. Е. Хинтоном и Рональдом Дж. Вильямсом переоткрыт и существенно развит метод обратного распространения ошибки. Начался взрыв интереса к обучаемым нейронным сетям.

2007 г. – Джеффри Хинтоном в университете Торонто создал алгоритмы глубокого обучения многослойных нейронных сетей. Успех обусловлен тем, что Хинтон при обучении нижних слоев сети использовал ограниченную машину Больцмана (*RBM – Restricted Boltzmann Machine*).

Глубокое обучение по Хинтону – это очень медленный процесс. Необходимо использовать много примеров распознаваемых образов (например, множество лиц людей на разных фонах). После обучения

получается готовое быстро работающее приложение, способное решать конкретную задачу (например, осуществлять поиск лиц на изображении).

1.1.4. Биологический прототип нейронной сети

Нейрон (нервная клетка) является особой биологической клеткой (рис. 1.2), которая обрабатывает информацию и состоит из *сомы* – тела нейрона, к ней через отростки – *дендриты* подходят *аксоны* – линии передач от других нейронов.

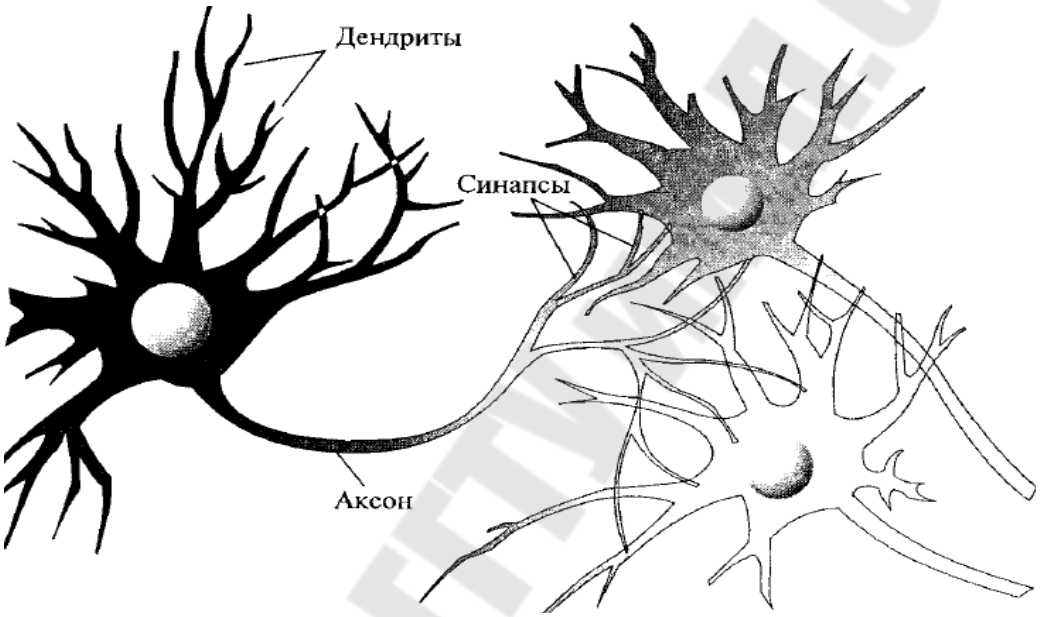


Рис. 1.2. Биологические нейроны

Синапс преобразует входную информацию, переводя нейрон в состояние возбуждения или торможения.

Нейроны взаимодействуют посредством короткой серии импульсов, как правило, продолжительностью несколько миллисекунд (мс). Сообщение передается посредством частотно-импульсной модуляции. Частота может изменяться от нескольких единиц до сотен герц, что в миллион раз медленнее, чем самые быстродействующие переключательные электронные схемы. Тем не менее, сложные решения по восприятию информации, такие, как, например, распознавание лица, человек принимает за несколько сотен миллисекунд. Эти решения контролируются сетью нейронов, которые имеют скорость выполнения операций всего несколько миллисекунд. Другими словами, для таких сложных задач мозг «запускает» параллельные программы, содержащие около 100 шагов.

Количество информации, посылаемое от одного нейрона другому, должно быть очень маленьким (несколько бит).

Мозг человека состоит из 10^{11} параллельно работающих нейронов, каждый связан с 10^4 других нейронов, так что всего в биологической нейросети 10^{15} нейросвязей.

1.1.5. Формальная модель искусственного нейрона

Искусственный нейрон – узел нейронной сети, представляющий собой упрощенную модель естественного нейрона (рис. 1.3).

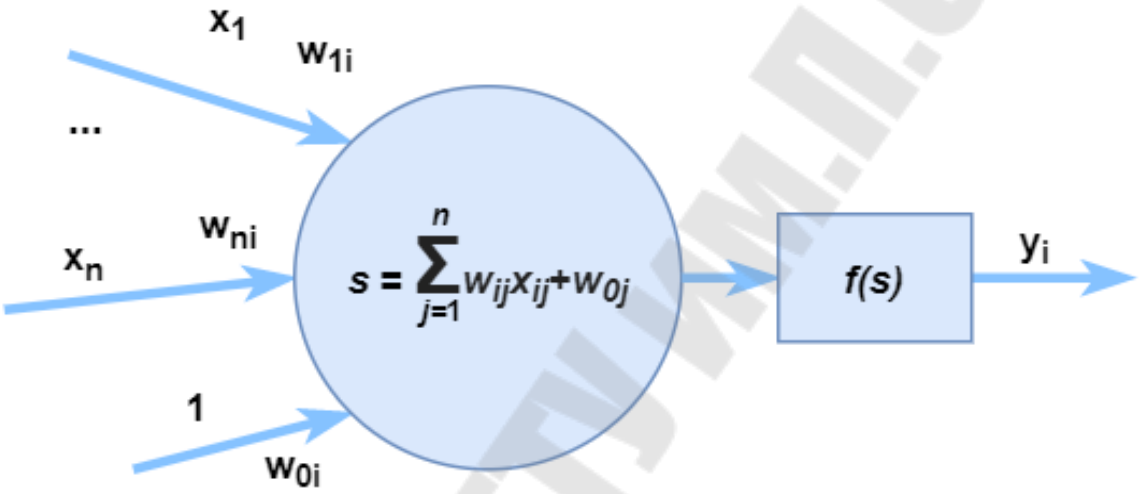


Рис. 1.3. Искусственный нейрон

С математической точки зрения он является сумматором всех входящих сигналов, применяющим к полученной сумме некоторую простую функцию. Нейроны оперируют числами в диапазоне [0,1] или [–1,1]. Если числа выходят за пределы диапазона, производится их нормализация:

$$y_i = F\left(\sum_{j=1}^{n} w_j x_j\right), \qquad (1.1)$$

где $F(S)$ – функция активации.

Функция активации (активационная функция, функция возбуждения) – функция, вычисляющая выходной сигнал искусственного нейрона. В качестве аргумента принимает сигнал Y, получаемый на выходе входного сумматора. Наиболее часто используются три функции активации. Их основное отличие – диапазон значений.

Графики различных функций активации приведены на рис. 1.4.

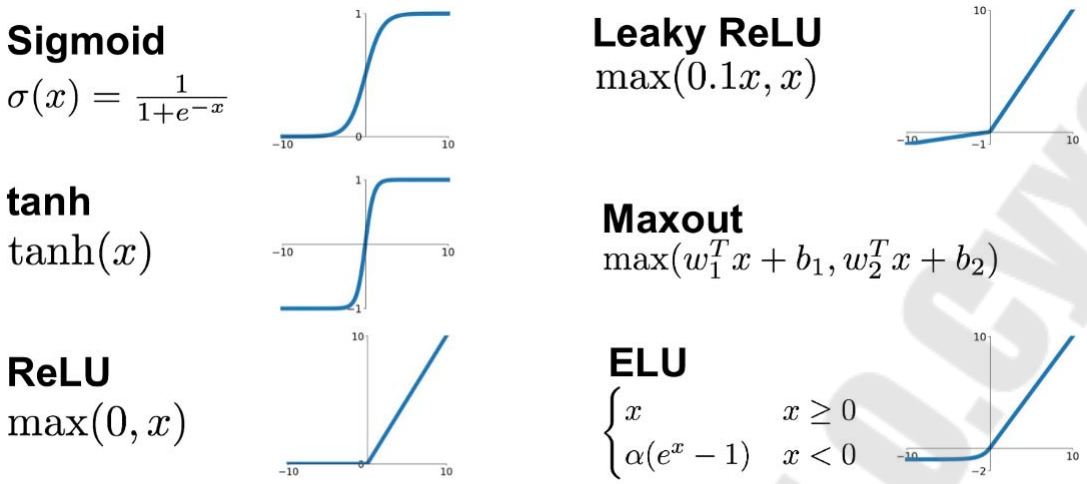


Рис. 1.4. Графики различных функций активации

Пороговая функция. Если входное значение меньше порогового, то значение функции активации равно минимальному допустимому, иначе – максимально допустимому:

$$F(S) = \begin{cases} 0, & \text{если } S < S_p; \\ 1, & \text{если } S \geq S_p. \end{cases} \tag{1.2}$$

Простая кусочно-линейная функция:

$$F(S) = \begin{cases} 0, & \text{если } S \leq 0; \\ 1, & \text{если } S \geq 1; \\ \alpha S. \end{cases} \tag{1.3}$$

Сигмоидальная функция, *или сигмоид*. Монотонно возрастающая всюду дифференцируемая нелинейная функция.

Например, логистическая функция:

$$F(S) = \frac{1}{1 + e^{-\alpha S}}, \tag{1.4}$$

где α – параметр наклона сигмоидальной функции активации. Изменяя этот параметр, можно построить функции с различной крутизной.

Гиперболический тангенс:

$$F(S) = \tanh\left(\frac{S}{\alpha}\right). \tag{1.5}$$

1.1.6. Классификация нейронных сетей

Классификация нейронных сетей по ***характеру обучения*** подразделяется следующим образом:

– нейронные сети, использующие обучение с учителем;

– нейронные сети, использующие обучение без учителя;

– смешанная парадигма обучения.

Обучение с учителем предполагает, что для каждого входного вектора существует целевой вектор, представляющий собой требуемый выход.

Обучение без учителя является намного более правдоподобной моделью обучения с точки зрения биологических корней искусственных нейронных сетей. Развитая Кохоненом и многими другими, она не нуждается в целевом векторе для выходов и, следовательно, не требует сравнения с предопределенными идеальными ответами. Обучающее множество состоит лишь из входных векторов.

Обучающий алгоритм подстраивает веса сети так, чтобы получались согласованные выходные векторы, т. е. чтобы **предъявление достаточно близких входных векторов давало одинаковые выходы**.

Процесс обучения, следовательно, выделяет статистические свойства обучающего множества и группирует сходные векторы в классы.

При *смешанном обучении* часть весов определяется посредством обучения с учителем, в то время как остальная получается с помощью самообучения.

Классификация по настройке весов представлена следующим образом:

– сети с фиксированными связями – весовые коэффициенты нейронной [3] сети выбираются сразу, исходя из условий задачи;

– сети с динамическими связями – для них в процессе обучения происходит настройка синаптических весов.

Классификация по типу входной информации подразделяется таким образом:

– аналоговая – входная информация приведена в форме действительных чисел;

– двоичная – вся входная информация в таких сетях представляется в виде нулей и единиц.

Классификация по характеру связей подразделяется следующим образом:

– сети прямого распространения (*Feedforward*). Все связи направлены строго от входных нейронов к выходным. Примерами таких сетей являются перцептрон Розенблатта, многослойный перцептрон;

– рекуррентные нейронные сети. Сигнал с выходных нейронов или нейронов скрытого слоя частично передается обратно на входы нейронов входного слоя (обратная связь).

1.1.7. Выбор данных для обучения

Выбор данных для обучения сети и их обработка является самым сложным этапом решения задачи. Набор данных для обучения должен удовлетворять нескольким критериям:

– *репрезентативность* – данные должны иллюстрировать истинное положение вещей в предметной области;

– *непротиворечивость* – противоречивые данные в обучающей выборке приведут к плохому качеству обучения сети.

Исходные данные преобразуются к виду, в котором их можно подать на входы сети. Каждая запись в файле данных называется обучающей парой или обучающим вектором. Обучающий вектор содержит по одному значению на каждый вход сети и в зависимости от типа обучения (с учителем или без) – по одному значению для каждого выхода сети.

Существует ряд способов улучшить «восприятие» сети:

– *нормировк*а – выполняется, когда на различные входы подаются данные разной размерности. Например, на первый вход сети подается величины со значениями от нуля до единицы, а на второй – от ста до тысячи. При отсутствии нормировки значения на втором входе будут всегда оказывать существенно большее влияние на выход сети, чем значения на первом входе. При нормировке размерности всех входных и выходных данных сводятся воедино;

– *квантование* – осуществляется над непрерывными величинами, для которых выделяется конечный набор дискретных значений. Например, квантование используют для задания частот звуковых сигналов при распознавании речи;

– *фильтрация* – проводится для «зашумленных» данных.

1.1.8. Сеть МакКаллока–Питтса

Нейрон МакКаллока–Питтса (рис. 1.5) представляет собой формальную модель нервной клетки, состоящей из тела и нескольких отростков, называемых нервными волокнами.

В модели МакКаллока–Питтса клетка – это ячейка, снабженная специальным числовым атрибутом, так называемым порогом. Из каждого нейрона выходит ребро, моделирующее нервное волокно, – аксон.

Аксон может разветвляться, но обязательно является входом для одного или нескольких других нейронов. Таким образом, кроме исходящего волокна (аксона) нейрон имеет несколько входящих волокон (синапсов).

Некоторые из входов могут быть возбуждающими, другие – тормозящими. Возбуждающие волокна обозначаются обычной стрелкой, а тормозящие – прерывистой стрелкой.

Каждый нейрон генерирует сигнал на аксоне только в том случае, если количество возбуждающих входных сигналов превышает порог данного нейрона.

Аксоны обозначаются сплошными исходящими стрелками. Тело нейрона обозначается треугольником (рис. 1.5).

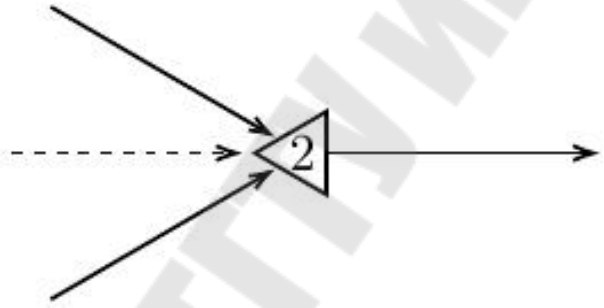


Рис. 1.5. Нейрон МакКаллока–Питтса

Нейроны объединяются в сети, именуемые «нейронные сети МакКаллока–Питтса». Вся сеть функционирует как конечный автомат.

Для этого в сети вводится понятие времени. В каждый момент времени выходной сигнал от некоторого нейрона переходит на вход того нейрона, к которому этот нейрон подключен в качестве входа.

Таким образом, фиксируя моменты времени, мы получаем состояния нервной сети, а переход из одного состояния в другое происходит при увеличении текущего момента времени.

Аксоны некоторых нейронов в сети могут быть синапсами для этих же нейронов, т. е. нейроны могут образовывать «петли», фактически – обратные связи.

Наличие петель очень важно для нейронной сети МакКаллока–Питтса. Петли позволяют запоминать сигналы, которые были поданы на вход. Если на вход нейрона был подан сигнал, то можно возбудить

аксон, который является входом для этого же нейрона, и, таким образом, с этого момента данный нейрон будет находиться в активном состоянии, самовозбуждаясь с каждым новым моментом времени.

Возбуждение некоторого сигнала на выходе сети называется событием.

Для описания событий в сети МакКаллока–Питтса используются три оператора:

– «и» («*»);

– «или» («|»);

– оператор конкатенации (присоединения, «+»).

Событий, которые не описывались бы подобным образом, в сетях МакКаллока–Питтса не существует. Это предмет доказательства теоремы Клини.

События, которые могут быть описаны посредством указанных выше операторов, Клини назвал регулярными (в том смысле, что других событий не имеется).

Регулярные события можно описывать выражениями, используя для этого операторы «*», «|» и «+».

Такие выражения получили название «регулярные выражения». Учитывая, что каждая сеть МакКаллока–Питтса представляет собой конечный автомат, мы получили удобный способ описания автоматных языков с помощью регулярных выражений.

1.1.9. Спайковые нейронные сети

Спайковые нейронные сети представляют собой третье поколение нейронных сетей. В отличие от предыдущих для обмена сообщениями, нейроны в данных сетях используют импульсы, или спайки – кратковременное изменение напряжения.

Время задержки между импульсами и частота их прихода представляет собой иной способ кодирования информации, отличающийся от сетей второго и первого поколений. Отвечая на входные стимулы, спайковые нейроны с помощью механизмов синаптической пластичности способны группироваться в ансамбли – таким образом потенциально увеличивается информационная емкость сети, так как информация хранится распределенно, и одни и те же нейроны могут участвовать в различных ансамблях одновременно.

Пример модели спайкового нейрона представлен на рис. 1.6.

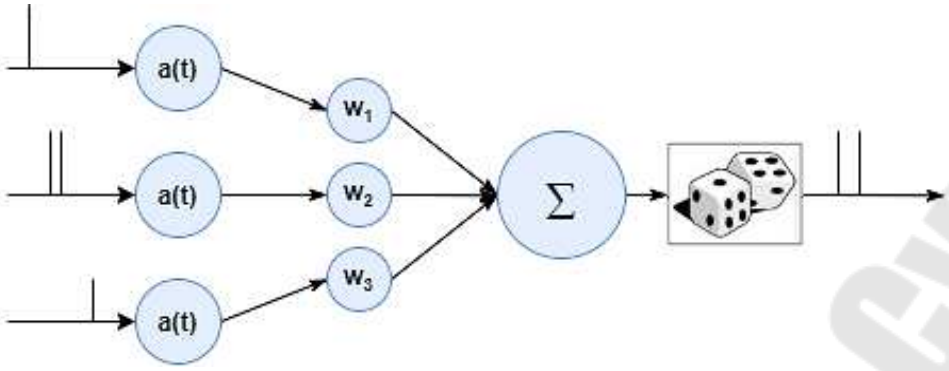


Рис. 1.6. Спайковый нейрон

Существует множество моделей спайкового нейрона:

– модель Ходжкина–Хаксли;

– модель «Интегрировать-и-сработать» (Integrate-and-Fire);

– модель «Интегрировать-и-сработать» с утечками (Leaky Integrate-and-Fire);

– тэта-нейрон и др.

1.2. Персептроны

1.2.1. Персептрон Розенблатта

В 1957 г. американский физиолог Ф. Розенблатт предпринял попытку технически реализовать физиологическую модель восприятия. Он исходил из предположения, что восприятие осуществляется сетью нейронов [4].

Схема персептрона Розенблатта представлена на рис. 1.7.

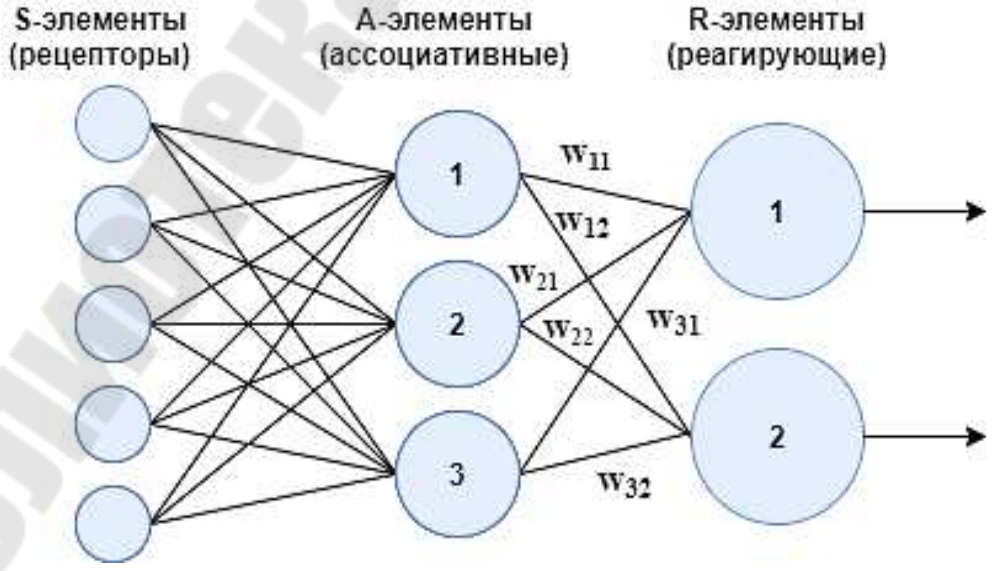


Рис. 1.7. Схема персептрона Розенблатта

Модель восприятия состоит из следующих элементов:

– рецепторного слоя *S*;

– слоя преобразующих нейронов *A*;

– слоя реагирующих нейронов *R*.

Первый слой обеспечивает трансформацию представления задачи из несепарабельного (линейно неразделимого) в сепарабельное. Второй слой разделяет полученную после трансформации линейную задачу.

Перцептроны могут быть классифицированы как искусственные нейронные сети:

– с одним скрытым слоем;

– с пороговой передаточной функцией;

– с прямым распространением сигнала.

Внешнее раздражение воспринимается рецепторами. Каждый рецептор связан с одним или несколькими нейронами преобразующего слоя, при этом каждый нейрон преобразующего слоя может быть связан с несколькими рецепторами.

Выходы преобразующих (ассоциативных) нейронов, в свою очередь, соединяются с входами нейронов третьего слоя.

Нейроны этого слоя – реагирующие – тоже имеют несколько входов (дендритов) и один выход (аксон), который возбуждается, если суммарная величина входных сигналов превосходит порог срабатывания. Но в отличие от нейронов второго слоя, где суммируются сигналы с одним и тем же коэффициентом усиления (но, возможно, разными знаками), для реагирующих нейронов коэффициенты суммирования различны по величине и, возможно, – по знаку.

Каждый рецептор может находиться в одном из двух состояний: возбужденном или невозбужденном. В зависимости от характера внешнего раздражения в рецепторном слое образуется тот или иной набор импульсов, который, распространяясь по нервным путям, достигает слоя преобразующих нейронов. Здесь в соответствии с набором пришедших импульсов формируется набор импульсов второго слоя, который поступает на входы реагирующих нейронов.

Восприятие какого-либо объекта определяется возбуждением соответствующего нейрона третьего слоя, причем различным наборам импульсов рецепторного слоя может соответствовать возбуждение одного и того же реагирующего нейрона.

Гипотеза как раз и состоит в том, что коэффициенты усиления реагирующего нейрона подобраны так, чтобы в случае, когда объекты

принадлежат к одному классу, отвечающие им наборы импульсов возбуждали бы один и тот же нейрон реагирующего слоя.

Алгоритм работы персептрона. Первыми в работу включаются *S*-элементы. Они могут находиться либо в состоянии покоя (сигнал равен 0), либо в состоянии возбуждения (сигнал равен 1).

Далее сигналы от *S*-элементов передаются *A*-элементам по так называемым *S–A* связям. Эти связи могут иметь веса, равные только –1, 0 или 1.

Затем сигналы от сенсорных элементов, прошедших по *S–A* связям, попадают в *A*-элементы, которые еще называют ассоциативными элементами (*одному A-элементу может соответствовать несколько S-элементов*). Если сигналы, поступившие на *A*-элемент, в совокупности превышают некоторый его порог θ, то этот *A*-элемент возбуждается и выдает сигнал, равный 1. В противном случае (сигнал от *S*-элементов не превысил порога *A*-элемента) генерируется нулевой сигнал.

Далее сигналы, которые произвели возбужденные *A*-элементы, направляются к сумматору (*R*-элемент) по *A–R* связям, у которых есть веса. Однако здесь они уже могут принимать любые значения (в отличие от *S–A* связей).

R-элемент складывает друг с другом взвешенные сигналы от *A*-элементов и, если превышен определенный порог, генерирует выходной сигнал, равный 1.

Однослойный персептрон – это персептрон Розенблатта, у которого каждый *S*-элемент однозначно соответствует одному *A*-элементу, все *S–A* связи имеют вес, равный +1, а порог *A*-элементов равен 1 [5].

В персептроне *R*-элементы суммируют взвешенные входные сигналы и, если взвешенная сумма выше некоторого порога, выдают 1. Иначе выходы *R*-элементов были бы равны –1.

В качестве функции активации используется биполярная функция единичного скачка (1.2).

Правила обучения Розенблатта. Процедура обучения Розенблатта называется алгоритмом обучения с подкреплением (метод коррекции ошибки). Она характеризуется тем, что весовые коэффициенты нейронной сети изменяются только в том случае, если выходная реакция сети не совпадает с эталонной.

Существует пять основных модификаций данного метода:

1) метод коррекции ошибок без квантования;

2) метод коррекции ошибок с квантованием;

3) метод коррекции ошибок со случайным знаком подкрепления;

4) метод коррекции ошибок со случайными возмущениями;

5) метод коррекции с обратной передачей сигнала ошибки.

Метод коррекции с обратной передачей сигнала ошибки – стохастический метод обучения персептрона, необходимый, чтобы гарантировать сходимость при переменных связях больше чем у одного слоя.

Метод был предложен Розенблаттом для перцептрона с переменными *S–A* связями и может быть использован для бинарных многослойных перцептронов:

1. Для каждого *R*-элемента устанавливается ошибка $\varepsilon = y_j - \tilde{y}_j$.

2. Для каждого *A*-элемента (A_i) ошибка вычисляется следующим образом:

а) вначале $\varepsilon_i = 0$;

б) если элемент A_i активен и связь c_{ir} оканчивается на *R*-элементе с ненулевой ошибкой ε, отличающейся по знаку от веса связи w_{ir}, то с вероятностью p_1 к ε_i следует прибавить коррекцию, равную –1;

в) если элемент A_i неактивен и связь c_{ir} оканчивается на *R*-элементе с ненулевой ошибкой ε, не отличается (совпадает) по знаку от веса связи w_{ir}, то с вероятностью p_2 к ε_i следует прибавить коррекцию, равную +1;

г) если элемент A_i неактивен и связь c_{ir} оканчивается на *R*-элементе с ненулевой ошибкой ε, отличающейся по знаку от веса связи w_{ir} (или $w_{ir} = 0$), то с вероятностью p_3 к ε_i необходимо прибавить коррекцию, равную +1;

д) при всех остальных условиях ε_i не изменяется.

Если $\varepsilon_i \neq 0$, то ко всем активным связям, оканчивающимся на *A*- или *R*-элементе, прибавляем коррекцию η со знаком, совпадающим со знаком ε_i, т. е.

$$\Delta w_{ij} = \eta A_i \operatorname{sign}(\varepsilon_i). \qquad (1.7)$$

Практическое применение персептрона. Любое практическое применение перцептрона предполагает введение больше одного *R*-элемента.

И, как показано Розенблаттом, характеристики таких систем не отличаются существенно от характеристик элементарного перцептрона, за тем исключением, что теперь оказывается возможным просты-

ми *R*-элементами вырабатывать классификации, состоящие более, чем из двух классов, или обучать перцептрон реагировать одновременно на несколько различных признаков, принадлежащих возбуждающему образу (стимулу).

Важным элементом в применении персептронов и вообще нейронных сетей являются способы кодирования реакций:

– конфигурационное кодирование – способ кодирования реакции, при котором каждой реакции приписывается некоторое двоичное число. Поэтому полное число реакций, которое может быть закодировано таким образом с помощью *N* бит, равно 2^N. Данная реакция может служить для идентификации независимого признака или свойства стимула, такого, как местоположение, размер, вытянутость в горизонтальном или вертикальном направлении;

– позиционное кодирование – способ кодирования реакции, при котором для любого стимула в состоянии «включено» может находиться только один вариант реакции. Код принимает форму двоичного числа, все разряды которого, за исключением одного, состоят из нулей. Положение ненулевого разряда указывает на класс идентифицируемого стимула. С помощью такой системы может быть опознано только *N* типов стимулов, что значительно меньше, чем при конфигурационном кодировании, но зато существенно возрастает вероятность идентификации стимула.

Классы решаемых задач с помощью персептрона:

– классификация – сеть с одним скрытым слоем, содержащим *H* нейронов со ступенчатой функцией активации, способна осуществить произвольную классификацию *Hd*-точек *d*-мерного пространства (т. е. классифицировать *Hd*-примеров). Одного скрытого слоя нейронов с сигмоидной функцией активации достаточно для аппроксимации любой границы между классами со сколь угодно высокой точностью;

– аппроксимация – одного скрытого слоя нейронов с сигмоидной функцией активации достаточно для аппроксимации любой функции со сколь угодно высокой точностью. Более того, такая сеть может одновременно аппроксимировать и саму функцию и ее производные;

– прогнозирование – определение значений целевой функции по «неизвестному» (не использовавшемуся в обучении) значению аргумента;

– управление интеллектуальными агентами – приходится полагаться только на локальную экспертную информацию и в условиях неопределенности выбирать набор действий для агента, которые в долго-

срочной перспективе должны приспособить его к окружающей среде. В таких задачах становится важным анализировать не только текущую информацию, но и общий контекст ситуации, в которую попал агент. Таким образом, становятся необходимы обратные связи, и применение персептронов с обратной связью.

1.2.2. Правило обучения Хебба

Математическая формулировка:

• Есть два класса, помеченных значениями 0 и 1.

• Имеется объект $X_i = (x_1, x_2, \ldots, x_n)$ из обучающей выборки $X^L = (X_1, X_2, \ldots, X_N)$, для которого известен правильный ответ y_i.

• Предъявляем на вход один объект. Если выходной сигнал персептрона совпадает с правильным ответом, то никаких действий предпринимать не надо.

• В случае ошибки необходимо обучить персептрон правильно решать данный пример.

Ошибки могут быть двух типов:

– на выходе 0, а должен быть 1. Для достижения нейроном правильного ответа, необходимо, чтобы скалярное произведение стало больше. Поскольку переменные принимают значения 0 или 1, увеличение суммы может быть достигнуто за счет увеличения весов. Однако нет смысла увеличивать веса при переменных, которые равны нулю. Увеличиваем веса только при тех, которые равны 1;

– на выходе 1, а должен быть 0. Для уменьшения скалярного произведения в правой части, необходимо уменьшить веса связей при тех переменных, которые равны 1. Необходимо также провести эту процедуру для всех активных нейронов предыдущих слоев.

Привило Хебба может быть использовано для обучения в сетях как с учителем, так и без учителя [5].

Таким образом, классические формулы обучения по правилам Хебба выглядят следующим образом:

Первое правило Хебба: если выходной сигнал неверен и равен нулю – нужно увеличить веса всех активных входов, добавив к каждому величину входного сигнала, т. е.

$$w_{ij}(t+1) = w_{ij}(t) + x_i. \qquad (1.8)$$

Второе правило Хебба: если выходной сигнал неверен и равен единице – нужно уменьшить веса всех активных входов, отняв от каждого величину входного сигнала [6], т. е.

$$w_{ij}(t+1) = w_{ij}(t) - x_i. \tag{1.9}$$

На практике пользуются следующими формулами:

$$w_{ij}(t+1) = (1-\gamma)w_{ij}(t) + \eta\Delta w_{ij}, \tag{1.10}$$

где η – коэффициент обучения, $\eta \in (0,1)$; γ – коэффициент забывания, $\gamma \in (0,1)$; $\Delta w_{ij} = \eta x_i y_j$ – для случая биполярного нейрона или

$$\Delta w_{ij} = \begin{cases} 1, \text{ если } x_i y_j = 1, \\ 0, \text{ если } x_i = 0 - \text{ для случая бинарного нейрона,} \\ -1, \text{ если } x_i \neq 0\, y_j = 0. \end{cases}$$

Пример обучения по правилам Хебба

Рассмотрим простейший искусственный нейрон (рис. 1.3). Обучим его для выполнения операции логического «И» (табл. 1.1). В качестве активационной фунции выберем функцию скачка (1.2).

Сформируем обучающую выборку:

$$\begin{aligned} X_0 &= \{(0 \quad 0) \quad 0\}; \\ X_1 &= \{(1 \quad 0) \quad 0\}; \\ X_2 &= \{(0 \quad 1) \quad 0\}; \\ X_3 &= \{(1 \quad 1) \quad 1\}. \end{aligned} \tag{1.11}$$

Таблица 1.1

Логическое «И»

Шаги	x_1	x_2	x_0	s	y	Δw_{11}	Δw_{21}	Δw_{11}	$w_{11}(t)$	$w_{21}(t)$	$w_{01}(t)$
t									0	0	0
t	0	0	1	0	0	0	0	0	0	0	0
t	1	0	1	0	0	0	0	0	0	0	0
t	0	1	1	0	0	0	0	0	0	0	0
$t+1$	1	1	1	0	1	1	1	1	1	1	1
$t+2$	0	0	1	1	0	0	0	–1	1	1	0
$t+3$	1	0	1	1	0	–1	0	–1	0	1	–1
$t+3$	0	1	1	0	0	0	0	0	0	1	–1
…	…										
$t+9$	1	1	1	0	1	1	1	1	2	2	–1
$t+9$	0	0	1	–1	0	0	0	0	2	2	–1
$t+10$	1	0	1	1	0	–1	0	–1	1	2	–2
$t+10$	0	1	1	0	0	0	0	0	1	2	–2

Окончание табл. 1.1

Шаги	x_1	x_2	x_0	s	y	Δw_{11}	Δw_{21}	Δw_{11}	$w_{11}(t)$	$w_{21}(t)$	$w_{01}(t)$
$t+10$	1	1	1	1	1	0	0	0	1	2	–2
$t+10$	0	0	1	–2	0	0	0	0	1	2	–2
$t+10$	1	0	1	–1	0	0	0	0	1	2	–2

Обучим нейрон Хебба для выполнения операции логического «ИЛИ» (табл. 1.2).

Сформируем обучающую выборку:

$$\begin{aligned} X_0 &= \{(0 \quad 0) \quad 0\}; \\ X_1 &= \{(1 \quad 0) \quad 1\}; \\ X_2 &= \{(0 \quad 1) \quad 1\}; \\ X_3 &= \{(1 \quad 1) \quad 1\}. \end{aligned} \tag{1.12}$$

Таблица 1.2

Логическое «ИЛИ»

Шаги	x_1	x_2	x_0	s	y	Δw_{11}	Δw_{21}	Δw_{11}	$w_{11}(t)$	$w_{21}(t)$	$w_{01}(t)$
t									0	0	0
t	0	0	1	0	0	0	0	0	0	0	0
$t+2$	1	0	1	0	1	1	0	1	1	0	1
$t+3$	0	1	1	2	1	0	0	0	1	0	1
$t+3$	1	1	1	2	1	0	0	0	1	0	1
$t+3$	0	0	1	1	0	0	0	–1	1	0	0
…	…										
$t+6$	1	0	1	1	1	0	0	0	1	1	0
$t+6$	0	1	1	1	1	0	0	0	1	1	0
$t+6$	1	1	1	2	1	0	0	0	1	1	0

1.2.3. Дельта-правило

Из обобщения правил Хебба было получено «дельта-правило».

Введем величину δ, которая равна разности между требуемым y и реальным выходом $\tilde{y}$:

$$\delta = y - \tilde{y}. \tag{1.13}$$

Если требуемое значение не равно реальному, то выполняется пересчет весов по формуле

$$w_{ij}(t+1) = w_{ij}(t) + \eta \delta x_i. \tag{1.14}$$

Алгоритм Уидроу–Хоффа. Персептрон ограничивается бинарными выходами. Однако Б. Уидроу вместе со студентом университета М. Хоффом расширили алгоритм обучения персептрона для случая непрерывных выходов, используя сигмоидальную функцию.

Второй их впечатляющий результат – это разработка математического доказательства, что сеть при определенных условиях будет сходиться к любой функции, которую она может представить.

Их первая модель – *адалин* – имеет один выходной нейрон, более поздняя модель – *мадалин* – расширяет ее для случая с многими выходными нейронами.

Выражения, описывающие процесс обучения адалина, очень схожи с персептронными. Основные отличия заключаются в формуле для вычисления ошибки.

Процедура Уидроу–Хоффа разработана применительно к «черному ящику», в котором между входами и выходами существуют только прямые связи. Процедура обучения основывается на минимизации ошибки обучения в процессе подачи на вход сети L входных образов:

$$\frac{1}{2}\sum_{j=1}^{L}\left(y_j - A_j\right)^2 \to \min. \tag{1.15}$$

Процедура заключается в градиентном спуске по настраиваемым параметрам нейронной сети. Такими параметрами являются весовые коэффициенты и пороги сети.

Алгоритм Уидроу–Хоффа:

1. Случайным образом задаются все веса сети w_{ij}.

2. На вход сети подается обучающий вектор $X\left(x_1, x_2, \ldots, x_N\right)$ и вычисляется выходной сигнал от каждого нейрона:

$$A_j = \sum_i x_i w_{ij}. \tag{1.16}$$

3. Для каждого нейрона вычисляется значение пороговой функции активации:

$$f_j\left(A_j\right) = \begin{cases} 1, \text{ если } A_j > \theta_j; \\ 0. \end{cases} \tag{1.17}$$

4. Вычисляется ошибка для каждого нейрона посредством вычитания полученного выхода из требуемого выхода:

$$\varepsilon_j = \frac{1}{2}\left(y_j - A_j\right)^2. \qquad (1.18)$$

5. Каждый вес пересчитывается по формуле

$$w_{ij}(t+1) = w_{ij}(t) + \eta x_i \varepsilon_j. \qquad (1.19)$$

6. Пересчет осуществляется до тех пор, пока ошибка не станет меньше наперед заданной величины.

1.2.4. Стохастическое обучение

Стохастические методы обучения выполняют псевдослучайные изменения величин весов, сохраняя те изменения, которые ведут к улучшениям. Чтобы показать это наглядно, рассмотрим рис. 1.6, на котором изображена типичная сеть, где нейроны соединены с помощью весов.

Выход нейрона является здесь взвешенной суммой его входов, которая преобразована с помощью нелинейной функции. Для обучения сети могут быть использованы следующие процедуры:

1. Выбрать вес случайным образом и подкорректировать его на небольшое случайное число. Предъявить множество входов и вычислить получающиеся выходы.

2. Сравнить эти выходы с желаемыми выходами и вычислить величину разности между ними.

3. Выбрать вес случайным образом и подкорректировать его на небольшое случайное значение. Если коррекция помогает (уменьшает целевую функцию), то сохранить ее, в противном случае вернуться к первоначальному значению веса.

4. Повторять шаги с 1 по 3 до тех пор, пока сеть не будет обучена в достаточной степени.

1.2.5. Многослойный персептрон

Многослойными персептронами (рис. 1.8) называют нейронные сети прямого распространения. Входной сигнал в таких сетях распространяется в прямом направлении, от слоя к слою [5].

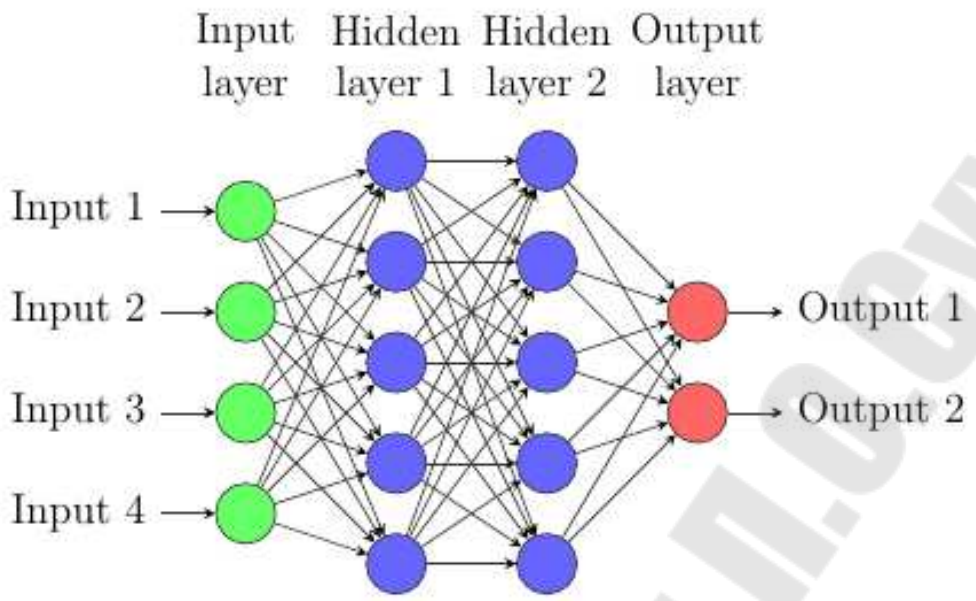


Рис. 1.8. Схема многослойного персептрона

В *многослойных нейронных сетях* [7] нейроны объединяются в слои. Слой содержит совокупность нейронов с едиными входными сигналами. Число нейронов в слое может быть любым и не зависит от количества нейронов в других слоях. В общем случае сеть состоит из Q слоев, пронумерованных слева направо. Внешние входные сигналы подаются на входы нейронов входного слоя (его часто нумеруют как нулевой), а выходами сети являются выходные сигналы последнего слоя. Кроме входного и выходного слоев в многослойной нейронной сети есть один или несколько скрытых слоев. Связи от выходов нейронов некоторого слоя q к входам нейронов следующего слоя (q +1) называются последовательными.

Многослойный персептрон в общем представлении состоит из следующих элементов:

– множества входных узлов, которые образуют входной слой;

– одного или нескольких скрытых слоев вычислительных нейронов;

– одного выходного слоя нейронов.

Многослойный персептрон по Розеблатту – персептрон, у которого имеется более 1 слоя *A*-элементов.

Многослойный персептрон по Румельхарту – многослойный персептрон по Розенблатту, у которого обучению подлежат еще и *S–A* связи, а также само обучение производится по методу *обратного распространения ошибки*.

Свойство 1. Каждый нейрон сети имеет нелинейную функцию активации – нелинейная функция должна быть гладкой (т. е. всюду дифференцируемой) в отличие от жесткой пороговой функции, используемой в персептроне Розенблатта. Самой популярной формой функции, удовлетворяющей этому требованию, является сигмоидальная.

Свойство 2. Несколько скрытых слоев – многослойный персептрон содержит один или несколько слоев скрытых нейронов, не являющихся частью входа или выхода сети. Эти нейроны позволяют сети обучаться решению сложных задач, последовательно извлекая наиболее важные признаки из входного образа.

Высокая связность – многослойный персептрон обладает высокой степенью связности, реализуемой посредством синаптических соединений. Изменение уровня связности сети требует изменения множества синаптических соединений или их весовых коэффициентов.

1.2.6. Проблема линейной неразделимости

Персептроны гарантированно решают задачи классификации линейно разделимых объектов.

Пример линейной разделимости можно рассмотреть применительно к задаче классификации логических функций.

Таблицы истинности логических функций даны в табл. 1.3.

Таблица 1.3

Таблицы истинности логических функций

x_1	x_2	*AND*	*OR*	*XOR*
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Для изображения любой логической функции на плоскости достаточно 4 точки (с координатами (0,0) (1,0) (0,1) (1,1)) (рис. 1.9).

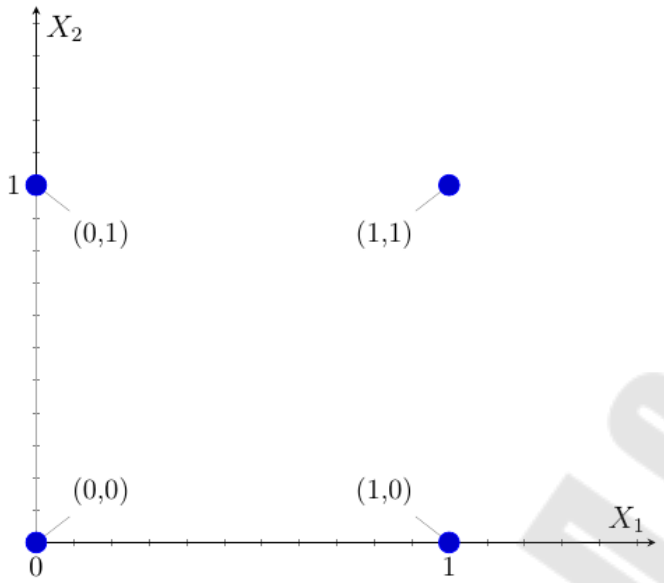


Рис. 1.9. Точки на плоскости для изображения логических функций

Приведем постановку задачи классификации:

– дано 4 точки;

– необходимо провести прямую так, чтобы по одну сторону у нас оказались точки, принадлежащие одному классу (значение функции равно 1), а по другую – второму классу (значение функции равно 0).

График *логического* «*И*» на плоскости представлен на рис. 1.10.

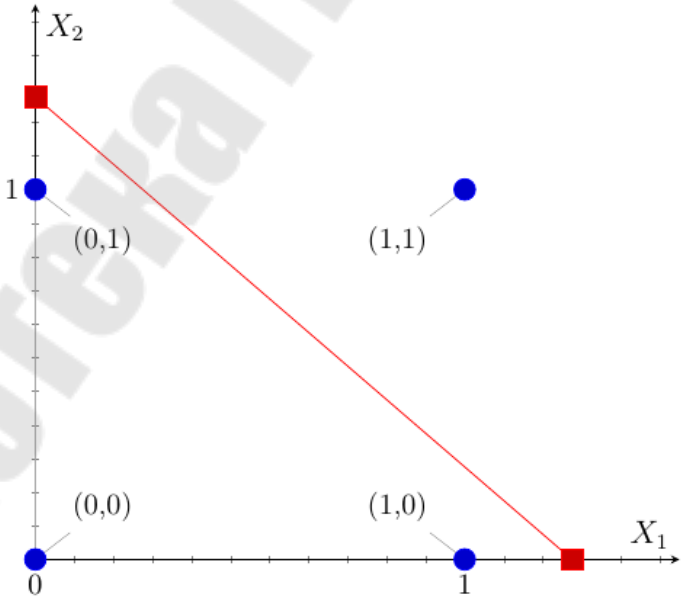


Рис. 1.10. График логического «И» на плоскости

График *логического* «*ИЛИ*» на плоскости дан на рис. 1.11.

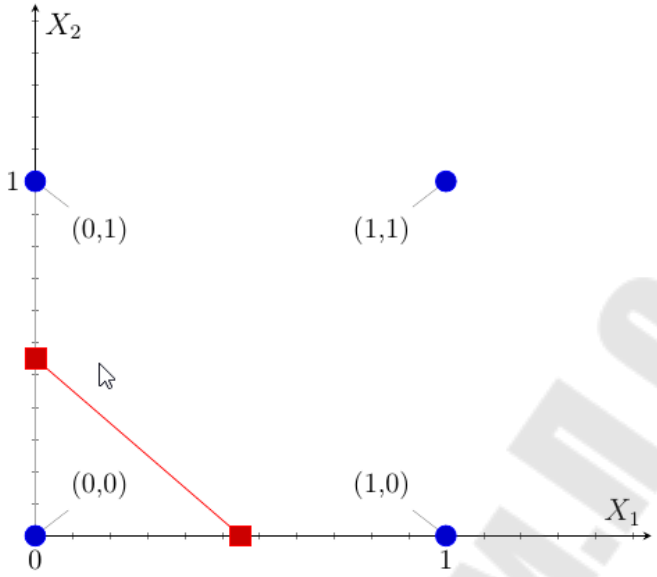


Рис. 1.11. График логического «ИЛИ» на плоскости

График *исключающего* «*ИЛИ*» на плоскости представлен на рис. 1.12.

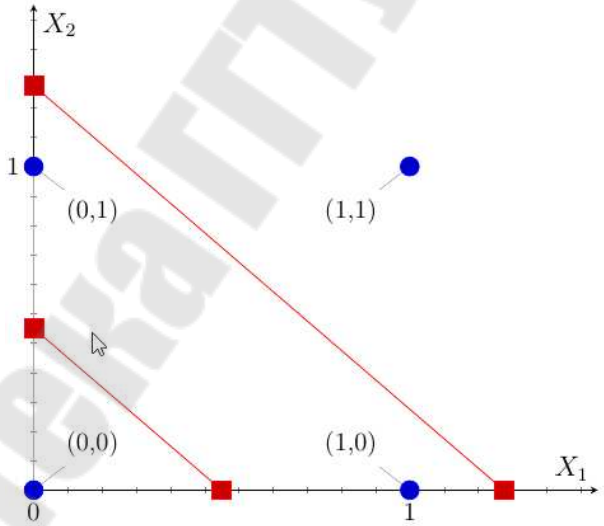


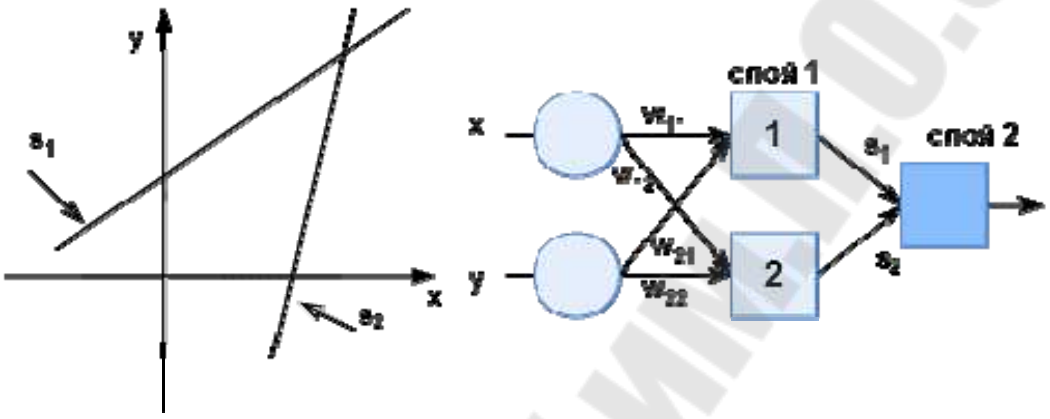
Рис. 1.12. График исключающего «ИЛИ» на плоскости

Двухслойные сети можно получить каскадным соединением двух однослойных сетей. Они способны выполнять более общие классификации, отделяя те точки, которые содержатся в выпуклых ограниченных или неограниченных областях.

Область называется выпуклой, если для любых двух ее точек соединяющий их отрезок целиком лежит в области.

Область называется ограниченной, если существует некоторое конечное положительное число $R > 0$ – такое, что окружность, проведенная из центра координат радиуса R, будет содержать все точки области.

Рассмотрим простую двухслойную сеть с двумя входами, подведенными к двум нейронам первого слоя, соединенными с единственным нейроном в слое *2* (рис. 1.13).



а) *б*)

Рис. 1.13. Изображение области, определяемой трехнейронным персептроном (*а*); трехнейронный персептрон (*б*)

В этом случае для того чтобы порог был превышен и на выходе появилась единица, требуется, чтобы оба нейрона первого уровня на выходе имели единицу. Таким образом, выходной нейрон реализует логическую функцию И.

Каждый нейрон слоя *1* разбивает плоскость *XY* на две полуплоскости, один обеспечивает единичный выход для входов ниже верхней линии, другой – для входов выше нижней линии.

Выходной сигнал нейрона второго слоя равен единице только внутри *V*-образной области.

Аналогично во втором слое может быть использовано три нейрона с дальнейшим разбиением плоскости и созданием области треугольной формы.

Включением достаточного числа нейронов во входной слой может быть образован выпуклый многоугольник любой желаемой формы. Так как они образованы с помощью операции И над областями, задаваемыми линиями, то все такие многогранники выпуклы, следовательно, только выпуклые области и возникают.

Точки, не составляющие выпуклой области, не могут быть отделены от других точек плоскости двухслойной сетью.

Таким образом, для решения задачи классификации в случае *исключающего* «*ИЛИ*» можно использовать трехнейронный персептрон (рис. 1.2), обучая который согласно дельта-правилу, можно найти следующие веса: $w_{11} = w_{22} = 0{,}5$; $w_{12} = w_{21} = -0{,}5$; $w_{31} = w_{32} = 1$.

Входы не обязательно должны быть двоичными.

Вектор непрерывных входов может представлять собой произвольную точку на плоскости *XY*. В этом случае мы имеем дело со способностью сети разбивать плоскость на непрерывные области.

Линейная разделимость показывает, что выход нейрона второго слоя равен единице только в части плоскости *XY*, ограниченной многоугольной областью.

Поэтому для разделения плоскостей *P* и *Q* необходимо, чтобы все *P* лежали внутри выпуклой многоугольной области, не содержащей точек *Q* (или наоборот).

Классифицирующие возможности ***трехслойной сети*** (рис. 1.14) ограничены лишь числом искусственных нейронов и весов. Ограничения на выпуклость отсутствуют. Теперь нейрон третьего слоя принимает в качестве входа набор выпуклых многоугольников, и их логическая комбинация может быть невыпуклой. Это позволяет аппроксимировать область любой формы с любой точностью. Вдобавок не все выходные области второго слоя должны пересекаться. Следовательно, возможно объединять различные области, выпуклые и невыпуклые, выдавая на выходе единицу всякий раз, когда входной вектор принадлежит одной из них.

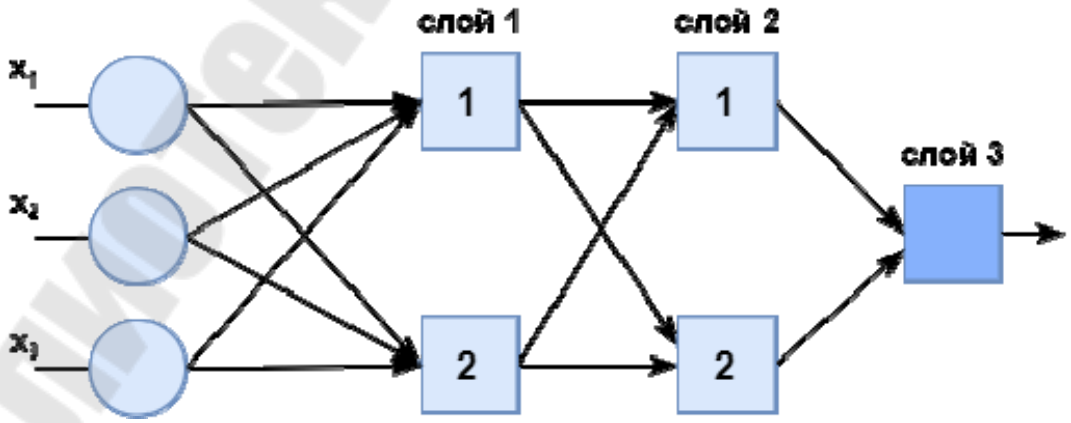


Рис. 1.14. Трехслойная нейронная сеть

1.3. Контролируемое обучение нейронных сетей

1.3.1. Алгоритмы обучения нейронных сетей

Обучение нейронной сети – это процесс настройки структуры связей между отдельными нейронами и синоптическими связями, которые влияют на сигналы коэффициентов.

Различают три парадигмы обучения:

– с учителем;

– без учителя;

– смешанное обучение.

Обучение с учителем предполагает, что для каждого входного вектора существует целевой вектор, представляющий собой требуемый выход. Вместе они называются обучающей парой [5]. Обычно сеть обучается на некотором числе таких обучающих пар. Предъявляется выходной вектор, вычисляется выход сети и сравнивается с соответствующим целевым вектором. Далее веса изменяются в соответствии с алгоритмом, стремящимся минимизировать ошибку. Векторы обучающего множества предъявляются последовательно, вычисляются ошибки и веса подстраиваются для каждого вектора до тех пор, пока ошибка по всему обучающему массиву не достигнет приемлемого уровня:

$$F\left(x_i^{train}\right) - y_i^{train} \to \min. \qquad (1.20)$$

Сети, обучающиеся с учителем, просматривают выборку множество раз, при этом один полный проход по выборке называется *эпохой обучения*.

При обучении с учителем набор исходных данных делят на две части – собственно *обучающую выборку* и *тестовые данные*; принцип разделения может быть произвольным.

Обучающие данные подаются сети для обучения, а проверочные используются для расчета ошибки сети (проверочные данные никогда для обучения сети не применяются). Таким образом, если на проверочных данных ошибка уменьшается, то сеть действительно выполняет обобщение.

Если ошибка на обучающих данных продолжает уменьшаться, а ошибка на тестовых данных увеличивается, значит, сеть перестала выполнять обобщение и просто «запоминает» обучающие данные. Это явление называется *переобучением сети*. В таких случаях обучение обычно прекращают.

Для борьбы с переобучением набор для тренировки классификатора разделяют на три части – ***учебный***, ***контрольный***, ***тестовый***:

– учебный набор применяют для корректировки параметров;

– контрольный – для текущей оценки состояния;

– тестовый – для оценки итогового результата обучения.

Для улучшения результатов можно использовать различные стратегии коррекции параметров: полную, частичную или стохастическую, т. е. на каждом цикле обучения (эпохе) можно обрабатывать не все примеры а только их часть.

Полная (***full batch***) – на каждом цикле прогоняется все учебное множество:

1) обрабатываем учебный набор;

2) определяем ошибки;

3) корректируем параметры;

4) проверка на контрольном множестве;

5) если результат удовлетворительный, то выполняем итоговый тест и конец работы;

6) переход на п. 1.

Частичная (***mini batch***) – на каждом цикле используется случайно выбранное подмножество учебного набора:

1) случайным образом выбираем подмножество учебного набора;

2) обрабатываем эту часть учебного набора;

3) определяем ошибки;

4) корректируем параметры;

5) проверка на контрольном множестве;

6) если результат удовлетворительный, то выполняем итоговый тест и конец работы;

7) переход на п. 1.

Стохастическая (***stochastic***) – на каждом цикле случайно выбирается один пример:

1) случайным образом выбираем пример из учебного набора;

2) обрабатываем этот пример;

3) определяем ошибку;

4) сохраняем текущее состояние параметров;

5) определяем новые параметры;

6) проверка на контрольном множестве;

7) если результат удовлетворительный, то выполняем итоговый тест и конец работы;

8) если ошибка выросла, то с вероятностью, пропорциональной величине прироста ошибки, выполняется откат к предыдущему состоянию параметров;

9) переход на п. 1.

Большинство методов обучения строится на идее поиска минимума вектора функции:

$$\delta\left(F\left(x^{train}\right), y^{train}\right) \to \min. \tag{1.21}$$

Причем фактически x^{train} и y^{train} представляют собой матрицы с количеством строк, равным количеству обучающих примеров, а количество столбцов равняется количеству входных и выходных нейронов соответственно; δ – функция потерь (может еще обозначаться как E).

Функция потерь:

$$h: \; X \times W \to Y; \tag{1.22}$$

$$\delta: \; Y \times \tilde{Y} \to \; \mathbb{R}, \tag{1.23}$$

где h – классификатор; W – веса сети; $X\left(x_1, x_2, \ldots, x_N\right)$ – вектор признаков размерности N; δ – функция потери; Y – выход классификатора; $\tilde{Y}$ – множество правильных ответов (номеров классов).

В качестве функции потери для нейронных сетей обычно используется среднеквадратичная ошибка (*MSQE*):

$$\delta = \frac{1}{2}\sum_{j=1}^{N}\left(y_j - \tilde{y}_j\right)^2. \tag{1.24}$$

Но *MSQE* – это не единственный вариант, для сетей с выходным слоем *softmax* обычно используют среднюю кросс-энтропию по всем учебным примерам:

$$\delta = \frac{1}{K}\sum_{j=1}^{K}\left(-\log\left(p_j\right)\right)^2, \tag{1.25}$$

где K – количество примеров; p_j – выданная классификатором вероятность принадлежности примера X_j к своему классу $\tilde{Y}_j$.

Математически задачу обучения можно описать следующим образом:

$$\min_W \delta\left(h\left(X, W\right), \tilde{Y}\right). \tag{1.26}$$

1.3.2. Градиентные методы обучения нейронных сетей

Для решения задачи используются градиентные методы оптимизации (первого порядка):

$$\min_{W} \delta\left(h(X,W),\tilde{Y}\right). \tag{1.27}$$

Градиент функции $\nabla\delta(X)$ в точке X – это направление наискорейшего ее возрастания.

Соответственно, для минимизации функции необходимо изменять параметры в сторону противоположную градиенту. Этот подход называется методом градиентного спуска.

Общая схема такого обучения выглядит следующим образом:

1) инициализируем веса W (случайными малыми значениями);

2) вычисляем ошибку $\delta\left(h(X,W),\tilde{Y}\right)$;

3) если результат удовлетворительный, то конец работы;

4) вычисляем значение градиента функции потери: $\nabla\delta\left(h(X,W),\tilde{Y}\right)$;

5) вычисляем изменение параметров: $\Delta W = \eta\nabla\delta$;

6) корректируем параметры: $W = W - \Delta W$;

7) переход на п. 2, где η – скорость обучения; $0 < \eta < 1$.

На практике начинают с достаточно большой η, а затем ее постепенно уменьшают.

1.3.3. Метод обратного распространения ошибки

Для случая нейронной сети градиент можно записать следующим образом:

$$\frac{\partial\delta}{\partial w_{ij}} = \frac{\partial\delta}{\partial y_j}\frac{\partial y_j}{\partial s_j}\frac{\partial s_j}{\partial w_{ij}}, \tag{1.28}$$

где δ – функция потери; w_{ij} – вес связи i нейрона с j; y_j – выход нейрона j; s_j – состояние нейрона j.

Значения ошибки определены явно только для нейронов выходного слоя.

Для определения ошибок скрытых слоев используется *метод обратного распространения ошибки* [8].

Суть его заключается в последовательном вычислении ошибок скрытых слоев с помощью значений ошибки выходного слоя, т. е. значения ошибки распространяются по сети в обратном направлении от выхода к входу.

Проведя вычисления, можно получить:

– для выходного слоя:

$$\delta_i = \frac{\partial \delta}{\partial y_j}; \tag{1.29}$$

– для скрытого слоя:

$$\delta_i = \frac{\partial y_i}{\partial s_i} \sum_j \delta_j w_{ij}. \tag{1.30}$$

Алгоритм метода выполняется за два прохода:

• прямой проход:

– вычисляем состояния нейронов s всех слоев и выход сети;

– определяем значения δ для выходного слоя;

• обратный проход:

– последовательно от конца к началу для всех скрытых слоев вычисляем δ;

– для каждого слоя определяем значение градиента $\nabla\delta$.

Вариации градиентного спуска первого порядка. Метод градиентного спуска в «чистом» виде может «застревать» в локальных минимумах функции потери.

Для борьбы с этим с одной стороны применяется стратегия *mini-batch*, а с другой стороны используются так называемые «моменты» и «регуляризация».

Метод моментов можно сравнить с поведением тяжелого шарика, который скатываясь по склону в ближайшую низину, некоторое расстояние способен двигаться вверх по инерции, выбираясь таким образом из локальных минимумов.

Формально это выглядит как добавка для изменения весов сети.

Регуляризация «накладывает штраф» на чрезмерный рост значений весов.

Алгоритм метода градиентного спуска выполняется следующим образом:

1) инициализировать веса W (случайными малыми значениями);

2) инициализировать нулями начальные значение изменения весов ΔW;

3) вычисляем ошибку $\delta\left(h\left(X,W\right),\widetilde{Y}\right)$ и ее изменение $\Delta\delta$ на контрольном наборе;

4) если результат удовлетворительный, то выполняем итоговый тест и конец работы;

5) случайным образом выбираем подмножество $\left(X,\widetilde{Y}\right)_L$ из учебного набора;

6) определяем значение градиента функции потери $\nabla\delta$ на выбранном подмножестве

7) вычисляем изменение параметров: $\Delta W = \eta(\nabla + \rho W) + \mu\Delta W$;

8) корректируем параметры: $W = W - \Delta W$;

9) корректируем скорость обучения η;

10) переход на п. 3, где μ – коэффициент момента; ρ – коэффициент регуляризации.

Метод quickProp – параметр момента μ и коэффициент скорости обучения η задаются индивидуально для каждого параметра:

$$\Delta W = \eta(\nabla + \rho W) + \mu\Delta W. \tag{1.31}$$

Метод rProp (*resilient back propagation*) – используется стратегия *full-batch*, а параметр скорости обучения η *рассчитывается для каждого веса индивидуально*:

$$\Delta W_t = \eta\left(\text{sign}\left(\nabla\delta\right) + \rho W_{t-1}\right) + \mu W_{t-1}. \tag{1.32}$$

Метод сопряженных градиентов (*conjugate gradient*) – основная идея заключается в специальном выборе направления изменения параметров. Оно выбирается таким образом, чтобы было ортогональным к предыдущим направлениям:

$$\Delta W_t = \eta\left(p + \rho W_{t-1}\right) + \mu\Delta W_{t-1}, \tag{1.33}$$

где p – направление изменения параметров.

Коэффициент скорости обучения η, выбирается на каждой итерации путем решения задачи оптимизации:

$$\min_{\eta}\delta\left(\Delta W\left(\eta\right)\right). \tag{1.34}$$

Направление изменения параметров выбирается следующим образом:

• Начальное направление выбирается как $p_0 = \nabla\delta$.

• Ключевым моментом является вычисление коэффициента сопряжения β. Его обычно вычисляют либо по формуле Флетчера–Ривса или Полака–Рибьера:

$$\beta = \frac{\nabla\delta_t^T \nabla\delta_t}{\nabla\delta_{t-1}^T \nabla\delta_{t-1}}, \text{ или } \beta = \frac{\nabla\delta_t^T (\nabla\delta_t - \nabla\delta_{t-1})}{\nabla\delta_{t-1}^T \nabla\delta_{t-1}}. \quad (1.35)$$

• Для компенсации накапливающейся погрешности вычислений метод предусматривает сброс сопряженного направления через каждые *n* циклов, где *n* выбирается в зависимости от количества параметров *W*.

Метод NAG (*Nesterov's Accelerated Gradient*) – градиент вычисляется относительно сдвинутых на значение момента весов:

$$\Delta W_t = \eta(\nabla\delta(W_{t-1} + \mu\Delta W_{t-1}) + \rho W_{t-1}) + \mu\,\Delta W_{t-1}. \quad (1.36)$$

Метод AdaGrad (*Adaptive Gradient*) – учитывает историю значений градиента следующим образом [9]:

$$g_t = \frac{\nabla\delta_t}{\sqrt{\sum_{i=1}^{t} \nabla\delta_i^2}}. \quad (1.37)$$

Метод AdaDelta учитывает историю значений градиента и историю изменения весов:

$$S_t = \alpha S_{t-1} + (1-\alpha)\nabla\delta_t^2;\ S_0 = 0; \quad (1.38)$$

$$D_t = \beta D_{t-1} + (1-\beta)\Delta W_{t-1}^2;\ D_0 = 0; \quad (1.39)$$

$$g_t = \frac{\sqrt{D_t}}{\sqrt{S_t}}\nabla\delta_t; \quad (1.40)$$

$$\Delta W_t = \eta(g_t + \rho W_{t-1}) + \mu\,\Delta W_{t-1}; \quad (1.41)$$

$$\alpha = \beta = 0{,}9. \quad (1.42)$$

Метод Adam:

$$S_t = \alpha S_{t-1} + (1-\alpha)\nabla\delta_t^2;\ S_0 = 0; \quad (1.43)$$

$$D_t = \beta D_{t-1} + (1-\beta)\nabla\delta_t;\ D_0 = 0; \quad (1.44)$$

$$g_t = \frac{D_t}{1-\beta}\sqrt{\frac{1-\alpha}{S_t}}; \tag{1.45}$$

$$\Delta W_t = \eta\left(g_t + \rho W_{t-1}\right) + \mu\,\Delta W_{t-1}; \tag{1.46}$$

$$\alpha = 0{,}999;\ \ \beta = 0{,}9. \tag{1.47}$$

1.3.4. Градиентные методы второго порядка

Градиентные методы первого порядка для решения задачи используют только вектор градиента $\nabla\delta$, т. е. направления набольшего роста целевой функции.

В отличие от них, градиентные методы второго порядка помимо вектора градиента используют еще и информацию о кривизне целевой функции, для этого используется гессиан H – матрица вторых производных целевой функции:

$$\Delta W = H^{-1} g; \tag{1.48}$$

где $g = \nabla\delta = \begin{bmatrix} \dfrac{\partial\delta}{\partial w_1} \\ \vdots \\ \dfrac{\partial\delta}{\partial w_n} \end{bmatrix}$; $H = \begin{bmatrix} \dfrac{\partial^2\delta}{\partial w_1 \partial w_1} & \cdots & \dfrac{\partial^2\delta}{\partial w_1 \partial w_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2\delta}{\partial w_n \partial w_1} & \cdots & \dfrac{\partial^2\delta}{\partial w_n \partial w_n} \end{bmatrix}$.

Вычисление гессиана H является затратной и не всегда явно алгоритмически реализуемой процедурой, поэтому на практике вместо гессиана H вычислется его приближение, исходя из значений первой производной функции потери и изменения параметров ΔW.

Такой подход известен как *квазиньютоновские* методы.

Метод BFGS или алгоритм Бройдена–Флетчера–Гольдфарба–Шанно (*Broyden–Fletcher–Goldfarb–Shanno*) для вычисления обратного гессиана H^{-1} использует изменение значений градиента $\nabla\delta$ и изменения весов ΔW. Вектор градиента функции ошибки $g = \nabla\delta$ вычисляется с помощью обычной процедуры обратного распространения ошибки.

Обратный гессиан $H^{-1} \approx V_0 = 1$ – это матрица размера $n \times n$, где n – длина вектора градиента g.

Значения V вычисляются на каждом шаге алгоритма следующим образом:

$$V_0 = 1;$$

$$V_{k+1} = V_k - \frac{V_k s s^T V_k}{s^T V_k s} + \frac{r r^T}{s^T s}, \tag{1.49}$$

где $r = \Delta g_k = g_k - g_{k-1}$ – изменение градиента; $s = \Delta W_k = W_k - W_{k-1}$ – изменение весов.

Алгоритм *BFGS*:

1) инициализировать веса W (случайными малыми значениями) и задать начальное значение приближения обратного гессиана $H^{-1} \approx V_0 = 1$;

2) вычислить значение градиента g;

3) скорректировать веса:

$$\Delta W = g\tau; \tag{1.50}$$

$$W = W - \Delta W, \tag{1.51}$$

где $\tau = 0{,}01$ – параметр скорости обучения;

4) сохраняем старое значение градиента $g_{old} = g$ и вычисляем новое значение градиента $g(W)$ и изменение градиента:

$$\Delta g = g - g_{old}; \tag{1.52}$$

5) вычисляем приближенное значение обратного гессиана:

$$V(\Delta g, \Delta W); \tag{1.53}$$

6) вычисляем изменение весов и корректируем параметры:

$$\Delta W = Vg; \tag{1.54}$$

$$W = W - \Delta W; \tag{1.55}$$

7) вычисляем ошибку $\delta(W)$;

8) если результат $\delta(W)$ удовлетворительный, то вычисления заканчиваются;

9) переход на п. 4.

***Метод Левенберга–Марквардта* (*LMA*)**

Рассмотрим функцию ошибки сети на всей учебной выборке:

$$\delta = d - 0 = \begin{bmatrix} \delta_{11} \\ \vdots \\ \delta_{M1} \\ \delta_{12} \\ \vdots \\ \delta_{MP} \end{bmatrix}, \tag{1.56}$$

где 0 – выход сети; d – учебный (идеальный) выход; M – количество выходов; P – количество примеров.

Метод Левенберга–Марквардта вычисляет приближение гессиана H с помощью якобиана J (матрицы первых производных функции ошибки δ) следующим образом:

$$H \approx J^T J + \mu I, \tag{1.57}$$

где J – якобиан, матрица первых производных функции ошибки; μ – параметр; $I = (J^T J) \circ E$ – диагональная матрица из элементов главной диагонали $(J^T J)$; $\circ$ – поэлементное умножение матриц (*Hadamard product*).

Вектор градиента вычисляется следующим образом:

$$g = J^T \delta. \tag{1.58}$$

Если собрать все вместе, то получаем следующую формулу для изменения весов сети:

$$g = \left(J^T J + \mu I\right)^{-1} J^T \delta. \tag{1.59}$$

Здесь якобиан J или матрица частных производных функции ошибки сети по всем ее параметрам и на всей учебной выборке, ее размер:

$$(PM)T, \tag{1.60}$$

где P – количество примеров; M – количество выходов сети; T – количество параметров сети.

Матрица якобиана J состоит из следующих блоков:

$$J = \left(\frac{\partial \delta}{\partial w}\right) = \begin{bmatrix} J_1 \\ \vdots \\ J_p \end{bmatrix}, \tag{1.61}$$

где J_p – часть якобиана для примера p, она включает следующие части:

$$J_p = \begin{bmatrix} J_p^1 & \dots & J_p^L \end{bmatrix}, \tag{1.62}$$

где J_p^l – часть якобиана для примера p слоя сети l, состоящая из следующих компонентов:

$$J_p^l = \begin{bmatrix} \frac{\partial \delta_1(p)}{\partial W^l} \\ \vdots \\ \frac{\partial \delta_M(p)}{\partial W^l} \end{bmatrix}, \tag{1.63}$$

где $\frac{\partial \delta_m(p)}{\partial W^l}$ – часть якобиана для примера p слоя сети l выхода сети m;

$$\frac{\partial \delta_m(p)}{\partial W^l} = \begin{bmatrix} \frac{\partial \delta_m(p)}{\partial w_{11}^l} & \dots & \frac{\partial \delta_m(p)}{\partial w_{1k}^l} \frac{\partial \delta_m(p)}{\partial w_{21}^l} & \dots & \frac{\partial \delta_m(p)}{\partial w_{tk}^l} \end{bmatrix}, \tag{1.64}$$

т. е. это есть развернутая в строку матрица для слоя l:

$$\begin{bmatrix} \frac{\partial \delta_m(p)}{\partial w_{11}^l} & \dots & \frac{\partial \delta_m(p)}{\partial w_{1k}^l} \\ \vdots & \ddots & \vdots \\ \frac{\partial \delta_m(p)}{\partial w_{t1}^l} & \dots & \frac{\partial \delta_m(p)}{\partial w_{tk}^l} \end{bmatrix}. \tag{1.65}$$

При построении якобиана J элементы J для весов и сдвигов компонуются отдельно:

$$J = \begin{bmatrix} \frac{\partial \delta}{\partial W} & \frac{\partial \delta}{\partial B} \end{bmatrix}. \tag{1.66}$$

Процедура вычисления элементов якобиана J для функции ошибки δ нейронной сети похожа на метод обратного распространения ошибки.

Определим состояние нейрона *s* и выход нейрона *y* следующим образом.

Состояние нейрона *j* слоя *l* на примере *p*:

$$s_{j(l)}(p) = \sum_i w_{ji}^l y_{i(l-1)}(p). \quad (1.67)$$

Активированный *f*( ) выход нейрона нейрона *j* слоя *l* на примере *pp*:

$$y_{j(l)}(p) = f\left(s_{j(l)}(p)\right), \quad (1.68)$$

Элемент якобиана – это значение производной функции ошибки δ для примера p, выхода *m*, веса w_{ji} слоя *l*:

$$\frac{\partial \delta_m(p)}{\partial w_{ji}^l} = -\frac{\partial 0_m(p)}{\partial y_{j(l)}} \frac{\partial y_{j(l)}(p)}{\partial s_{j(l)}} \frac{\partial s_{j(l)}(p)}{\partial w_{ji}^l}. \quad (1.69)$$

Последняя часть – вход *i* нейрона *j* слоя *l* на примере *p*:

$$\frac{\partial s_{j(l)}(p)}{\partial w_{ji}^l} = y_{i(l-1)}(p). \quad (1.70)$$

Эта часть определена.

Вторая часть – значение производной функции активации для нейрона *j* слоя *l* на примере *p*:

$$\frac{\partial y_{j(l)}(p)}{\partial s_{j(l)}} = \partial f_{j(l)}(p). \quad (1.71)$$

Эту часть можно вычислить.

Первая часть – производная нелинейной функции, описывающей преобразование сигнала между выходом нейрона *j* слоя *l* и выходом сети *m* на примере *p*:

$$\frac{\partial 0_m(p)}{\partial y_{j(l)}} = \partial F_{mj(l)}(p). \quad (1.72)$$

Таким образом, элемент якобиана для выхода *m*, веса w_{ji} слоя *l* на примере *p* можно записать как

$$\frac{\partial \delta_m(p)}{\partial w_{ji}^l} = -F_{mj(l)}(p) \partial f_{j(l)}(p) y_{i(l-1)}(p), \quad (1.73)$$

при этом – для сдвига $y_{i(l-1)}(p)=1,$ т. е. вход сдвига нейрона j всегда равен 1:

$$\frac{\partial \delta_m(p)}{\partial w_{ji}^l} = -F_{mj(l)}(p)\partial f_{j(l)}(p). \tag{1.74}$$

LMA backProp

Введем ε для каждого нейрона jj слоя l выхода сети m на примере p:

$$\varepsilon_{mj(l)}(p) = -\partial F_{mj(l)}(p)\partial f_{j(l)}(p), \tag{1.75}$$

где $\partial f_{j(l)}(p)$ – значение производной функции активации для нейрона j слоя l на примере p.

Для нейрона j выходного слоя, выхода сети m и примера p значение $\varepsilon_{mj}(p)$ определяется следующим образом:

$$\varepsilon_{mi}(p) = \begin{cases} -\partial f_{j(l)}(p), & m = j; \\ 0, & -. \end{cases} \tag{1.76}$$

Здесь $\varepsilon_{mi}(p)$ – это P диагональных матриц размера $M \times M$, где P – количество примеров; M – количество выходов.

Для нейрона k скрытого слоя l, выхода сети m и примера p значение $\varepsilon_{mk(l)}(p)$ находится следующим образом:

$$\varepsilon_{mk(l)}(p) = \sum_{i(l+1)} \left(w_{k(l)i}^{l+1}\Delta\varepsilon_{mi(l)}(p)\right)\Delta\partial f_{k(l)}(p), \tag{1.77}$$

в матричном виде (для каждого примера p):

$$\varepsilon^l(p) = \varepsilon^{l+1}(p)W^l \circ \left(\partial f^l(p)\right)^T. \tag{1.78}$$

В данном случае $\varepsilon_{mj}(p)$ – это P матриц размера $M \times K$, где P – количество примеров; K – количествово нейронов в слое l; $\circ$ – поэлементное умножение матриц (*Hadamard product*).

Для выходного слоя с состоянием нейронов s и активацией $softmax(s)$ значение $\delta_{mj}(p)$ определяется по формуле

$$\mathrm{softmax}(s) = \frac{\exp(s_i)}{\sum_j \exp(s_j)}; \tag{1.79}$$

$$j(p)=\begin{cases}-s_j(p)\big(1-s_j(p)\big); & m=j;\\ s_m(p)s_j(p). & \end{cases} \tag{1.80}$$

Таким образом, значения элементов якобиана имеют следующий вид:

$$\frac{\partial\delta_m(p)}{\partial w_{ji}^l}=\varepsilon_{mk(l)}(p)y_{i(l-1)}(p). \tag{1.81}$$

Алгоритм *LMA backProp* выполняется таким образом:

1) инициализируем веса W (случайными малыми значениями) и параметр μ;

2) вычисляем значение якобиана J;

3) находим изменение параметров: $\Delta W(J,\mu)$;

4) корректируем параметры: $Wnew = W - \Delta W$;

5) вычисляем ошибку $(Wnew)$;

6) если ошибка выросла, то увеличиваем параметр $\mu = \mu \cdot 10$ и переход на п. 3;

7) сохраняем результат $W = Wnew$ уменьшаем параметр $\mu = \mu/10$;

8) если результат $\delta(Wnew)$ удовлетворительный, то завершаем вычисления;

9) переход на п. 2.

1.4. Нейросетевая аппроксимация и моделирование

1.4.1. Аппроксимация экспериментальных данных

Аппроксимацией называется получение некой функции, приближенно описывающей какую-то функциональную зависимость $f(x)$, заданную таблицей значений, либо заданную в неудобном для вычислений виде.

Согласно *универсальной теореме аппроксимации* нейронная сеть с одним скрытым слоем может аппроксимировать любую непрерывную функцию многих переменных с любой точностью. Главное чтобы в этой сети было достаточное количество нейронов. И еще важно удачно подобрать начальные значения весов нейронов. Чем удачнее будут подобраны веса, тем быстрее нейронная сеть будет сходиться к исходной функции.

Универсальную теорему аппроксимации можно рассматривать как естественное расширение теоремы Вейерштрасса. Эта теорема утверждает, что любая непрерывная функция на замкнутом интервале действительной оси может быть представлена абсолютно и равномерно сходящимся рядом полиномов.

Сеть имеет два входа и один выход:

– первый вход x – это значения функции, которые она может принимать по оси x;

– второй вход – это сдвиг или смещение.

Сеть имеет два слоя – скрытый и выходной. Количество нейронов в скрытом слое управляет «качеством» аппроксимации. Во втором слое всегда один нейрон (рис. 1.15).

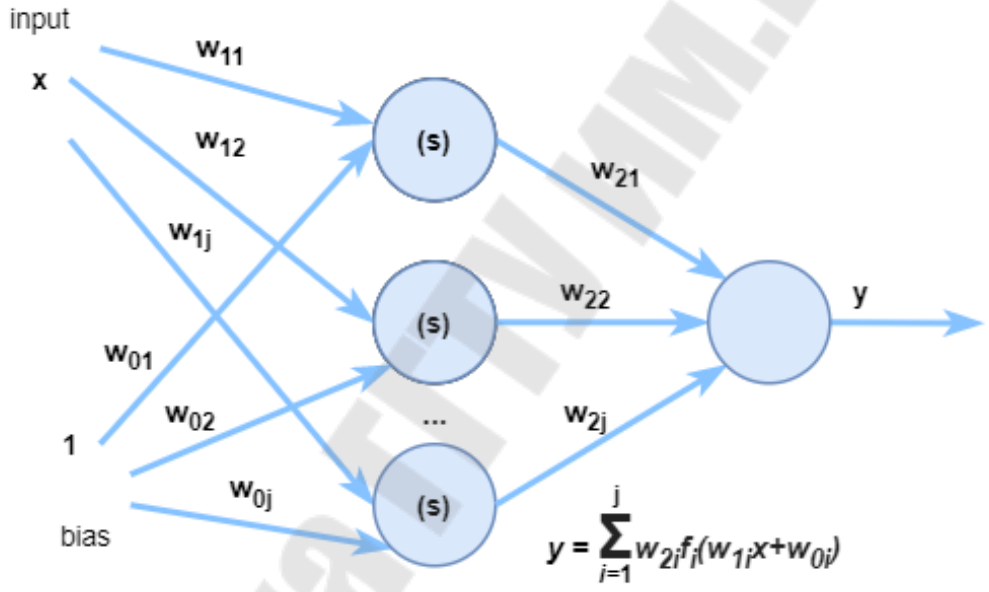


Рис. 1.15. Однослойная нейронная сеть

В нейронах скрытого слоя применяется активационная функция – это может быть и выпрямитель «*ReLU*» $\left(f(x)=\max(0,x)\right)$, и «сигмоид» (1.4), и гиперболический тангенс (1.5).

В нейроне второго слоя функции активации нет, а есть обычный сумматор. Если бы там тоже была функция активации, то мы бы уже не смогли построить график функции нейронной сети.

В общем случае под радиальной базисной нейронной сетью (*Radial Basis Function Network, сеть RBF*) понимается двухслойная сеть без обратных связей, которая содержит скрытый слой радиально симметричных нейронов (рис. 1.16).

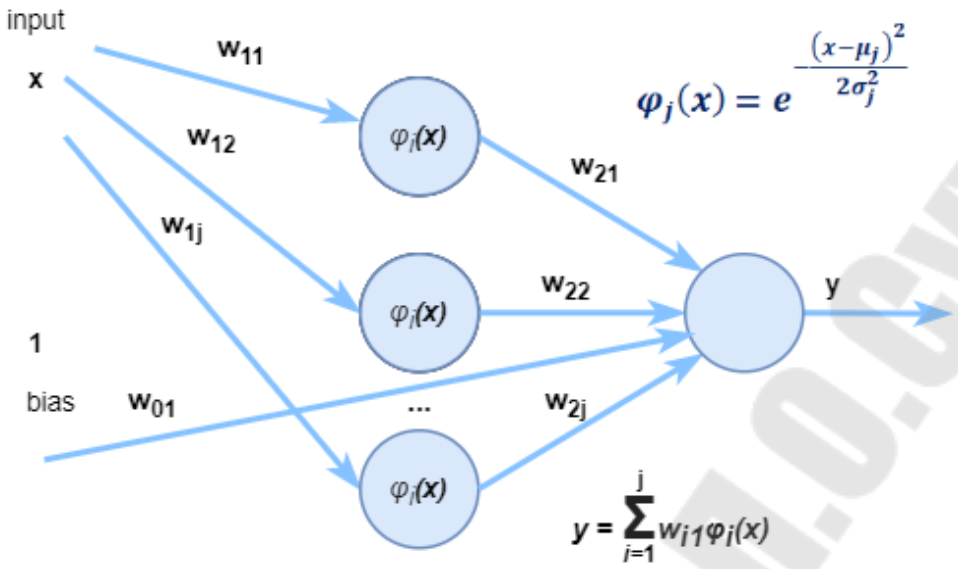


Рис. 1.16. Радиально-базисная сеть

Радиальные базисные нейронные сети состоят из бóльшего количества нейронов, чем стандартные сети с прямой передачей сигналов и обучением методом обратного распространения ошибки, но на их создание требуется значительно меньше времени. Эти сети наиболее эффективны, когда доступно большое количество обучающих векторов.

Сеть *RBF* является аналогом многослойного персептрона. Скорость обучения такой сети гораздо выше. Однако существует и ряд недостатков, главным из которых является ухудшение точности аппроксимации. Сеть обладает хорошей обобщающей способностью только для ограниченного класса аппроксимируемых функций. В качестве классификатора такая сеть может с успехом применяться в случае хорошей кластеризации классов в пространстве признаков.

При аппроксимации радиально-базисной сетью:

– число j базисных функций выбирается много меньше числа обучающих данных: $j << N$ (в качестве базисной можно использовать любую радиально-базисную функцию. На практике часто используют гауссиан);

– центры базисных функций μ_j не опираются на точки входных данных, т. е. не совпадают ни с одним из входных векторов. Определение центров функций становится частью процесса обучения (на практике в качестве центров могут быть выбраны значения из обучающей выборки, которые в дальнейшем перестают использоваться при обучении);

– для каждой из j базисных функций задается своя ширина окна σ_j, которая также определяется в процессе обучения *RBF*-сети. Как правило, значение σ_j делают чуть большим расстояния между центрами соответствующих базисных функций μ_j.

1.4.2. Обучение радиально-базисной сети

Имеется обучающий набор: множество входов $\{xn\}$ и соответствующих выходов $\{d^n\}$. На первом этапе определяются параметры базисных функций μ_i, σ_i. Причем используются только входные векторы $\{x^n\}$, т. е. обучение происходит по схеме «без учителя»:

1. При наличии небольшого количества эталонных образцов для обучения в качестве центров радиально-симметричных функций следует выбирать соответствующие им векторы. Если объем обучающей выборки достаточно велик, в качестве центров могут быть использованы:

– центры потенциальных кластеров, по которым можно распределить все примеры обучающей выборки вручную или с использованием дополнительных алгоритмов кластеризации, в том числе других архитектур нейронных сетей;

– отдельные случайные примеры обучающей выборки.

2. Выбор радиусов радиальных элементов определяется требуемым видом радиально-симметричной функции. При больших значениях радиусов график функции слишком острый, а это значит, что сеть не будет корректно интерполировать данные между известными точками на достаточно большом удалении от них, так как теряет способность к обобщению обучающих данных. Наоборот, при чрезмерно малых значениях сеть становится невосприимчивой к отдельным деталям. С учетом вышеуказанного радиусы могут задаваться следующими способами:

– пользователем нейронной сети в явном виде на основе эвристического подбора;

– рассчитываться автоматически по среднему расстоянию до нескольких (в зависимости от общего объема обучающей выборки и количества скрытых нейронов) ближайших примеров.

На практике для определения σ_i часто используется алгоритм «ближайшего соседа», который заключается в поиске разбиения множества $\{x^n\}$ на j несмежных подмножеств S_i. Таким образом, необходимо минимизировать функцию:

$$J = \sum_{i=1}^{J} \sum_{n \in S_i} \left(x^n - \mu_i \right)^2; \quad \mu_i = \frac{1}{N_i} \sum_{n \in S_i} x^n. \tag{1.82}$$

3. Фиксируются базисные функции, т. е. параметры μ_i, σ_i постоянны. На данном этапе *RBF*-сеть эквивалентна однослойной нейронной сети. Затем обучение происходит по правилу обучения с учителем.

4. Строится функция потери:

$$L = \frac{1}{2} \sum_{k=1}^{n} \left(\sum_{i=0}^{j} w_i \varphi_i(x) - d_k \right)^2 \to \min. \tag{1.83}$$

5. Так как L является квадратической функцией от весов w, то минимум L может быть найден решением системы линейных уравнений:

$$\frac{\partial L}{\partial w_i} = \sum_{k=1}^{n} \left(w_i \varphi_i(x) - d_k \right) \varphi_i(x) = 0. \tag{1.84}$$

Этап ***оптимизации весовых коэффициентов*** [10] линейного выходного слоя может быть записан в матричной форме:

1. Рассчитывается характеристическая матрица значений радиально-симметричных элементов всех обучающих примеров:

$$\Phi = \begin{bmatrix} \varphi_1(x_1) & \cdots & \varphi_j(x_1) \\ \vdots & \cdots & \vdots \\ \varphi_1(x_n) & \cdots & \varphi_j(x_n) \end{bmatrix}. \tag{1.85}$$

2. Методами линейной алгебры рассчитывается матрица весовых коэффициентов выходного слоя нейронов:

$$W = \left(\Phi^T \Phi \right)^{-1} \Phi^T Y, \tag{1.86}$$

где матрица выходов обучающих примеров содержит столбцы в количестве, равном числу обучающих примеров, и строки – в количестве, соответствующем числу выходов нейронной сети:

$$Y = \begin{bmatrix} y_{11} & \cdots & y_{K1} \\ \vdots & \cdots & \vdots \\ y_{1n} & \cdots & y_{Kn} \end{bmatrix}. \tag{1.87}$$

Пример аппроксимации приведен в табл. 1.4.

Таблица 1.4

Пример аппроксимации

Параметры	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
X	–2,0	–1,5	–1,0	–0,5	0	0,5	1	1,5	2
Y	–0,48	–0,78	–0,83	–0,67	–0,20	0,7	1,48	1,17	0,2

График исходной функции дан на рис. 1.17.

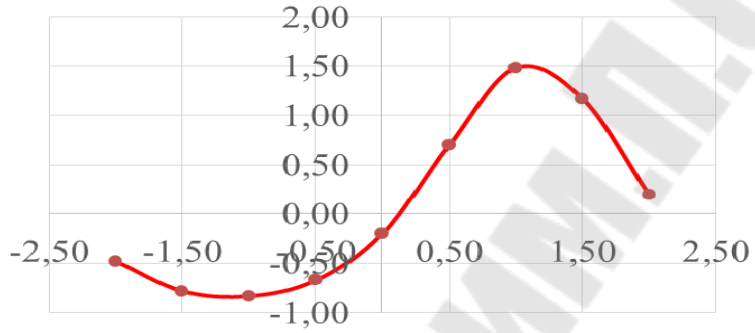


Рис. 1.17. График исходной функции

Зададимся структурой сети, включающей 5 скрытых нейронов (радиальных элементов).

В соответствии с имеющимся алгоритмом требуется указать центры и радиусы скрытых радиальных элементов.

Используем в качестве центров радиальных элементов значения независимой переменной в опытах 1, 3, 5, 7 и 9, а для каждого элемента примем $\sigma = 1{,}5$.

Характеристическая матрица значений радиально-симметричных элементов всех обучающих примеров приведена в табл. 1.5.

Таблица 1.5

Характеристическая матрица значений радиально-симметричных элементов всех обучающих примеров

Ф =	1	0,8	0,64	0,51	0,41
	0,89	0,89	0,72	0,57	0,46
	0,8	1	0,8	0,64	0,51
	0,72	0,89	0,89	0,72	0,57
	0,64	0,8	1	0,8	0,64

Окончание табл. 1.5

Ф =	0,57	0,72	0,89	0,89	0,72
	0,51	0,64	0,8	1	0,8
	0,46	0,57	0,72	0,89	0,89
	0,41	0,51	0,64	0,8	1

Выполним матричные вычисления и представим в табл. 1.6.

Таблица 1.6

W **=**	0,59	–2,48	–2,63	7,17	–2,75

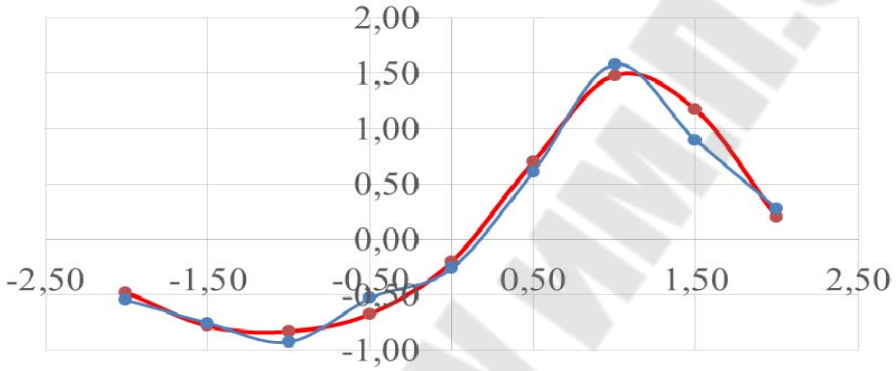


Рис. 1.18. Результат аппроксимации исходной функции радиально-базисной сетью

Результат аппроксимации исходной функции радиально-базисной сетью показан на рис. 1.18.

1.5. Анализ временных рядов

1.5.1. Понятие временного ряда

Пусть имеется наблюдаемый процесс P, нестационарный во времени. Результатом наблюдения являются измерения характеристики x процесса в моменты времени $t_1, t_2, \ldots, t_N$.

Полагая, что для всех значений времени выполняется $t_{i+1} - t_i = \text{const}, \quad i = \overline{1, N-1}$, множество измеренных значений $\{x(t_1), x(t_2), \ldots, x(t_N)\}$ можно обозначить:

$$X = \{x_1, x_2, \ldots, x_N\}. \tag{1.88}$$

Полученное множество величин X называется **временным рядом**.

В зависимости от того, является ли значение x_i скаляром или вектором, говорят об ***одномерном*** или ***многомерном*** временном ряде.

Задача ***экстраполяции*** (прогноза) временного ряда – одна из основных в *анализе временных рядов*.

Ее суть можно изложить следующим образом:

• Пусть известны значения временного ряда до некоторого момента времени T: $X = \{x_1, x_2, \ldots, x_T\}$.

• Необходимо, используя данные значения, получить оценки неизвестных значений ряда в будущем: $\widetilde{X} = \{\widetilde{x}_{T+1}, \widetilde{x}_{T+2}, \ldots, \widetilde{x}_{T+K}\}$.

• *Причем каждое последующее значение вычисляется с использованием имеющихся оценок*.

Ряд, полученный «конкатенацией» X и $\widetilde{X}$, называют ***реконструкцией*** временного ряда.

Для решения задачи прогнозирования разработано большое число методов.

1.5.2. Методы предсказания временных рядов

Экспертные методы прогнозирования. Самый распространенный метод из группы экспертных методов – метод Дельфи. Суть метода заключается в сборе мнений различных экспертов и их обобщение в единую оценку.

Методы логического моделирования. Основаны на поиске и выявлении закономерностей рынка в долгосрочной перспективе.

Сюда входят следующие методы:

– метод сценариев («если – то»), описание последовательностей исходов из того или иного события с созданием базы знаний;

– методы прогнозов по образу;

– метод аналогий.

Экономико-математические методы. Методы из этой группы базируются на создании моделей исследуемого объекта. Экономико-математическая модель [11] – это определенная схема, путь развития рынка ценных бумаг при заданных условиях. При прогнозировании финансовых временных рядов используют статистические, динамические, микро-, макро-, линейные, нелинейные, глобальные, локальные, отраслевые, оптимизационные, дескриптивные. Оптимизационные модели представляют собой систему уравнений, куда входят различные ограничения, а также особое уравнение, называемое функционалом оптимальности (или критерием оптимальности). С помощью него находят оптимальное, наилучшее решение по какому-либо показателю.

Статистические методы. Статистические методы прогнозирования применительно для финансовых временных рядов основаны на построении различных индексов (диффузный, смешанный), расчете значений дисперсии, математического ожидания, вариации, ковариации, интерполяции, экстраполяции.

Технический анализ. Прогнозирование изменений цен в будущем на основе анализа изменений цен в прошлом. В его основе лежит анализ временны́х рядов цен – «чартов» (от англ. *chart*). Помимо ценовых рядов в техническом анализе используется информация об объемах торгов и другие статистические данные. Наиболее часто методы технического анализа используются для анализа цен, изменяющихся свободно, например, на биржах. В техническом анализе множество инструментов и методов, но́ все они основаны на одном предположении: из анализа временны́х рядов, выделяя тренды, можно спрогнозировать поведение цен.

Фундаментальный анализ. Метод прогнозирования рыночной (биржевой) стоимости компании, основанный на анализе финансовых и производственных показателей ее деятельности. Фундаментальный анализ используется инвесторами для оценки стоимости компании (или ее акций), которая отражает состояние дел в компании, рентабельность ее деятельности. При этом анализу подвергаются финансовые показатели компании: выручка, *EBITDA* (*Earnings Before Interests Tax, Deprecation and Amortization*), чистая прибыль, чистая стоимость компании, обязательства, денежный поток, величина выплачиваемых дивидендов и производственные показатели компании.

1.6. Решение задач с помощью искусственных нейронных сетей

Существуют следующие этапы решения задач с помощью ИНС:

1. Сбор данных для обучения.
2. Подготовка и нормализация данных.
3. Выбор топологии сети.
4. Экспериментальный подбор характеристик сети.
5. Экспериментальный подбор параметров обучения.
6. Обучение сети.
7. Проверка адекватности обучения.
8. Корректировка параметров, окончательное обучение.
9. Вербализация сети с целью дальнейшего использования.

1.6.1. Сбор данных для обучения

Сбор данных для обучения осуществляется исходя из следующих критериев:

– репрезентативность – данные должны иллюстрировать истинное положение вещей в предметной области;

– непротиворечивость – противоречивые данные в обучающей выборке приведут к плохому качеству обучения сети.

Исходные данные преобразуются к виду, в котором их можно подать на входы сети. Каждая запись в файле данных называется ***обучающей парой*** или ***обучающим вектором***. Обучающий вектор содержит по одному значению на каждый вход сети и в зависимости от типа обучения (с учителем или без) – по одному значению для каждого выхода сети.

Подготовка и нормализация данных. Основная цель – это улучшение «восприятия» сети:

– ***нормировка*** выполняется, когда на различные входы подаются данные разной размерности. При нормировке размерности всех входных и выходных данных сводятся воедино;

– ***квантование*** выполняется над непрерывными величинами, для которых выделяется конечный набор дискретных значений. Например, квантование используют для задания частот звуковых сигналов при распознавании речи;

– ***фильтрация*** выполняется для «зашумленных» данных.

Выбор топологии сети следует исходя из постановки задачи и имеющихся данных для обучения.

Для обучения с учителем требуется наличие для каждого элемента выборки «экспертной» оценки.

Экспериментальный подбор характеристик сети. Для сетей, подобных перцептрону, это будет число слоев, число блоков в скрытых слоях (для сетей Ворда), наличие или отсутствие обходных соединений, передаточные функции нейронов.

При выборе количества слоев и нейронов в них следует исходить из того, что способности сети к обобщению тем выше, чем больше суммарное число связей между нейронами. Число связей ограничено сверху количеством записей в обучающих данных.

Экспериментальный подбор параметров обучения. Важен для сетей, обучающихся с учителем.

От правильного выбора параметров зависит не только то, насколько быстро ответы сети будут сходиться к правильным ответам.

Выбор скорости обучения, момент обучения нужно выбирать экспериментально, руководствуясь при этом критерием завершения обучения.

Обучение сети. В определенном порядке просматривает обучающую выборку. Порядок просмотра может быть последовательным, случайным и т. д.

При обучении с учителем набор исходных данных делят на две части – собственно обучающую выборку и тестовые данные; принцип разделения может быть произвольным.

Обучающие данные подаются сети для обучения, а проверочные используются для расчета ошибки сети (проверочные данные никогда для обучения сети не применяются). Таким образом, если на проверочных данных ошибка уменьшается, то сеть действительно выполняет обобщение. Если ошибка на обучающих данных продолжает уменьшаться, а ошибка на тестовых данных увеличивается, значит, сеть перестала выполнять обобщение и просто «запоминает» обучающие данные. Это явление называется переобучением сети, или ***оверфиттингом***. В таких случаях обучение обычно прекращают.

Проверка адекватности обучения. Даже в случае успешного, на первый взгляд, обучения сеть не всегда обучается именно тому, чего от нее хотел создатель. Сеть «понимает» не то, что от нее требовалось, а то, что проще всего обобщить.

Тестирование качества обучения нейросети необходимо проводить на примерах, которые не участвовали в ее обучении. При этом число тестовых примеров должно быть тем больше, чем выше качество обучения. Если ошибки нейронной сети имеют вероятность, близкую к одной миллиардной, то и для подтверждения этой вероятности нужен миллиард тестовых примеров. Получается, что тестирование хорошо обученных нейронных сетей становится очень трудной задачей.

Кодирование выходных значений. Задача классификации при наличии двух классов может быть решена на сети с одним нейроном в выходном слое, который может принимать одно из двух значений 0 или 1, в зависимости от того, к какому классу принадлежит образец, т. е. ***бинарное кодирование***.

При наличии нескольких классов возникает проблема, связанная с представлением этих данных для выхода сети.

Наиболее простым способом представления выходных данных в таком случае является вектор, компоненты которого соответствуют различным номерам классов. При этом *i*-я компонента вектора соответствует *i*-му классу. Все остальные компоненты при этом устанавли-

ваются в 0. Тогда, например, второму классу будет соответствовать 1 на *2* выходе сети и 0 – на остальных. При интерпретации результата обычно считается, что номер класса определяется номером выхода сети, на котором появилось максимальное значение.

При таком способе кодирования иногда вводится также понятие уверенности сети в том, что пример относится к этому классу.

Наиболее простой способ определения уверенности заключается в определении разности между максимальным значением выхода и значением другого выхода, которое является ближайшим к максимальному.

Соответственно, чем выше уверенность, тем больше вероятность того, что сеть дала правильный ответ. Этот метод кодирования является самым простым, но не всегда самым оптимальным способом представления данных.

На практике также часто используется так называемый способ кодирования, в котором выходной вектор представляет собой номер кластера, записанный в двоичной форме. Тогда при наличии 8 классов нам потребуется вектор из 3 элементов. Но при этом в случае получения неверного значения на одном из выходов мы можем получить неверную классификацию (неверный номер кластера), поэтому имеет смысл увеличить расстояние между двумя кластерами за счет использования кодирования выхода по коду Хемминга, который повысит надежность классификации.

Третий подход состоит в разбиении задачи с k классами на $k*(k-1)/2$ подзадач с двумя классами (2 на 2 кодирование) каждая. Под подзадачей в данном случае понимается то, что сеть определяет наличие одной из компонент вектора. То есть исходный вектор разбивается на группы по два компонента в каждой таким образом, чтобы в них вошли все возможные комбинации компонент выходного вектора. Число этих групп можно определить как количество неупорядоченных выборок по два из исходных компонент. Например, для задачи с четырьмя классами мы имеем 6 выходов (подзадач), 1 на выходе говорит о наличии одной из компонент. Тогда можно перейти к номеру класса по результату расчета сетью следующим образом:

– определяются, какие комбинации получили единичное (точнее близкое к единице) значение выхода (т.е. какие подзадачи активировались);

– номер класса будет тот, который вошел в наибольшее количество активированных подзадач.

Выбор объема сети. Правильный выбор объема сети имеет большое значение. Построить небольшую и качественную модель часто бывает просто невозможно, а большая модель будет только запоминать примеры из обучающей выборки и не производить аппроксимацию, что, естественно, приведет к некорректной работе классификатора.

Существуют два основных подхода к построению сети:

– ***конструктивный*** – вначале берется сеть минимального размера, и постепенно увеличивается до достижения требуемой точности. При этом на каждом шаге ее заново обучают. Также существует так называемый метод ***каскадной корреляции***, при котором после окончания эпохи происходит корректировка архитектуры сети с целью минимизации ошибки;

– ***деструктивный*** – вначале берется сеть завышенного объема, и затем из нее удаляются узлы и связи, мало влияющие на решение. При этом полезно помнить следующее правило: *число примеров в обучающем множестве должно быть больше числа настраиваемых весов*. Иначе вместо обобщения сеть просто запомнит данные и утратит способность к классификации – результат будет неопределен для примеров, которые не вошли в обучающую выборку.

При выборе архитектуры сети обычно опробуется несколько конфигураций с различным количеством элементов. При этом основным показателем является объем обучающего множества и обобщающая способность сети. Обычно используется алгоритм обучения ***Back Propagation*** (обратного распространения) с подтверждающим множеством.

1.6.2. Подготовка данных

Для распознавания объекты предъявляются в виде совокупности (выборки) наблюдений, обычно записываемой в виде матрицы:

$$X = \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \vdots & \vdots \\ x_{p1} & \dots & x_{pn} \end{bmatrix}, \qquad (1.89)$$

где $x_i = \{x_{i1}, \dots, x_{in}\}$ – вектор наблюдаемых значений признаков.

Совокупность признаков должна в наибольшей степени отражать те свойства объектов, которые важны для классификации. При этом от размерности p признакового пространства зависит вычислительная

сложность процедур обучения и принятия решения, достоверность классификации, затраты на измерение характеристик объектов.

Первоначальный набор признаков формируется из числа доступных измерению характеристик объекта, отражающих наиболее существенные для классификации свойства (обычно все определяемые свойства) [10].

На следующих этапах необходимо:

– уменьшить размерность признакового пространства;

– исключить зависимые (коррелирующие признаки);

– исключить несущественные признаки, влиянием которых на значение решающей функции можно пренебречь.

При подготовке данных для обучения нейронной сети нужно обращать внимание на следующие существенные моменты:

– количество наблюдений в наборе данных. Следует учитывать тот фактор, что чем больше размерность данных, тем больше времени потребуется для обучения сети. Требуется определить наличие выбросов и оценить необходимость их присутствия в выборке;

– обучающая выборка должна быть представительной (репрезентативной). Обучающая выборка не должна содержать противоречий, так как нейронная сеть однозначно сопоставляет выходные значения входным;

– нейронная сеть работает только с числовыми входными данными, поэтому важным этапом при подготовке данных является преобразование и кодирование данных. При использовании на вход нейронной сети следует подавать значения из того диапазона, на котором она обучалась;

– нормализация данных.

Нормализация данных – это преобразование данных к виду, который наиболее подходит для обработки, т. е. данные, поступающие на вход, должны иметь числовой тип, а их значения должны быть распределены в определенном диапазоне.

Нормализатор может приводить дискретные данные к набору уникальных индексов, либо преобразовывать значения, лежащие в произвольном диапазоне, в конкретный диапазон, например, [0…1]. Нормализация выполняется путем деления каждой компоненты входного вектора на длину вектора, что превращает входной вектор в единичный.

Емкость ИНС – число классов, предъявляемых на входы ИНС для распознавания. Для разделения множества входных классов, например, по двум классам достаточно всего одного выхода.

Для повышения достоверности классификации желательно ввести избыточность путем выделения каждому классу одного нейрона в выходном слое или, что еще лучше, нескольких, каждый из которых обучается определять принадлежность образа к классу со своей степенью достоверности.

Максимизация энтропии является целью предобработки.

Задача нейросетевого моделирования – найти статистически достоверные зависимости между входными и выходными переменными. Единственным источником информации для статистического моделирования являются примеры из обучающей выборки. Чем больше битов информации принесет пример – тем лучше используются имеющиеся в распоряжении данные.

Среднее количество информации, приносимой каждым примером, равно энтропии распределения значений этой компоненты.

Если эти значения сосредоточены в относительно небольшой области единичного интервала, информационное содержание такой компоненты мало. В пределе нулевой энтропии, когда все значения переменной совпадают, эта переменная не несет никакой информации. Напротив, если значения переменной равномерно распределены в единичном интервале, информация такой переменной максимальна.

1.6.3. Проблема «проклятия размерности»

Основные проблемы размерности – сложно определить метрику в пространствах больших размерностей, что приводит к проблемам при сравнении объектов между собой и определении их степени близости:

$$a = \{x_1, x_2, \ldots, x_N\}; \tag{1.90}$$

$$b = \{x_1 + \varepsilon, x_2 + \varepsilon, \ldots, x_N + \varepsilon\}; \tag{1.91}$$

$$c = \{x_1, x_2 + \delta, \ldots, x_N\}; \quad \delta > \varepsilon. \tag{1.92}$$

Какие объекты более схожи – a и b или a и c?

Экспонтенциальный рост объема необходимых данных:

– если $X = \{1, 0, \ldots, 1\}$ – бинарный вектор входных признаков, тогда всех возможных наборов признаков будет 2^N, т. е. требуются значительное количество элементов в обучающей выборке и большое число нейронов внутри самой сети;

– когда количество признаков сравнимо с количеством объектов, объекты могут оказаться равноудаленными друг от друга. Куб с длиной ребра $0{,}999$ будет составлять $0{,}999^N$ части объема единичного куба, тогда

$$\lim_{N\to\infty} 0{,}999^N = 0. \qquad (1.93)$$

В пространствах больших размерностей основная часть объема концентрируется вблизи границы области, т. е. объекты оказываются равноудалены друг от друга.

«*Проклятие размерности*» – это экспотенциальный рост необходимых экспериментальных данных и усложнение многомерной целевой функции в зависимости от размерности пространства, что приводит к увеличению погрешности (или невозможности) нахождения зависимости между входным вектором и результатом работы нейронной сети. При этом возрастает вычислительная ресурсоемкость используемых алгоритмов и накапливаемая погрешность.

1.6.4. Понижение размерности задачи

Неинформативные признаки являются источником дополнительного шума и влияют на точность оценки параметров модели.

Кроме того, наборы данных с большим числом признаков могут содержать группы коррелированных переменных. Наличие таких групп признаков означает дублирование информации, которое может искажать спецификацию модели и влиять на качество оценки ее параметров.

Чем выше размерность данных, тем выше объем вычислений при их алгоритмической обработке.

Можно выделить два направления в снижении размерности признакового пространства по принципу используемых для этого переменных:

– отбор признаков из имеющегося исходного набора;

– формирование новых признаков путем трансформации первоначальных данных.

В идеальном случае сокращенное представление данных должно иметь размерность, соответствующую размерности, внутренне присущей данным (*intrinsic dimensionality*).

Прямое решение задачи создания наилучшей модели связано с перебором всех возможных сочетаний признаков, что обычно представляется чрезмерно трудоемким.

Поэтому, как правило, прибегают к ***прямой*** или ***обратной*** *селекции признаков*.

В процедурах ***прямого отбора*** производится последовательное добавление переменных из исходного набора до достижения необходимого качества модели.

В алгоритмах ***последовательной редукции*** исходного признакового пространства (обратной селекции) производится поэтапное удаление наименее информативных переменных до допустимого снижения информативности модели.

Информативность признаков относительна. Отбор должен обеспечить высокую информативность набора признаков, а не суммарную информативность составляющих его переменных. Так, наличие корреляции между признаками снижает их общую информативность вследствие дублирования общей для них информации.

Отбор взаимно ортогональных признаков. Переменные ранжируются по информативности, и используется такой состав первых в этом рейтинге признаков, который обеспечивает заданную информативность.

Ограниченность методов отбора признаков с целью снижения размерности пространства связана с предположением о непосредственном присутствии необходимых признаков в исходных данных, что обычно оказывается неверным.

Формирование нового признакового пространства предполагает создание новых переменных, которые обычно являются функциями исходных признаков. Эти переменные, непосредственно не наблюдаемые, часто называют скрытыми, или латентными.

В процессе создания эти переменные могут быть наделены различными полезными свойствами, такими как ортогональность. На практике исходные признаки обычно взаимосвязаны, поэтому трансформация их пространства в ортогональное порождает новые координаты-признаки, в которых отсутствует эффект дублирования информации об исследуемых объектах.

Традиционно уменьшение размерности выполняется с помощью следующих групп методов:

– компонентный анализ (*PCA*, *ICA*);

– факторный анализ (метод цепных подстановок);

– эвристические методы снижения размерности (метод экстремальной группировки признаков, метод корреляционных плеяд);

– многомерное шкалирование (*multidimensional scaling*, *MDS*);

– автоэнкодеры.

Извлечение признаков. *Признак* (*feature*) – это результат измерения некоторой характеристики объекта, либо всего объекта.

Признак – это отображение объекта в пространство определения признака.

Есть следующие типы признаков:

– бинарные;

– номинальные (конечные);

– порядковые;

– количественные и др.

Если все признаки имеют одинаковый тип, то исходные данные называются ***однородными***, в противном случае – ***разнородными***.

Признак – это числовой вектор (или в общем любая сущность, которую можно представить в цифровом виде), описывающий объект и позволяющий его выделить из множества подобных.

Совокупность признаков объекта определяет его *признаковое описание*.

Признаковое описание объекта – это вектор, составленный из значений фиксированного набора признаков на данном объекте. Признаки в общем случае могут иметь различные типы, причем не обязательно числовые.

Совокупность признаковых описаний всех объектов обучающей выборки в виде матрицы называют матрицей *объектов-признаков*, *матрицей информации* или просто *матрицей исходных данных*.

Строки матрицы – это признаковые описания обучающих объектов. Столбцы матрицы соотвествуют признакам.

Приведение разнородных описаний объектов к стандартному матричному виду называется *извлечением признаков из данных* (*features extraction*) или *генерацией признаков* (*features generation*).

Таким образом, признаки – это характеристики объектов, которые либо измеряются непосредственно, либо вычисляются по «сырым» исходным данным.

Для разного типа сложных и разнородных объектов уже имеется ряд методов извлечения признаков:

– для графических – *HoG*, *SIFT* и др.

– для звуковых – *MFCC*, *Spectrogramme* и т. д.

– для текстовых – *Bag of Words* (или *term frequency*) и др.

Deep Learning позволяет «учить» признаки наравне с моделью и часто получается лучше разработанных вручную признаков.

ГЛАВА 2. НЕЙРОННЫЕ СЕТИ С ОБРАТНЫМИ И ЛАТЕРАЛЬНЫМИ СВЯЗЯМИ

2.1. Сети с обратными связями

2.1.1. Сеть Хопфилда

Нейронная сеть Хопфилда – полносвязная нейронная сеть с симметричной матрицей связей. В процессе работы динамика таких сетей сходится (конвергирует) к одному из положений равновесия. Эти положения равновесия являются локальными минимумами функционала, называемого энергией сети (в простейшем случае – локальными минимумами отрицательно определенной квадратичной формы на n-мерном кубе). Сеть может быть использована как автоассоциативная память, как фильтр, а также для решения некоторых задач оптимизации. В отличие от многих нейронных сетей, функционирующих до получения ответа через определенное количество тактов, сети Хопфилда функционируют до достижения равновесия, когда следующее состояние сети в точности равно предыдущему: начальное состояние является входным образом, а при равновесии получают выходной образ [6].

Архитектура сети. Архитектура нейронной сети Хопфилда представлена на рис. 2.1.

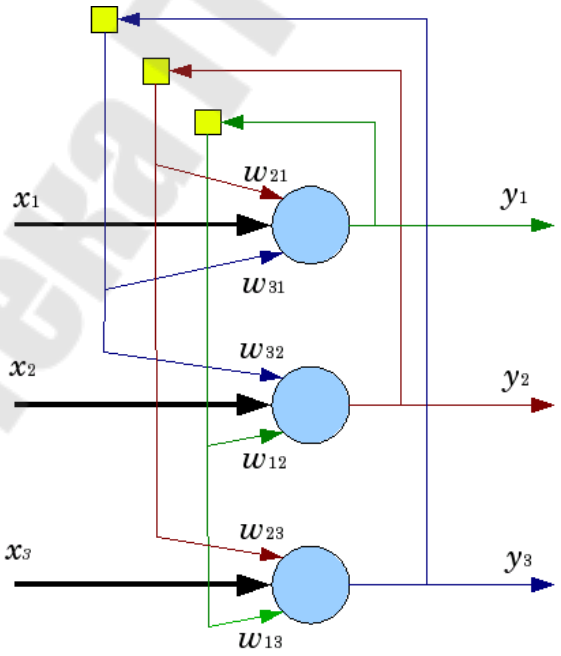


Рис. 2.1. Архитектура нейронной сети Хопфилда

Нейронная сеть Хопфилда состоит из N искусственных нейронов [4]. Каждый нейрон системы может принимать одно из двух состояний (что аналогично выходу нейрона с пороговой функцией активации):

$$x_i = \begin{cases} 1, \\ -1. \end{cases} \tag{2.1}$$

Благодаря своей биполярной природе, нейроны сети Хопфилда иногда называют спинами.

Взаимодействие спинов сети описывается выражением

$$E = \frac{1}{2}\sum_{i,j=1}^{N} w_{ij} x_i x_j, \tag{2.2}$$

где w_{ij} – элемент матрицы взаимодействий W, которая состоит из весовых коэффициентов связей между нейронами. В эту матрицу в процессе обучения записывается M «образов» – N-мерных бинарных векторов: $S_m = (s_{m1}, s_{m2}, \ldots, s_{mN})$.

В сети Хопфилда матрица связей симметрична $w_{ij} = w_{ji}$, а диагональные элементы матрицы полагаются равными нулю $(w_{ij} = 0)$, что исключает эффект воздействия нейрона на самого себя и является необходимым для сети Хопфилда, но недостаточным условием устойчивости в процессе работы сети. Достаточным является асинхронный режим работы сети.

Обучение сети. Обучение сети заключается в том, что находятся веса матрицы взаимодействий так, чтобы запомнить m векторов (эталонных образов, составляющих «память» системы).

Вычисление коэффициентов основано на следующем правиле: *для всех «запомненных» образов X_i матрица связи должна удовлетворять уравнению*

$$X_i = WX_i, \tag{2.3}$$

поскольку именно при этом условии состояния сети X_i будут устойчивы – попав в такое состояние, сеть в нем и останется.

Запоминаемые векторы должны иметь бинарный вид. Расчет весовых коэффициентов проводится по следующей формуле:

$$w_{ij} = \frac{1}{N} \sum_{d=1...m} X_{id} X_{jd}, \qquad (2.4)$$

где N – размерность векторов; m – число запоминаемых выходных векторов; d – номер запоминаемого выходного вектора; X_{ij} – i-я компонента запоминаемого выходного j-го вектора.

Это выражение может стать более ясным, если заметить, что весовая матрица W может быть найдена вычислением внешнего произведения каждого запоминаемого вектора с самим собой и суммированием матриц, полученных таким образом. Это может быть записано в виде

$$W = \frac{1}{N} \sum_{i} X_i X_i^T, \qquad (2.5)$$

где X_i – i-й запоминаемый вектор-столбец.

Расчет этих весовых коэффициентов и называется обучением сети, которое проводится только за одну эпоху.

Применение обученной сети. Как только веса заданы, обученная сеть становится способной «распознавать» входные сигналы, т. е. определять, к какому из запомненных образцов они относятся.

Входной вектор проходит некоторое количество итераций до достижения сходимости (конвергенции). При этом должны распознаваться частично искаженные или неполные образцы. На вход сети сначала передают значения исходного вектора (поэтому обозначение на схеме сети входных синапсов в явном виде носит чисто условный характер). Затем сеть последовательно меняет свои состояния согласно формуле

$$X_{i+1} = F(WX_i), \qquad (2.6)$$

где F – активационная функция; X_i и X_{i+1} – текущее и следующее состояния сети до тех пор, пока эти состояния не совпадут (или в случае синхронного режима работы не совпадут состояния X_{i-1} с X_{i+1} и одновременно X_{i-2} с X_i). Именно этот процесс называется конвергенцией сети. Полученное устойчивое состояние X_i (статический аттрактор), или, возможно, в синхронном случае – пара $\{X_i, X_{i+1}\}$ (динамический аттрактор) является ответом сети на данный входной образ.

На выходе сети может получаться также инверсный вектор (в котором значения –1 и 1 в запомненных образцах перевернуты).

В случае, если система не нашла решения, на выходе системы могут получаться также тривиальные векторы, состоящие только из 1 или только из –1.

Работа сети в режиме фильтрации (восстановление поврежденных образов). Так как сети с обратными связями имеют пути, передающие сигналы от выходов к входам, то отклик таких сетей является динамическим, т. е. после приложения нового входа вычисляется выход и, передаваясь по сети обратной связи, модифицирует вход. Затем выход повторно вычисляется, и процесс повторяется снова и снова. Для устойчивой сети последовательные итерации приводят к все меньшим изменениям выхода, пока в конце концов выход не становится постоянным. Для некоторых сетей процесс никогда не заканчивается, такие сети называют неустойчивыми.

Рассмотрим основной цикл работы сети. Как только веса заданы, сеть может быть использована для получения запомненного выходного вектора по данному входному вектору, который может быть частично неправильным или неполным. Для этого выходам сети сначала придают значения этого начального вектора. Затем сеть последовательно меняет свои состояния согласно формуле

$$X(t+1)^T = F(WX(t)), \tag{2.7}$$

где F – активационная функция; $X(t)$ и $X(t+1)$ – текущее и следующее состояния сети до тех пор, пока эти состояния не совпадут (или, в случае синхронного режима работы, не совпадут состояния $X(t-1)$ с $X(t+1)$ и одновременно $X(t-2)$ с $X(t)$). Именно этот процесс называется конвергенцией сети.

Это же можно описать локальным полем a_i, действующим на нейрон x_i со стороны всех остальных нейронов сети:

$$a_i(t) = \sum_{j=1, j\neq i}^{N} w_{ij} x_j(t-1). \tag{2.8}$$

После расчета локального поля нейрона $a_i(t)$ это значение используется для расчета значения выхода через функцию активации, которая в данном случае является пороговой (с нулевым порогом). Соответственно, значение выхода нейрона i в текущий момент времени $x_i(t)$ рассчитывается по формуле

$$x_i(t) = \text{sign}\left(\sum_{j=1, j\neq i}^{N} w_{ij} x_j(t-1) \right), \qquad (2.9)$$

где w_{ij} – весовой коэффициента между нейронами i и j; $x_j(t-1)$ – значения выходов нейрона j в предыдущий момент времени.

Во время работы сети Хопфилда признаком нахождения решения является момент, когда достигается аттрактор статический (если на каждом следующем шаге повторяется устойчивое состояние $X(t)$), или, возможно, динамический (когда до бесконечности чередуются два разных состояния $\{X(t), X(t+1)\}$. Это конечное состояние сети и является ее реакцией на данный образ.

Обычно ответом является такое устойчивое состояние, которое совпадает с одним из запомненных при обучении векторов, однако при некоторых условиях (в частности, при слишком большом количестве запомненных образов) результатом работы может стать так называемый ложный аттрактор («химера»), состоящий из нескольких частей разных запомненных образов, а также в синхронном режиме сеть может прийти к динамическому аттрактору. Обе эти ситуации в общем случае являются нежелательными, поскольку не соответствуют ни одному запомненному вектору, а соответственно, не определяют класс, к которому сеть отнесла входной образ.

Для сети Хопфилда могут существовать две модификации, отличающиеся по времени передачи сигнала: асинхронный и синхронный режимы.

Если во время обучения сформировать матрицу весовых коэффициентов (межнейронных связей) на основании эталонных бинарных векторов, то нейронная сеть в процессе работы под действием описанных выше полей будет менять состояния нейронов до тех пор, пока не перейдет к одному из устойчивых состояний.

Ограничения сети. К сожалению, у нейронной сети Хопфилда есть ряд недостатков:

1. Относительно небольшой объем памяти, величину которого можно оценить выражением

$$M \approx \frac{N}{2\log_2 N}. \qquad (2.10)$$

Попытка записи большего числа образов приводит к тому, что нейронная сеть перестает их распознавать.

2. Достижение устойчивого состояния не гарантирует правильный ответ сети. Это происходит из-за того, что сеть может сойтись к так называемым ложным аттракторам – «химерой» (как правило, химеры склеены из фрагментов различных образов).

2.1.2. Сеть Хэмминга

Нейронная сеть Хэмминга – вид нейронной сети, использующийся для классификации бинарных векторов, основным критерием в которой является расстояние Хэмминга. Данная сеть – развитие нейронной сети Хопфилда [8].

Сеть используется для того, чтобы соотнести бинарный вектор $x = (x_1, x_2, x_3, ..., x_m)$, где $x_i = \{-1,1\}$, с одним из эталонных образов (каждому классу соответствует свой образ), или же решить, что вектор не соответствует ни одному из эталонов. В отличие от сети Хопфилда она выдает не сам образец, а его номер.

Сеть предложена Ричардом Липпманном в 1987 г. Она позиционировалась как специализированное гетероассоциативное запоминающее устройство.

Сеть Хэмминга – трехслойная нейронная сеть с обратной связью. Количество нейронов во втором и третьем слоях равно количеству классов классификации. Синапсы нейронов второго слоя соединены с каждым входом сети, нейроны третьего слоя связаны между собой отрицательными связями, кроме синапса, связанного с собственным аксоном каждого нейрона, – он имеет положительную обратную связь.

Архитектура нейронной сети Хэмминга представлена на рис. 2.3.

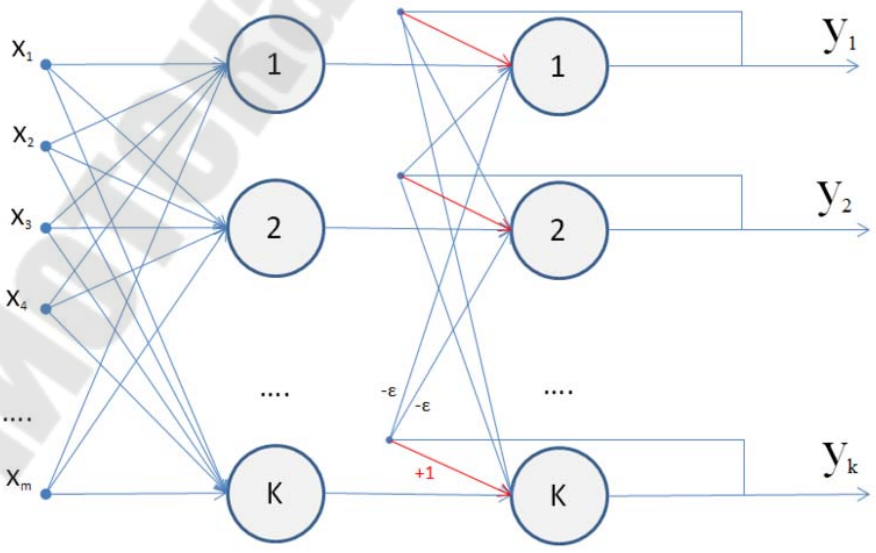


Рис. 2.3. Архитектура нейронной сети Хэмминга

Обучение сети. Матрица весовых коэффициентов первого слоя получается из матрицы эталонных образов X как $w_{ij} = \frac{x_{ij}}{2}$, где матрица эталонных образов – это матрица $K \times M$, каждая строка которой – соответствующий эталонный бинарный вектор.

Функция активации определяется как

$$f(s) = \begin{cases} 0, & s \le 0; \\ s, & 0 < s \le T; \\ T, & s > T, \end{cases} \tag{2.11}$$

где

$$T = \frac{M}{2}. \tag{2.12}$$

Матрица весовых коэффициентов второго слоя имеет размер $K \times K$, определяется как

$$\begin{bmatrix} 1 & -\varepsilon & \cdots & -\varepsilon \\ -\varepsilon & 1 & \cdots & -\varepsilon \\ \cdots & \cdots & \cdots & \cdots \\ -\varepsilon & -\varepsilon & \cdots & 1 \end{bmatrix}, \tag{2.13}$$

где $\varepsilon \in \left(0, \frac{1}{K}\right]$.

Таким образом, обучение производится за один цикл.

Работа сети. На вход подается классифицируемый вектор $\overrightarrow{x^*}$. Состояние нейронов первого слоя рассчитывается как $s_{1j} = w_{ji} x_i^*$.

Выход нейронов первого слоя получается путем применения функции активации к состоянию, и становится начальным значением соответствующих нейронов второго слоя. Далее, состояния нейронов второго слоя получаются из их предыдущего состояния, исходя из матрицы весовых коэффициентов второго слоя, и процедура повторяется итерационно до стабилизации вектора состояния второго слоя – пока норма разницы векторов двух последовательных итераций не станет меньше определенного значения $E_{\max}$ (на практике достаточно значений порядка 0,1).

В случае, если в итоге один вектор положительный, а остальные отрицательные, то он указывает на подходящий образец. В случае же, если несколько векторов положительны, и при этом не один из них не превышает $E_{\max}$, то это значит, что нейросеть не может отнести входящий вектор ни к одному из классов, однако положительные выходы указывают на наиболее схожие эталоны.

2.1.3. Сети Элмана и Джордана

Нейронная сеть Элмана – один из видов рекуррентной сети, которая так же как и сеть Джордана получается из многослойного перцептрона введением обратных связей, только связи идут не от выхода сети, а от выходов внутренних нейронов [13]. Это позволяет учесть предысторию наблюдаемых процессов и накопить информацию для выработки правильной стратегии управления.

Рекуррентная сеть Элмана дана на рис. 2.4.

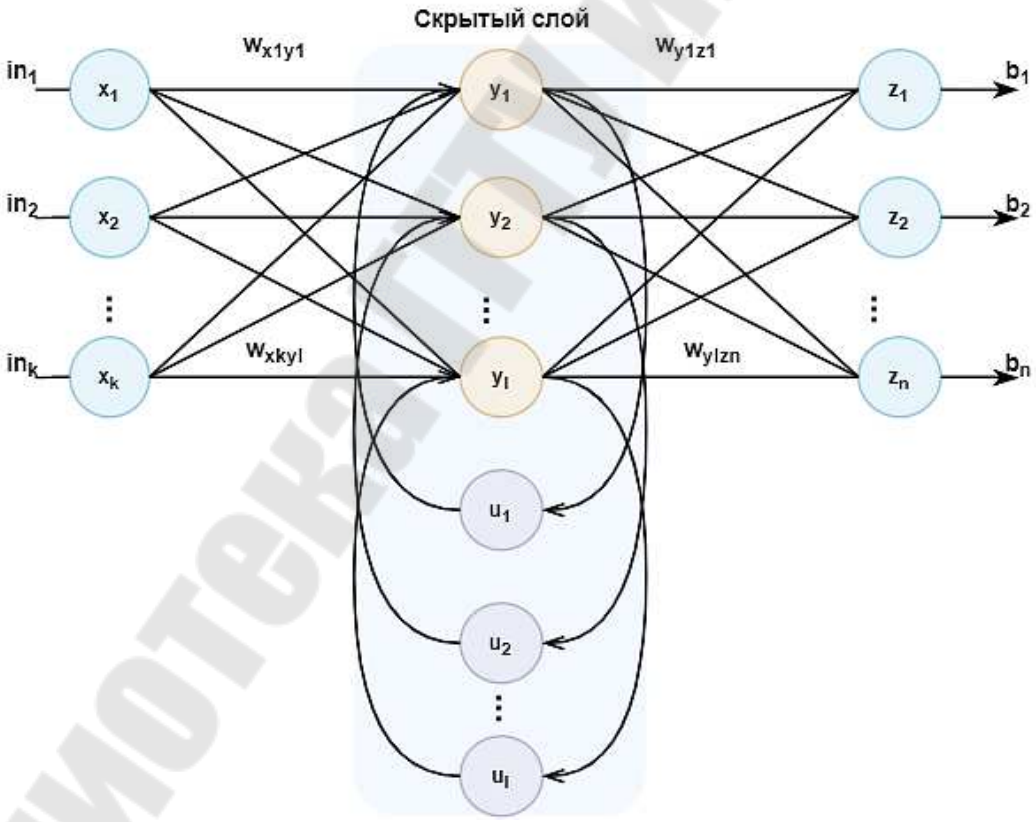


Рис. 2.4. Рекуррентная сеть Элмана

Нейронная сеть Элмана представляет из себя трехслойную нейронную сеть, слои которой на рис. 2.4 обозначены как *x*, *y* и *z*. Дополнительно к сети добавлен набор «контекстных блоков» (*u* на рис. 2.4).

Средний (скрытый) слой соединен с контекстными блоками с фиксированным весом, равным единице. С каждым шагом времени на вход поступает информация, которая проходит прямой ход к выходному слою в соответствии с правилами обучения. Фиксированные обратные связи сохраняют предыдущие значения скрытого слоя в контекстных блоках (до того как скрытый слой поменяет значение в процессе обучения). Таким способом сеть сохраняет свое состояние, что может использоваться в предсказании последовательностей, выходя за пределы мощности многослойного перцептрона.

Рекуррентная сеть Элмана характеризуется частичной рекуррентностью в форме обратной связи между скрытым и входным слоем, реализуемой с помощью единичных элементов запаздывания z^{-1}. Обобщенная структура этой сети представлена на рис. 2.5. Каждый скрытый нейрон имеет свой аналог в контекстном слое, образующем совместно с внешними входами сети входной слой. Выходной слой состоит из нейронов, однонаправленно связанных только с нейронами скрытого слоя, подобно сети *RMLP*. Обозначим внутренний вектор возбуждения сети x (в его состав входит также единичный сигнал поляризации), состояния скрытых нейронов – $v \in R^K$, а выходные сигналы сети – $y \in R^M$. При таких обозначениях входной вектор сети в момент *t* имеет форму

$$x(k) = [x_0(k), x_1(k), \ldots, x_N(k), v_1(k-1), v_2(k-1), \ldots, v_K(k-1)]. \quad (2.14)$$

Веса синаптических связей первого (скрытого) слоя сети обозначим $w_{ij}^{(1)}$, второго (выходного) слоя – $w_{ij}^{(2)}$. Если взвешенную сумму *i*-го нейрона скрытого слоя обозначить u_i, а его выходной сигнал – ν_i, то

$$u_i(k) = \sum_{j=0}^{N+K} w_{ij}^{[1]} x_j(k), \quad (2.15)$$

$$\nu_i(k) = f_1(u_i(k)). \quad (2.16)$$

Веса $w_{ij}^{(1)}$ образуют матрицу $W^{(1)}$ синаптических связей скрытого слоя, а $f_1(u_i)$ – функция активации *i*-го нейрона этого слоя. Аналогично можно обозначить взвешенную сумму *i*-го нейрона выходного слоя g_i, а соответствующий ему выходной сигнал сети – y_i. Эти сигналы описываются формулами

$$g_i(k) = \sum_{j=0}^{K} w_{ij}^{[2]} v_j(k); \quad (2.17)$$

$$y_i(k) = f_2(g_i(k)). \quad (2.18)$$

В свою очередь, веса $w_{ij}^{(2)}$ образуют матрицу $W^{(2)}$, описывающую синаптические связи нейронов выходного слоя, а $f_2(g_i)$ – функция активации *i*-го нейрона выходного слоя.

В общем случае можно рассмотреть объединенную сеть Джордана–Элмана.

Сеть Джордана – этот вид сетей получается из многослойного перцептрона, если на его вход подать помимо входного вектора выходной с задержкой на один или несколько тактов [13].

В первых рекуррентных сетях главной идеей было дать сети видеть свой выходной образ на предыдущем шаге. У такой сети только часть рецепторов принимает сигналы из окружающего мира, на другие рецепторы приходит выходной образ из предыдущего момента времени. Рассмотрим прохождение последовательности сигналов через сеть. Сигнал поступает на группу рецепторов, соединенных с внешним миром (*INPUT*), и проходит в скрытый слой (*HIDDEN*). Преобразованный скрытым слоем сигнал пойдет на выходной слой (*OUTPUT*) и выйдет из сети, а его копия попадет на задержку. Далее в сеть, на рецепторы, воспринимающие внешние сигналы, поступает второй образ, а на контекстную группу рецепторов (*CONTEXT*) – выходной образ с предыдущего шага из задержки. Далее со всех рецепторов сигнал пойдет в скрытый слой, затем – на выходной.

Нейронная сеть Джордана подобна сети Элмана. Однако контекстные блоки связаны не со скрытым слоем, а с выходным слоем. Контекстные блоки таким образом сохраняют свое состояние. Они обладают рекуррентной связью с собой.

Один из простейших способов построения рекуррентной сети на базе однонаправленной нейронной сети состоит во введении в персептронную сеть обратной связи. Ее обобщенная структура представлена на рис. 2.5.

Это динамическая сеть, характеризующаяся запаздыванием входных и выходных сигналов, объединяемых во входной вектор сети.

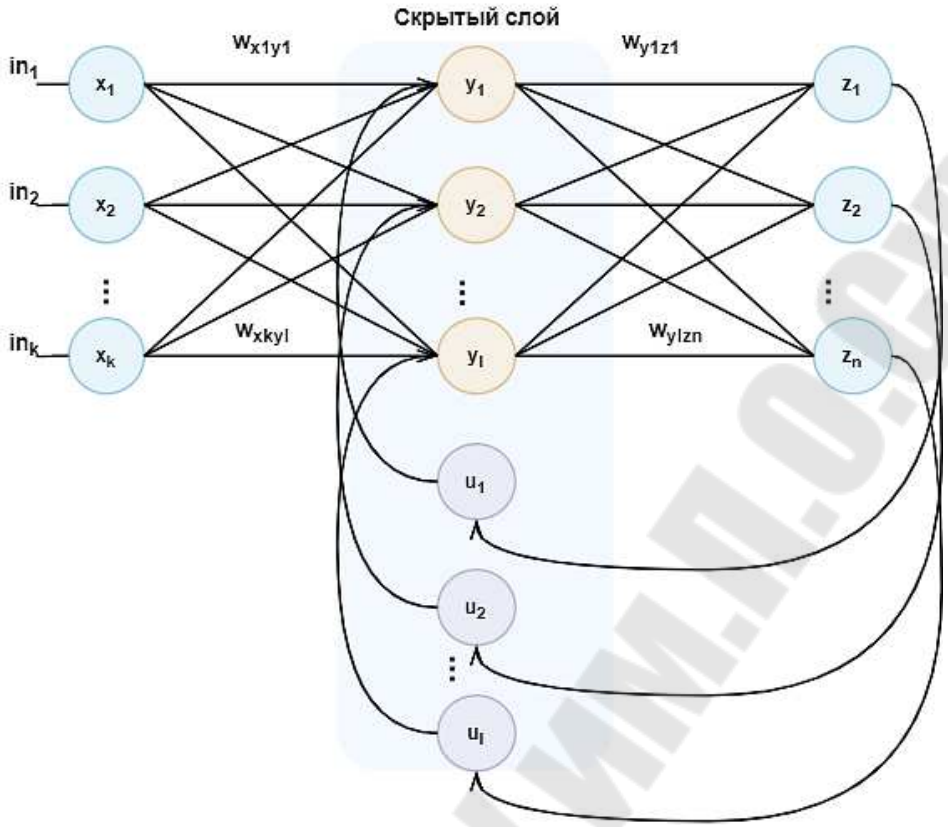


Рис. 2.5. Рекуррентная сеть Джордана

Сети Элмана и Джордана называют также «простыми рекуррентными сетями» (*SRN*).

Сеть Элмана:

$$h_t = \sigma_h\left(W_h x_t + U_h h_{t-1} + b_h\right); \tag{2.19}$$

$$y_t = \sigma_y\left(W_y h_t + b_y\right). \tag{2.20}$$

Сеть Джордана:

$$h_t = \sigma_h\left(W_h x_t + U_h y_{t-1} + b_h\right); \tag{2.21}$$

$$y_t = \sigma_y\left(W_y h_t + b_y\right), \tag{2.22}$$

где x_t – вектор входного слоя; h_t – вектор скрытого слоя; y_t – вектор выходного слоя; W, U и b – матрица и вектор параметров; σ_h и σ_y – функция активации.

2.2. Стохастические нейронные сети и алгоритмы

2.2.1. Стохастический нейрон

Стохастические нейроны – это модель где переключение нейрона происходит с вероятностью, зависящей от индуцированного локального поля, то есть передаточная функция определена как:

$$f(u)=\begin{cases}1 & \text{с вероятностью } P(u),\\ 0 & \text{с вероятностью } 1-P(u),\end{cases} \tag{2.23}$$

где распределение вероятности $P(u)$ обычно имеет вид сигмоида:

$$\sigma(u)=\frac{A(T)}{1+\exp(-u/T)}, \tag{2.24}$$

а нормировочная константа $A(T)$ вводится для условия нормализации распределения вероятности $\int_0^1 \sigma(u)du=1$. Таким образом, нейрон активируется с вероятностью $P(u)$. Параметр T – аналог температуры (но не температуры нейрона) и определяет беспорядок в нейронной сети. Если T устремить к 0, стохастический нейрон перейдет в обычный нейрон с передаточной функцией Хевисайда (пороговой функцией).

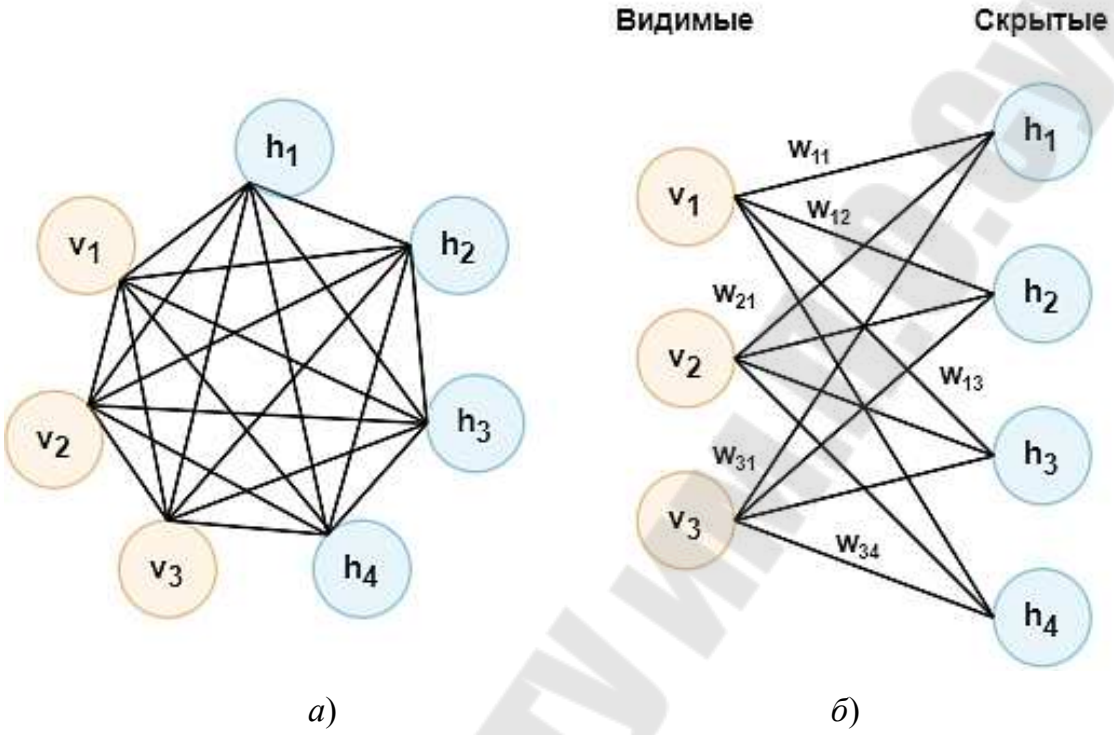
2.2.2. Полносвязанная модель машины Больцмана

Машина Больцмана (англ. *Boltzmann machine*) (рис. 2.6, *а*) – вид стохастической рекуррентной нейронной сети, изобретенной Джеффри Хинтоном и Терри Сейновски в 1985 г. Машина Больцмана может рассматриваться как стохастический генеративный вариант сети Хопфилда.

Специалисты по статистике называют такие сети случайными марковскими полями. Сеть названа машиной Больцмана в честь австрийского физика Людвига Больцмана, одного из создателей статистической физики.

Эта сеть использует для обучения алгоритм имитации отжига и является первой нейронной сетью, способной обучаться внутренним представлениям, решать сложные комбинаторные задачи. Несмотря на это, из-за ряда проблем машины Больцмана с неограниченной связностью не могут применяться для решения практических проблем. Если же связность ограничена, то обучение может быть доста-

точно эффективным для использования на практике. В частности, из каскада ограниченных машин Больцмана строится так называемая глубокая сеть доверия.



а) *б*)

Рис. 2.6. Схематичное изображение:
а – полносвязной машины Больцмана;
б – ограниченной машины Больцмана

Модель. Как и сеть Хопфилда, машина Больцмана является сетью нейронов с определенным для нее понятием «энергии». Расчет глобальной энергии производится идентичным с сетью Хопфилда образом:

$$E = -\sum_{i<j} w_{ij} s_i s_j - \sum_i \theta_i s_i, \qquad (2.25)$$

где w_{ij} – сила связи между нейронами j и i; s_i состояние нейрона i, где w_{ij} сила связи между нейронами j и i; s_i – состояние нейрона $i, s_i \in \{0, 1\}$; θ_i – порог для нейрона i.

Связи имеют следующие ограничения:

– $w_{ii} = 0 \quad \forall i$ (нейрон не может иметь связь с самим собой);

– $w_{ij} = w_{ji} \quad \forall i, j$ (все связи являются симметричными).

Вероятность состояния блока. Разница в глобальной энергии, получаемой из одной единицы *i*, равной 0 (выключено), и 1 (включено), записанной ΔE_i в предположении симметричной матрицы весов задается как

$$\Delta E_i = \sum_{j>i} w_{ij} s_j + \sum_{j<i} w_{ji} s_j + \theta_i. \tag{2.26}$$

Это может быть выражено в виде разницы энергий двух состояний:

$$\Delta E_i = E_{i=off} - E_{i=on}. \tag{2.27}$$

Подстановка энергии каждого состояния вместе с его относительной вероятностью по коэффициенту Больцмана дает:

$$\Delta E_i = -k_B T \ln\left(p_{i=off}\right) - \left(-k_B T \ln\left(p_{i=on}\right)\right), \tag{2.28}$$

где k_B является константой Больцмана и поглощается в искусственное понятие температуры *T*. Затем происходит перестановка терминов, и считается, что вероятности включения и выключения элемента должны суммироваться в единицу:

$$\frac{\Delta E_i}{T} = \ln\left(p_{i=on}\right) - \ln\left(p_{i=off}\right); \tag{2.29}$$

$$\frac{\Delta E_i}{T} = \ln\left(p_{i=on}\right) - \ln\left(1 - p_{i=on}\right); \tag{2.30}$$

$$\exp\left(-\frac{\Delta E_i}{T}\right) = \frac{1}{p_{i=on}} - 1. \tag{2.31}$$

Решением для $p_{i=on}$ является то, что *i*-й элемент включен:

$$p_{i=on} = \frac{1}{1 + \exp\left(-\frac{\Delta E_i}{T}\right)}, \tag{2.32}$$

где скаляр *T* называется температурой системы. Это отношение становится источником логистической функции, содержащейся в выражениях вероятности в вариантах машины Больцмана.

Один из основных недостатков сети Хопфилда – это тенденция «стабилизации» состояния сети в локальном, а не в глобальном минимуме. Желательно, чтобы сеть переходила в глубокие минимумы энер-

гии чаще, чем неглубокие, при этом относительная вероятность перехода сети в один из двух минимумов с разной энергией зависела только от соотношения их глубин. Это позволило бы управлять вероятностями получения конкретных выходных векторов состояния путем изменения профиля энергетической поверхности системы за счет модификации весов связей. На основе этих соображений и построена машина Больцмана.

Идея использования «теплового шума» для выхода из локальных минимумов и повышения вероятности попадания в более глубокие минимумы принадлежит С. Кирпатрику. На основе этой идеи разработан алгоритм имитации отжига.

Введем некоторый параметр t – аналог уровня теплового шума. Тогда вероятность активности некоторого нейрона k определяется на основе вероятностной функции Больцмана:

$$Pk = 1/\left(1 + e^{-E_k / t}\right), \tag{2.33}$$

где t – уровень теплового шума в сети; E_k – сумма весов связей k-го нейрона со всеми активными в данный момент нейронами.

Обучение. Единицы в машине Больцмана делятся на «видимые» единицы V и «скрытые» единицы H. Видимые единицы – это те, которые получают информацию из «окружающей среды», т. е. обучающий набор представляет собой набор двоичных векторов над множеством V. Распределение по обучающей выборке обозначено как $P^{+}(V)$.

Распределение по глобальным состояниям сходится по мере того, как машина Больцмана достигает теплового равновесия. Это распределение обозначается после его преобразования в скрытые единицы $P^{-}(V)$.

Цель этой сети – предсказать «реальное» распределение $P^{+}(V)$ с помощью $P^{-}(V)$, произведенного машиной. Сходство двух распределений измеряется расхождением Куллбек–Лейблера G:

$$G = \sum_{v} P^{+}(v) \ln\left(\frac{P^{+}(v)}{P^{-}(v)}\right), \tag{2.34}$$

где сумма находится над всеми возможными состояниями V; G является функцией весов, так как они определяют энергию состояния, а энергия определяет $P^{-}(v)$, как обещано в распределении Больцмана.

Алгоритм градиентного спуска через G изменяет заданный вес w_{ij} путем вычитания частичной производной G по отношению к весу.

Обучение машины Больцмана включает в себя две чередующиеся фазы. Одна из них – «положительная» фаза, в которой состояния видимых элементов фиксируются на определенном двоичном векторе состояния, отобранном из обучающего комплекта (согласно P^+). Другая – «отрицательная» фаза, в которой сети разрешено свободно функционировать, т. е. ни один элемент не имеет своего состояния, определяемого внешними данными. Градиент относительно заданного веса w_{ij} задается уравнением

$$\frac{\partial G}{\partial w_{ij}} = -\frac{1}{R}\left[p_{ij}^{+} - p_{ij}^{-}\right], \tag{2.35}$$

где p_{ij}^{+} – это вероятность того, что элементы i и j включены, когда машина находится в равновесии на положительной фазе; p_{ij}^{-} – вероятность того, что элементы i и j включены, когда машина находится в состоянии равновесия на отрицательной фазе; R обозначает скорость обучения.

Этот результат вытекает из того, что при тепловом равновесии значение $P^{-}(s)$ любого глобального состояния s, когда сеть свободно работает, определяется распределением Больцмана.

Это правило обучения биологически правдоподобно, поскольку единственной информацией, необходимой для изменения весов, является «локальная» информация. То есть соединение (синапс) не нуждается в информации ни о чем, кроме двух нейронов, с которыми оно соединяется. Это более реалистично с биологической точки зрения, чем информация, необходимая соединению во многих других алгоритмах обучения нейронов, таких, как обратное распространение.

Обучение смещениям аналогично, но использует только активность отдельных узлов:

$$\frac{\partial G}{\partial \theta_i} = -\frac{1}{R}\left[p_i^{+} - p_i^{-}\right]. \tag{2.36}$$

2.2.3. Ограниченная машина Больцмана

Ограниченная машина Больцмана (англ. *restricted Boltzmann machine*), сокращенно *RBM* – вид генеративной стохастической

нейронной сети, которая определяет распределение вероятности на входных образцах данных.

У *RBM* имеется ряд состояний, которые можно наблюдать (видимые нейроны, которые предоставляют интерфейс для общения с внешней средой) и ряд состояний, которые скрыты (скрытые нейроны). Но возможно сделать вероятностный вывод относительно скрытых состояний, опираясь на состояния, которые можно наблюдать.

После обучения такой модели появляется возможность делать выводы относительно видимых состояний, зная скрытые, и тем самым генерировать данные из того вероятностного распределения, на котором обучена модель.

Таким образом, цель обучения *RBM* – это настройка параметров 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 обозреваемых состояний, т. е. из оригинального вектора.

Особенностью ограниченных машин Больцмана является возможность проходить обучение без учителя, но в определенных приложениях ограниченные машины Больцмана обучаются с учителем. Скрытый слой машины представляет собой глубокие признаки в данных, которые выявляются в процессе обучения.

Структура сети. Ограниченная машина Больцмана базируется на бинарных элементах с распределением Бернулли, составляющих видимый v_i и скрытый h_j слои сети (рис. 2.6, *б*). Связи между слоями задаются с помощью матрицы весов $W = \left(w_{i,j}\right)$ (размера $m \times n$), а также смещений a_i – для видимого слоя и b_j – для скрытого слоя.

Вводится понятие «энергии» сети (v, h) как

$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_i \sum_j v_i w_{i,j} h_j, \qquad (2.37)$$

или в матричной форме:

$$E(v,h) = -a^T v - b^T h - v^T W h. \qquad (2.38)$$

Подобной функцией энергии обладает также сеть Хопфилда. Как и для обычной машины Больцмана, через энергию определяется вероятность распределения на векторах видимого и скрытого слоев:

$$P(v,h)=\frac{1}{Z}e^{-E(v,h)}, \tag{2.39}$$

где Z – статсумма, определяемая как $\sum e^{-E(v,h)}$ для всех возможных сетей (иными словами, Z – константа нормализации, которая гарантирует, что сумма всех вероятностей равна единице). Определение вероятности для отдельного входного вектора (маргинальное распределение) проводится аналогично через сумму конфигураций всевозможных скрытых слоев:

$$P(v))=\frac{1}{Z}\sum_{h}e^{-E(v,h)}. \tag{2.40}$$

По причине структуры сети как двудольного графа отдельные элементы скрытого слоя независимы друг от друга и активируют видимый слой, и наоборот, отдельные элементы видимого слоя независимы друг от друга и активируют скрытый слой. Для m видимых элементов и для n скрытых элементов условные вероятности определяются через произведения вероятностей h:

$$P(v\mid h)=\prod_{i=1}^{m}P(v_i\mid h), \tag{2.41}$$

и обратно, условные вероятности h определяются через произведение вероятностей v:

$$P(h\mid v)=\prod_{j=1}^{n}P(h_j\mid v). \tag{2.42}$$

Конкретные вероятности активации для одного элемента определяются как

$$P(h_j=1\mid v)=\sigma\left(b_j+\sum_{i=1}^{m}w_{i,j}v_i\right); \tag{2.43}$$

$$P(v_i=1\mid h)=\sigma\left(a_i+\sum_{j=1}^{n}w_{i,j}h_j\right), \tag{2.44}$$

где σ – логистическая функция для активации слоя.

Видимые слои могут иметь также мультиномиальное распределение, в то время как скрытые слои распределены по Бернулли.

В случае мультиномиальности вместо логистической функции используется *softmax*:

$$P\left(v_i^k = 1 \mid h\right) = \frac{\exp\left(a_i^k + \Sigma_j W_{ij}^k h_j\right)}{\sum_{k'=1}^{K} \exp\left(a_i^{k'} + \Sigma_j W_{ij}^{k'} h_j\right)}, \tag{2.45}$$

где K – количество дискретных значений видимых элементов. Такое представление используется в задачах тематического моделирования и в рекомендательных системах.

Алгоритм обучения. Целью обучения является максимизация вероятности системы с заданным набором образцов V (матрицы, в которой каждая строка соответствует одному образцу видимого вектора v), определяемой как произведение вероятностей:

$$\operatorname{argmax}_W \prod_{v \in V} P(v), \tag{2.46}$$

или же, что одно и то же, максимизации логарифма произведения:

$$\operatorname{argmax}_W \mathbb{E}\left[\log P(v)\right]. \tag{2.47}$$

Для тренировки нейронной сети используется алгоритм контрастивной дивергенции (*CD*) с целью нахождения оптимальных весов матрицы W, его предложил Джеффри Хинтон первоначально для обучения моделей *PoE* («произведение экспертных оценок»).

В целом один шаг контрастивной дивергенции (*CD*-1) выглядит следующим образом:

– для одного образца данных v вычисляются вероятности скрытых элементов и применяется активация для скрытого слоя h для данного распределения вероятностей;

– вычисляется внешнее произведение (семплирование) для v и h, которое называют «позитивным градиентом»;

– через образец h проводится реконструкция образца видимого слоя v', а потом выполняется снова семплирование с активацией скрытого слоя h'. (Этот шаг называется «семплирование по Гиббсу»);

– далее вычисляется внешнее произведение, но уже векторов v' и h', которое называют «негативным градиентом»;

– матрица весов W поправляется на разность позитивного и негативного градиента, помноженного на множитель, задающий скорость обучения:

$$\Delta W = \varepsilon\left(vh^T - v'h'^T\right); \qquad (2.48)$$

– вносятся поправки в биасы a и b похожим способом:

$$\Delta a = \varepsilon\left(v - v'\right), \;\; \Delta b = \varepsilon\left(h - h'\right). \qquad (2.49)$$

2.3. Сети с латеральными связями

2.3.1. Слой и самоорганизующаяся карта Кохонена

Нейронные сети Кохонена (рис. 2.7) – класс нейронных сетей, основным элементом которых является слой Кохонена [4]. Слой Кохонена состоит из адаптивных линейных сумматоров («линейных формальных нейронов»). Как правило, выходные сигналы слоя Кохонена обрабатываются по правилу «победитель получает все»: наибольший сигнал превращается в единичный, остальные обращаются в ноль.

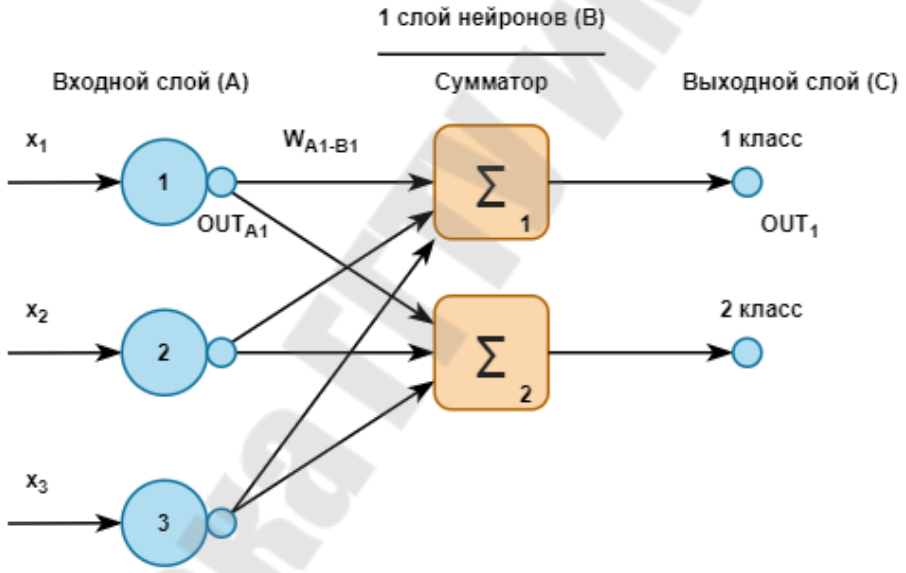


Рис. 2.7. Схема нейронной сети Кохонена

По способам настройки входных весов сумматоров и по решаемым задачам различают много разновидностей сетей Кохонена. Наиболее известные из них:

– сети векторного квантования сигналов, тесно связанные с простейшим базовым алгоритмом кластерного анализа (метод динамических ядер или *K*-средних [14]);

– самоорганизующиеся карты Кохонена (англ. *self-organising maps – SOM*);

– сети векторного квантования, обучаемые с учителем (англ. *learning vector quantization*).

Слой Кохонена. *Базовая версия.* Слой Кохонена состоит из некоторого количества n параллельно действующих линейных элементов. Все они имеют одинаковое число входов m и получают на свои входы один и тот же вектор входных сигналов $x = (x_1, ..., x_m)$. На выходе j-го линейного элемента получаем сигнал

$$y_j = w_{j0} + \sum_{i=1}^{m} w_{ji} x_i, \qquad (2.50)$$

где w_{ji} – весовой коэффициент i-го входа j-го нейрона; I – номер входа; j – номер нейрона; w_{j0} – пороговый коэффициент.

После прохождения слоя линейных элементов сигналы посылаются на обработку по правилу «победитель забирает все»: среди выходных сигналов выполняется поиск максимального y_j; его номер $j_{\max} = \arg\max_j \{y_j\}$. Окончательно, на выходе сигнал с номером $j_{\max}$ равен единице, остальные – нулю. Если максимум одновременно достигается для нескольких $j_{\max}$, то:

– либо принимают все соответствующие сигналы равными единице;

– либо равным единице принимают только первый сигнал в списке (по соглашению).

Геометрическая интерпретация. Большое распространение получили слои Кохонена, построенные следующим образом: каждому (j-му) нейрону сопоставляется точка $W_j = (w_{j1}, ..., w_{jm})$ в m-мерном пространстве (пространстве сигналов). Для входного вектора $x = (x_1, ..., x_m)$ вычисляются его евклидовы расстояния $\rho_j(x))$ до точек W_j и «ближайший получает все» – тот нейрон, для которого это расстояние минимально, выдает единицу, остальные – нули. Следует заметить, что для сравнения расстояний достаточно вычислять линейную функцию сигнала:

$$\rho_j(x)^2 = \| x - W_j \|^2 = \| W_j \|^2 - 2\sum_{i=1}^{m} w_{ji} x_i + \| x \|^2, \qquad (2.51)$$

где $\| y \|$ – евклидова длина вектора $\| y \|^2 = \sum_i y_i^2$ (2.52)

Последнее слагаемое $\| x \|^2$ одинаково для всех нейронов, поэтому для нахождения ближайшей точки оно не нужно. Задача сводится к поиску номера наибольшего из значений линейных функций:

$$j_{\max} = \arg\max_j \left\{ \sum_{i=1}^{m} w_{ji} x_i - \frac{1}{2} \| W_j \|^2 \right\}, \qquad (2.53)$$

Таким образом, координаты точки $W_j = (w_{j1}, ..., w_{jm})$ совпадают с весами линейного нейрона слоя Кохонена (при этом значение порогового коэффициента $w_{j0} = - \| W_j \|^2 / 2$).

Если заданы точки $W_j = (w_{j1}, ..., w_{jm})$, то m-мерное пространство разбивается на соответствующие многогранники Вороного–Дирихле V_j: многогранник V_j состоит из точек, которые ближе к W_j, чем к другим $W_k (k \neq j)$.

Самоорганизующаяся карта Кохонена. Самоорганизующаяся карта Кохонена (англ. *Self-organizing map – SOM)* – нейронная сеть с обучением без учителя, выполняющая задачу визуализации и кластеризации. Идея сети предложена финским ученым Т. Кохоненом. Является методом проецирования многомерного пространства в пространство с более низкой размерностью (чаще всего двумерное), применяется также для решения задач моделирования, прогнозирования, выявления наборов независимых признаков, поиска закономерностей в больших массивах данных, разработки компьютерных игр, квантизации цветов к их ограниченному числу индексов в цветовой палитре (при печати на принтере и ранее на ПК или же на приставках с дисплеем с пониженным числом цветов, для архиваторов или видеокодеков). Это одна из версий нейронных сетей Кохонена [6].

Самоорганизующаяся карта состоит из компонентов, называемых узлами или нейронами. Их количество задается аналитиком. Каждый из узлов описывается двумя векторами. Первый – так называемый вектор веса *m*, имеющий такую же размерность, что и входные данные. Второй – вектор *r*, представляющий собой координаты узла на карте. Карта Кохонена визуально отображается с помощью ячеек прямоугольной или шестиугольной формы; последняя применяется чаще, поскольку в этом случае расстояния между центрами смежных ячеек одинаковы, что повышает корректность визуализации карты.

Изначально известна размерность входных данных, по ней некоторым образом строится первоначальный вариант карты. В процессе обучения векторы веса узлов приближаются к входным данным. Для каждого наблюдения (семпла) выбирается наиболее похожий по вектору веса узел, и значение его вектора веса приближается к наблюдению. Также к наблюдению приближаются векторы веса нескольких узлов, расположенных рядом, поэтому, если в множестве входных данных два наблюдения были схожи, на карте им будут соответствовать близкие узлы. Циклический процесс обучения, перебирающий входные данные, заканчивается по достижении картой допустимой (заранее заданной аналитиком) погрешности, или по совершении заданного количества итераций. Таким образом, в результате обучения карта Кохонена классифицирует входные данные на кластеры и визуально отображает многомерные входные данные в двумерной плоскости, распределяя векторы близких признаков в соседние ячейки и раскрашивая их в зависимости от анализируемых параметров нейронов.

В результате работы алгоритма получаются следующие карты:

– *карта входов нейронов* – визуализирует внутреннюю структуру входных данных путем подстройки весов нейронов карты. Обычно используется несколько карт входов, каждая из которых отображает один из них и раскрашивается в зависимости от веса нейрона. На одной из карт определенным цветом обозначают область, в которую включаются приблизительно одинаковые входы для анализируемых примеров;

– *карта выходов нейронов* – визуализирует модель взаимного расположения входных примеров. Очерченные области на карте представляют собой кластеры, состоящие из нейронов со схожими значениями выходов;

– *специальные карты* – это карта кластеров, полученных в результате применения алгоритма самоорганизующейся карты Кохонена, а также другие карты, которые их характеризуют.

Работа сети построена следующим образом:

1. Инициализация карты, т. е. первоначальное задание векторов веса для узлов.

2. Цикл:

– выбор следующего наблюдения (вектора из множества входных данных);

– нахождение для него лучшей единицы соответствия (*best matching unit*, *BMU*, или *Winner*) – узла на карте, вектор веса которого

меньше всего отличается от наблюдения (в метрике, задаваемой аналитиком, чаще всего, евклидовой);

– определение количества соседей *BMU* и обучение – изменение векторов веса *BMU* и его соседей с целью их приближения к наблюдению;

– определение ошибки карты.

Алгоритм:

Инициализация. Наиболее распространены три способа задания первоначальных весов узлов:

– задание всех координат случайными числами;

– присваивание вектору веса значения случайного наблюдения из входных данных;

– выбор векторов веса из линейного пространства, натянутого на главные компоненты набора входных данных.

Цикл:

– пусть t – номер итерации (инициализация соответствует номеру 0);

– выбрать произвольное наблюдение $x(t)$ из множества входных данных;

– найти расстояния от него до векторов веса всех узлов карты и определить ближайший по весу узел $M_c(t)$. Это – *BMU* или *Winner*. Условие на $M_c(t)$:

$$\|x(t) - m_c(t)\| \le \|x(t) - m_i(t)\|, \tag{2.54}$$

для любого $m_i(t)$, где $m_i(t)$ – вектор веса узла $M_i(t)$. Если находится несколько узлов, удовлетворяющих условию, *BMU* выбирается случайным образом среди них;

– определить с помощью функции h (функции соседства) соседей M_c и изменение их векторов веса. Функция определяет «меру соседства» узлов M_i и M_c и изменение векторов веса. Она должна постепенно уточнять их значения, сначала у большего количества узлов и сильнее, потом у меньшего и слабее. Часто в качестве функции соседства используется гауссовская функция:

$$h_{ci}(t) = \alpha(t)\exp\left(-\frac{\| r_c - r_i \|^2}{2\sigma^2(t)}\right), \tag{2.55}$$

где $0 < \alpha(t) < 1$ – обучающий сомножитель, монотонно убывающий с каждой последующей итерацией (т. е. определяющий приближение

значения векторов веса *BMU* и его соседей к наблюдению; чем больше шаг, тем меньше уточнение); r_i, r_c – координаты узлов $M_i(t)$ и $M_c(t)$ на карте; $\sigma(t)$ – сомножитель, уменьшающий количество соседей с итерациями, монотонно убывает.

Параметры α, σ и их характер убывания задаются аналитиком.

Более простой способ задания функции соседства $h_{ci}(t) = \alpha(t)$, если $M_i(t)$ находится в окрестности $M_c(t)$ заранее заданного аналитиком радиуса, и 0 – в противном случае.

Функция $h(t)$ равна $\alpha(t)$ для *BMU* и уменьшается с удалением от *BMU*;

– изменить вектор веса по формуле

$$m_i(t) = m_i(t-1) + h_{ci}(t)(x(t) - m_i(t-1)). \quad (2.56)$$

Таким образом, вектора веса всех узлов, являющихся соседями *BMU*, приближаются к рассматриваемому наблюдению;

– вычислить ошибки карты. Например, как среднее арифметическое расстояний между наблюдениями и векторами веса соответствующих им *BMU*:

$$\frac{1}{N}\sum_{i=1}^{N}\|x_i - m_c\|, \quad (2.57)$$

где N – количество элементов набора входных данных.

Существуют недостатки карт Кохонена:

– субъективность. Не всегда ясно, какие особенности карты обусловлены кластерной структурой данных, а какие – свойствами сглаживающего ядра. От выбора параметра β существенно зависит сглаженность границ кластеров и степень детализации карты;

– наличие искажений. Близкие объекты исходного пространства могут переходить в далекие точки на карте. В частности, объективно существующие кластеры могут разрываться на фрагменты, и наоборот, далекие объекты могут случайно оказаться на карте рядом, особенно, если они были одинаково далеки от всех кластеров. Искажения неизбежны при проецировании многомерной выборки на плоскость. Распределение точек на карте позволяет судить лишь о локальной структуре многомерных данных, и то не всегда;

– зависимость от инициализации. Начальное распределение весов существенно влияет на процесс обучения и может сказываться

не только на расположении кластеров, но даже на их количестве. Иногда рекомендуется построить несколько карт и выбрать из них наиболее «удачную».

2.3.2. Конкурентное обучение. Правило Кохонена

Конкурентное обучение. Конкурентное обучение является формой обучения без учителя в искусственных нейронных сетях, в которых узлы конкурируют за право отвечать на подмножества входных данных. Конкурентные способы обучения работают за счет увеличения специализации каждого узла в сети. Это хорошо подходит для нахождения кластеров в пределах данных.

Модели и алгоритмы, основанные на принципе конкурентного обучения, включают векторное квантование и самоорганизующиеся карты Кохонена.

Конкурентное обучение представляет собой итерационный процесс, в котором на каждой итерации выполняется три действия:

– конкуренция;

– объединение;

– подстройка весов.

При конкуренции на вход сети подается входной вектор признаков и производится поиск нейрона с наиболее близким к нему набором весов. Такой нейрон объявляется победителем. В процессе объединения вокруг него образуется группа (соседство) нейронов, которые будут участвовать в процессе обучения. Размер группы определяется радиусом обучения.

Модель конкурентного обучения дана на рис. 2.8.

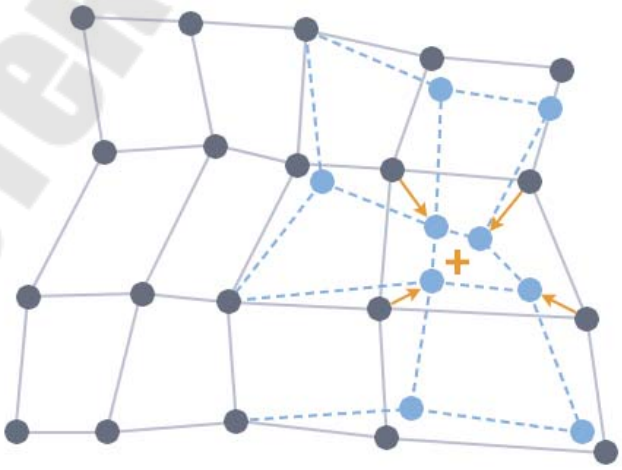


Рис. 2.8. Модель конкурентного обучения

На этапе подстройки происходит корректировка весов нейронов, расположенных в пределах радиуса обучения от нейрона-победителя, таким образом, чтобы их векторы стали ближе к нему.

Основные принципы:

– набор одинаковых нейронов со случайно распределенными синоптическими весами, приводящими к разной реакции нейронов на заданный набор входных шаблонов;

– ограничение наложено на значение «силы» каждого нейрона;

– механизм, который позволяет нейронам конкурировать за право реагировать на подмножество входных данных, устроен таким образом, что только один выходной нейрон (или только один нейрон в группе) является активным (т. е. «включен») за один раз. Нейрон, который побеждает в соревновании, называется нейроном «победитель получает все».

Соответственно, отдельные нейроны в сети учатся специализироваться на ансамблях подобных моделей и тем самым становятся «детекторами» для различных классов входных шаблонов.

Тот факт, что конкурентоспособные сети перекодируют множество коррелированных входов к одному из нескольких выходных нейронов, по сути устраняет избыточность в представлении, что является важной частью обработки в биологических сенсорных системах.

Архитектура и реализация. Конкурентное обучение обычно реализуется с нейронными сетями, которые содержат скрытый слой известный как «конкурентный слой». Каждый конкурентный нейрон описывается вектором весов $\mathbf{w}_i = (w_{i1}, ..., w_{id})^T$, $i = 1, ..., M$ и вычисляет коэффициент сходства между входными данными $\mathbf{x}^n = (x_{n1}, ..., x_{nd})^T \in R^d$ и весовым вектором $\mathbf{w}_i$.

Для каждого входного вектора конкурентные нейроны «соревнуются» друг с другом, чтобы определить, какой из них является наиболее близким к этому конкретному входному вектору. Нейрон победителя устанавливает свой выход $o_m = 1,$ а все остальные конкурентные нейроны устанавливают свой выход $o_i = 0,\ i = 1, ..., M,$ $i \neq m.$

Обычно для измерения сходства используется обратное евклидово расстояние $\|\mathbf{x} - \mathbf{w}_i\|$: между входным вектором $\mathbf{x}^n$ и весовым вектором $\mathbf{w}_i$.

Правило Кохонена. Обучение осуществляется без учителя по принципу «победитель забирает все», т. е. для заданного входного образа активизируется лишь один нейрон. Такие методы относятся к обучению методом соревнования. Вследствие этого нейронные сети Кохонена используют для решения задач классификации входных образов (разбиения на классы).

Эффект этого правила достигается за счет такого изменения сохраненного в сети образца (вектора весов связей победившего нейрона), при котором он становится чуть ближе к входному примеру.

Правило Кохонена. Вектор весов входных синапсов нейрона стремиться стать равным тому вектору активности пресинаптических нейронов, который наиболее интенсивно этот нейрон возбуждает:

$$\Delta w_{ij} = c(t)\left(a_j(t) - w_{ij}(t)\right). \qquad (2.58)$$

В соответствии с данным алгоритмом коррекция вектора весов нейрона-победителя осуществляется в направлении, уменьшающем разность между этим вектором и вектором входного образа A_i. В процессе обучения происходит вращение n-мерного вектора весовых коэффициентов нейрона i $W_i = [w_{1i}, \ldots, w_{ni}]$ в направлении входного вектора A_i без существенного изменения его длины.

Существуют различные модификации правила Кохонена:

1. Если нейрон слоя чаще других выигрывает «соревнование», то его значение выхода искусственно уменьшается или нейрон тормозиться, чтобы дать возможность выиграть другим нейронам.

2. Метод окрестности R (рис. 2.9).

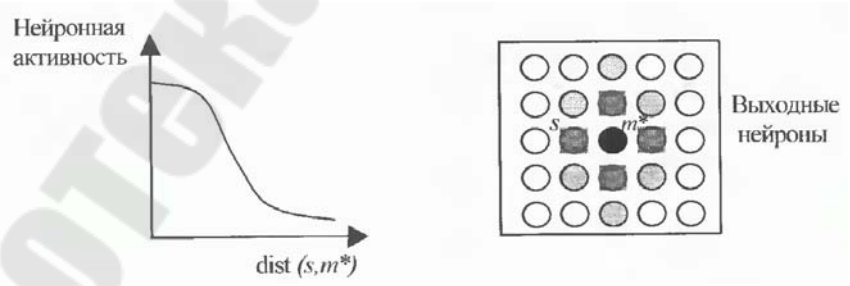


Рис. 2.9. Распределение нейронной активности соседей нейрона-победителя: *dist*(s, m*) – расстояние между выигравшим нейроном m* и его соседом s

После выбора из слоя n нейрона i с минимальным расстоянием D_i обучается не только этот нейрон, но и его соседи, расположенные в окрестности R. Величина R на первых итерациях очень боль-

шая, так что обучаются все нейроны, но с течением времени она уменьшается до нуля. Таким образом, чем ближе конец обучения, тем точнее определяется группа нейронов, отвечающих каждому классу образов [2].

В сети Кохонена используется только положительный стимулирующий эффект, действующий в окрестности выигравшего нейрона. Этим достигается возможность уже на первых шагах обучения выявить группы нейронов, отображающих топологическую карту взаимосвязи входных векторов. С каждой новой итерацией скорость обучения и размер окрестности r уменьшаются, тем самым внутри участков топологической карты выявляются все более мелкие различия. Это в конечном итоге приводит к более точной настройке каждого нейрона.

2.3.3. Проблема «мертвых» нейронов и ее преодоление

При «слепом» (как правило, случайном) выборе начальных значений весов часть нейронов может оказаться в области пространства, в которой отсутствуют обучающие данные, или где их количество ничтожно мало. Такие нейроны имеют очень мало шансов на победу в конкурентной борьбе и адаптацию своих весов, вследствие чего они остаются «мертвыми». В итоге уменьшается количество активных нейронов, участвующих в анализе входных данных, и, следовательно, увеличивается погрешность их интерпретации, называемая погрешностью квантования. Встает проблема активации всех нейронов сети на этапе обучения.

Такую активацию можно осуществить, базируясь на учете количества побед, одержанных каждым нейроном в ходе обучения. Существуют разные механизмы такого учета.

В одном из таких подходов каждому нейрону сети приписывается потенциал p_i, значение которого модифицируется после предъявления каждого обучающего вектора X_k по следующей формуле (в ней w – индекс нейрона-победителя):

$$p_i^{k+1} = p_i^k + 1/M \quad - \text{для} \quad i \neq w; \tag{2.59}$$

$$p_i^{k+1} = p_i^k - p_{\min} \quad - \text{для} \quad i = w, \tag{2.60}$$

где $p_{\min}$ – минимальный потенциал, разрешающий участие в конкурентной борьбе. Максимальное значение потенциала устанавливается равным 1. На практике хорошие результаты получены для $p_{\min} = 0{,}75$.

В другом подходе для выявления победителя в конкурентной борьбе предлагается использовать не фактические значения расстоянии между векторами $d\left(X^k, W_i\right)$, а величины, промасштабированные количеством побед $N_i^w * d\left(X^k, W_i\right)$, где N_i^w – количество побед, одержанных i-м нейроном к текущему моменту.

Как показали эксперименты, при использовании описанных выше механизмов двух-трех циклов обучения обычно достаточно для активации всех нейронов сети, поэтому в последующих циклах эти механизмы отключаются.

2.4. Когнитрон и неокогнитрон Фукушимы

Когнитрон – самоорганизующаяся многослойная нейросеть (рис. 2.11) [15]. Создание Когнитрона (*K. Fukushima*, 1975 г.) явилось плодом синтеза усилий нейрофизиологов и психологов, а также специалистов в области нейрокибернетики, совместно занятых изучением системы восприятия человека. Данная нейронная сеть одновременно является как моделью процессов восприятия на микроуровне, так и вычислительной системой, применяющейся для технических задач распознавания образов [8].

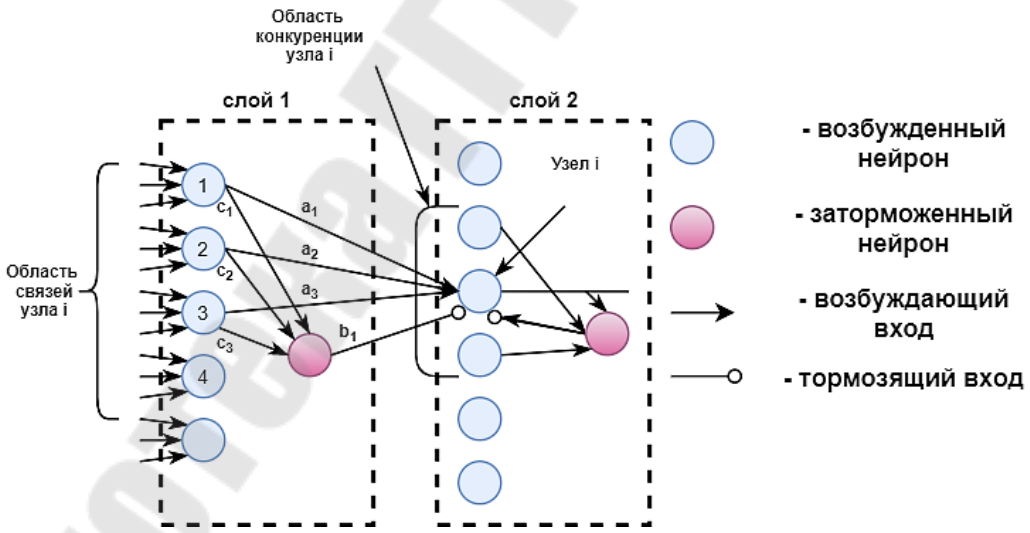


Рис. 2.11. Схема слоев когнитрона

Когнитрон состоит из иерархически связанных слоев нейронов двух типов – тормозящих и возбуждающих. Состояние возбуждения каждого нейрона определяется суммой его тормозящих и возбуждающих входов. Синаптические связи идут от нейронов одного слоя (далее – слоя 1) к следующему (слою 2). Относительно данной синап-

тической связи соотвествующий нейрон слоя 1 является пресинаптическим, а нейрон второго слоя – постсинаптическим. Постсинаптические нейроны связаны не со всеми нейронами слоя 1, а лишь с теми, которые принадлежат их локальной области связей. Области связей близких друг к другу постсинаптических нейронов перекрываются, поэтому активность данного пресинаптического нейрона будет сказываться на все более расширяющейся области постсинаптических нейронов следующих слоев иерархии.

Вход возбуждающего постсинаптического нейрона (на рис. 2.11 – нейрон *i*) определяется отношением суммы ***E*** его возбуждающих входов (a_1, a_2 и a_3) к сумме ***I*** тормозящих входов (b_1 и вход от нейрона *X*):

$$E = \sum_j a_j u_j; \tag{2.61}$$

$$I = \sum_j b_j v_j, \tag{2.62}$$

где *u* – возбуждающие входы с весами; *v* – тормозящие входы с весами. Все веса имеют положительные значения. По значениям *E* и *I* вычисляется суммарное воздействие на *i*-й нейрон: $net_i = \big((1+E)/(1+I)\big) - 1.$ Его выходная активность и затем устанавливается равной net_i, если $net_i > 0$. В противном случае выход устанавливается равным нулю. Анализ формулы для суммарного воздействия показывает, что при малом торможении *I* оно равно разности возбуждающего и тормозящего сигналов. В случае же когда оба эти сигнала велики, воздействие ограничивается отношением. Такие особенности реакции соответствуют реакциям биологических нейронов, способных работать в широком диапазоне воздействий.

Пресинаптические тормозящие нейроны имеют ту же область связей, что и рассматриваемый возбуждающий постсинаптический нейрон *i*. При этом веса таких тормозящих нейронов (c_1, c_2 и c_3) являются заданными и не изменяются при обучении. Их сумма равна единице, таким образом, выход тормозного пресинаптического нейрона равен средней активности возбуждающих пресинаптических нейронов в области связей:

$$v_i = \sum_j c_j u_j. \tag{2.63}$$

Обучение весов возбуждающих нейронов происходит по принципу «победитель забирает все» в области конкуренции – некоторой окрестности данного возбуждающего нейрона. На данном шаге модифицируются только веса a_i нейрона с максимальным возбуждением:

$$\delta a_i = q c_j u_j, \tag{2.64}$$

где c_j – тормозящий вес связи нейрон a_j в первом слое; u_j – состояние его возбуждения; q – коэффициент обучения. Веса тормозящего нейрон a_i второго слоя модифицируются пропорционально отношению суммы возбуждающих входов к сумме тормозящих входов:

$$\delta a_i = (q/2)\sum_j a_j u_j / \sum_j c_j u_j. \tag{2.65}$$

В случае, когда победителя в области конкуренции (на слое 2) нет, как это имеет место, например, в начале обучения, веса подстраиваются по другим формулам:

$$\delta a_i = q' c_j u_j; \tag{2.66}$$

$$\delta a_i = q' \sum_x a_x u_x; \tag{2.67}$$

$$0 < q' < q. \tag{2.68}$$

Данная процедура обучения приводит к дальнейшему росту возбуждающих связей активных нейронов и торможению пассивных. При этом веса каждого из нейронов в слое 2 настраиваются на некоторый образ, часто предъявляемый при обучении. Новое предъявление этого образа вызовет высокий уровень возбуждения соответствующего нейрона, при появлении же других образов его активность будет малой и будет подавлена при латеральном торможении.

Веса нейрона *X*, осуществляющего латеральное торможение в области конкуренции, являются немодифицируемыми, их сумма равна единице. При этом во втором слое выполняются итерации, аналогичные конкурентным итерациям в сети Липпмана–Хемминга.

Перекрывающиеся области конкуренции близких нейронов второго слоя содержат относительно небольшое число других нейронов, поэтому конкретный нейрон-победитель не может осуществить торможение всего второго слоя. Следовательно, в конкурентной борьбе могут выиграть несколько нейронов второго слоя, обеспечивая более полную и надежную переработку информации.

В целом когнитрон представляет собой иерархию слоев, последовательно связанных друг с другом, как было рассмотрено выше для пары слой 1 – слой 2. При этом нейроны слоя образуют не одномерную цепочку, как на рис. 2.11, а покрывают плоскость, аналогично слоистому строению зрительной коры человека. Каждый слой реализует свой уровень обобщения информации. Входные слои чувствительны к отдельным элементарным структурам, например, линиям определенной ориентации или цвета. Последующие слои реагируют уже на более сложные обобщенные образы. В самом верхнем уровне иерархии активные нейроны определяют результат работы сети – узнавание определенного образа. Для каждого в значительной степени нового образа картинка активности выходного слоя будет уникальной. При этом она сохранится и при предъявлении искаженной или зашумленной версии этого образа. Таким образом, обработка информации когнитроном происходит с формированием ассоциаций и обобщений.

Автором когнитрона Фукушимой эта сеть применялась для оптического распознавания символов – арабских цифр. В экспериментах использовалась сеть с четырьмя слоями нейронов, упорядоченными в матрицы 12 × 12 с квадратной областью связей каждого нейрона размером 5 × 5 и областью конкуренции в форме ромба с высотой и шириной 5 нейронов. Параметры обучения равнялись $q = 16;\ \ q' = 2.$ В результате было получено успешное обучение системы на пяти образах цифр (аналогичных картинкам с буквами, которые мы рассматривали для сети Хопфилда), при этом потребовалось около 20 циклов обучения для каждой картинки.

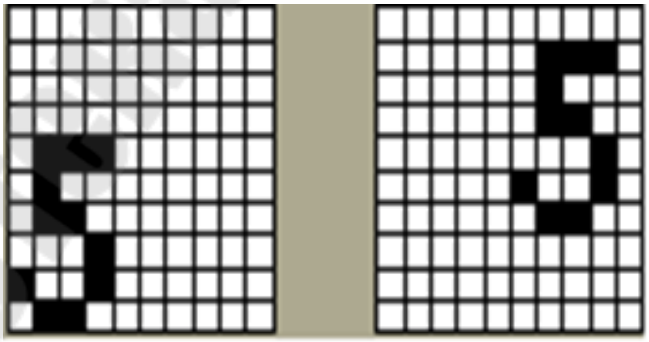


Рис. 2.12. Пример распознаваемого когнитроном изображения

Неокогнитрон (рис. 2.13) представлен иерархией нейронных слоев, каждый из которых состоит из массива плоскостей. Каждый элемент массива складывается из пары плоскостей нейронов. Первая

плоскость состоит из так называемых простых нейроклеток, которые получают сигналы от предыдущего слоя и выделяют определенные образы. Эти образы далее обрабатываются сложными нейронами второй плоскости, задачей которых является сделать выделенные образы менее зависимыми от их положения [14].

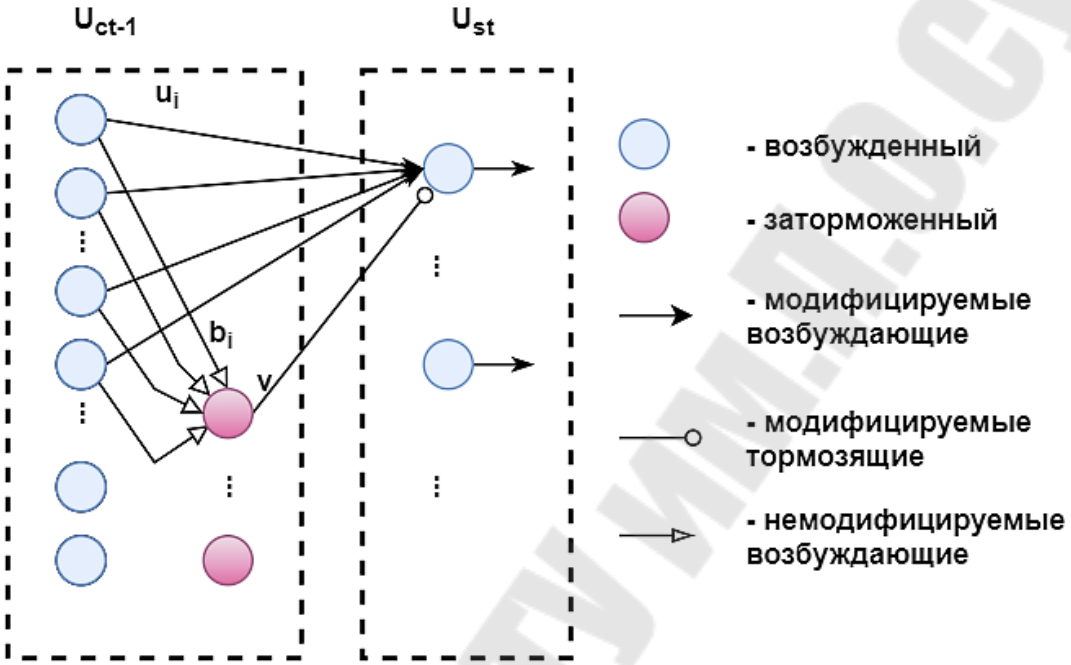


Рис. 2.13. Схема слоев неокогнитрона

Нейроны каждой пары плоскостей обучаются реагировать на определенный образ, представленный в определенной ориентации. Для другого образа или для нового угла поворота образа требуется новая пара плоскостей. Таким образом, при больших объемах информации, неокогнитрон представляет собой огромную структуру с большим числом плоскостей и слоев нейронов.

Простые нейроны чувствительны к небольшой области входного образа, называемой рецептивной областью (или областью связей). Простой нейрон приходит в возбужденное состояние, если в его рецептивной области возникает определенный образ. Рецептивные области простых клеток перекрываются и покрывают все изображение. Сложные нейроны получают сигналы от простых клеток, при этом для возбуждения сложного нейрона достаточно одного сигнала от любого простого нейрона. Тем самым сложная клетка регистрирует определенный образ независимо от того, какой из простых нейронов выполнил детектирование, и, значит, независимо от его расположения.

По мере распространения информации от слоя слою картинка нейронной активности становится все менее чувствительной к ориентации и распололожению образа и в определенных пределах – к его размеру. Нейроны выходного слоя выполняют окончательное инвариантное распознавание.

Обучение неокогнитрона аналогично уже рассмотренному обучению когнитрона. При этом изменяются только синаптические веса простых клеток. Тормозящие нейроны вместо средней активности нейронов в области связей используют квадратный корень из взвешенной суммы квадратов входов:

$$v_i = \sqrt{\sum_j \left(b_j u_j\right)^2}. \qquad (2.69)$$

Такая формула для активности тормозящей клетки менее чувствительна к размеру образа. После выбора простого нейрона, веса которого выбраны для обучения, он рассматривается в качестве представителя слоя, и веса всех остальных нейронов будут обучаться по тем же правилам. Таким образом, все простые клетки обучаются одинаково, выдавая при распознавании одинаковую реакцию на одинаковые образы.

Для уменьшения объема обрабатываемой информации решептивные поля нейронов при переходе со слоя на слой расширяются, а число нейронов уменьшается. В выходном слое на каждой плоскости остается только один нейрон, рецептивное поле которого покрывает все поле образа предыдущего слоя. В целом функционирование неокогнитрона происходит следующим образом. Копии входного изображения поступают на все плоскости простых клеток первого слоя. Далее все плоскости функционируют параллельно, передавая информацию следующему слою. По достижении выходного слоя, в котором каждая плоскость содержит один нейрон, возникает некоторое окончательное распределение активности. На результат распознавания указывает тот нейрон, активность которого оказалась максимальной. При этом существенно разным входным изображениям будут соответствовать разные результаты распознавания.

Неокогнитрон успешно проявил себя при распознавании символов. Нужно отметить, что структура этой сети необычайно сложна, и объем вычислений очень велик, поэтому компьютерные модели неокогнитрона будут слишком дорогими для промышленных приложений.

Особенности обучения когнитрона в сравнении с персептроном. Обучение неокогнитрона происходит без учителя. Оно соответствует

процедуре выделения полного набора факторов. Когда на вход неокогнитрона подаются реальные изображения, нейронам не остается ничего другого, кроме как выделять свойственные этим изображениям компоненты. Так, если подавать на вход рукописные цифры, то малые рецептивные поля простых нейронов первого слоя увидят линии, углы и сопряжения. Размеры зон конкуренции определяют, сколько различных факторов может выделиться в каждой пространственной области. В первую очередь, выделяются наиболее значимые компоненты. Для рукописных цифр это будут линии под различными углами. Если останутся свободные факторы, то далее могут выделиться и более сложные элементы.

От слоя к слою сохраняется общий принцип обучения – выделяются факторы, характерные для множества входных сигналов. Подавая рукописные цифры на первый слой, на определенном уровне мы получим факторы, соответствующие этим числам. Каждая цифра окажется сочетанием устойчивого набора признаков, что выделится как отдельный фактор. Последний слой неокогнитрона содержит столько нейронов, сколько образов предполагается детектировать.

ГЛАВА 3. МНОГОСЛОЙНЫЕ НЕЙРОННЫЕ СЕТИ

3.1. Глубокие сети прямого распространения

3.1.1. Полносвязные нейронные сети прямого распространения

Полносвязанные нейронные сети прямого распространения (*Fully Connected Feed-Forward Neural Networks – FNN*) – это классический тип нейронных сетей где соединения между нейронами не формируют замкнутых циклов, а сигнал по сети передается в одном направлении от входного слоя к выходному через скрытые слои [7]. При этом каждый нейрон предыдущего слоя связан со всеми нейронами последующего слоя.

3.1.2. Многослойный персептрон

Многослойный персептрон (*Multilayer perceptron – MLP*) – первая многослойная полносвязанная нейронная сеть прямого распространения.

Схема многослойного персептрона дана на рис. 3.1.

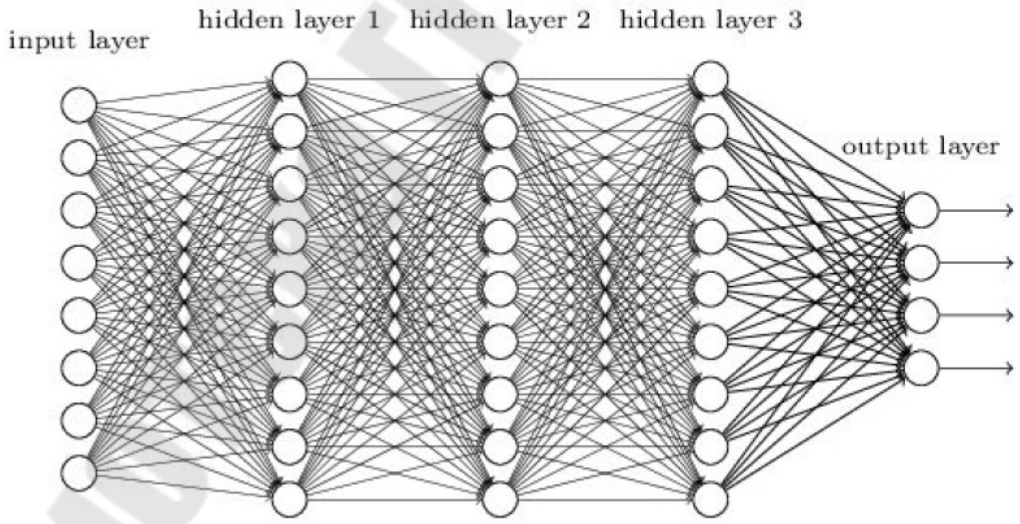


Рис. 3.1. Схема многослойного персептрона

Многослойные персептроны, как и в целом *FNN*, хорошо справляются с задачей классификации [15] и распознавания образов, но имеют ряд существенных недостатков, затрудняющих, или делающих невозможным их практическое использование:

– большое число параметров, например, если взять нейросеть из трех скрытых слоев, которой нужно обрабатывать картинки 100 × 100 ps, это значит, что на входе будет 10000 ps, и они заводятся на три слоя. В общем, если честно посчитать все параметры, у такой сети их будет порядка миллиона. Чтобы обучить нейросеть с миллионом параметров, нужно очень много обучающих примеров, которые не всегда есть. Кроме того, сеть, у которой много параметров, имеет дополнительную склонность переобучаться. Она может обучиться на то, чего в реальности не существует: какой-то шум *Data Set*. Даже если в конце концов сеть запомнит примеры, на тех, которые она не видела, она потом не сможет нормально использоваться;

– при обучении сети обратным распространением, когда ошибка с выходов отправляется на вход, распределяется по всем весам, градиент (производная ошибки) фактически может оказаться незначимым, т. е. сеть нельзя будет обучить.

3.1.3. Автоэнкодер

Автоэнкодер (*AE*) – сеть прямого распространения с «очень маленьким» слоем в середине сети [17].

Схема автоэнкодера приведена на рис. 3.2.

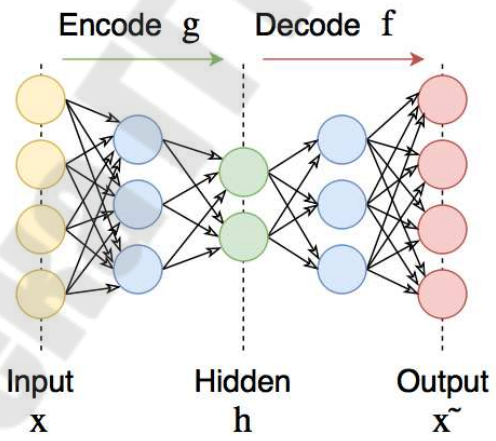


Рис. 3.2. Схема автоэнкодера

Цель этой нейросети – взять какой-то вход, прогнать через себя и на выходе сгенерировать тот же самый вход, т. е. чтобы они совпадали.

Автоэнкодер состоит из двух частей:

1) энкодер – отвечает за сжатие входа в *latent-space*. Представлен функцией кодирования $h = g(x)$;

2) декодер – предназначен для восстановления ввода из *latent-space*. Представлен функцией декодирования $x = f(h)$.

Если получится обучить подобную сеть, то это означает, что слой h может описать входной слой x. Таким образом можно уменьшить размерность исходной задачи. Кроме того, слой h позволяет выявить скрытые признаки исходных данных.

Отсюда следует практическое применение автоэнкодера:

– снижение размерности исходного вектора признаков;

– сглаживание шума в исходных данных за счет выявления значимых (базисных) признаков.

Можно выделить следующие типы автоэнкодеров:

– сжимающий автоэнкодер (*undercomplete*) – в своем простейшем случае представляет нейронную сеть с одним скрытым слоем. Количество нейронов на входном и выходном слое совпадает, на скрытом слое – значительно меньше;

– многослойный автоэнкодер – использует более одного скрытого слоя. Любой из скрытых слоев может быть использован для представления функции кодирования. На практике используются симметричные скрытые слои и в качестве результата функции кодирования – средний слой;

– сверточный автоэнкодер – многослойный автоэнкодер, использующий внутренние сверточные слои;

– шумоподавляющий автоэнкодер (*denoising*) – на вход автоэнкодера подаются зашумленные данные, а выход сравнивается с данными без шума. Таким образом, энкодер будет обучаться извлекать из данных наиболее значимые признаки и отбрасывать шум;

– разреженный автоэнкодер (sparse) – это регуляризованный автокодер, использующий функцию потерь в виде

$$L(x, f(g(x))) + \Omega(h), \tag{3.1}$$

где $h = g(x)$ – код; $\Omega(h)$ – обычный регуляризатор (например L1):

$$\Omega(h) = \lambda |h|. \tag{3.2}$$

Разреженный автоэнкодер не обязательно сужается к центру. Его код может иметь и большую размерность, чем входной сигнал.

Таким образом разреженный автоэнкодер реагирует на уникальные характеристики датасета, на котором он был обучен, и формирует модель, которая в качестве побочного эффекта изучила полезные образы в коде.

3.1.4. Вариационный автоэнкодер

Вариационный автоэнкодер (*variational Autoencoders – VAE*) – это автоэнкодер, который учится отображать объекты в заданное скрытое пространство и обратно. То есть во время обучения обеспечивается упорядоченное распределение кодировок, чтобы гарантировать, что его скрытое пространство является непрерывным, позволяя выполнять случайные преобразования и интерполяцию [16].

Термин «вариационный» происходит от тесной связи, существующей между регуляризацией и методом вариационного вывода в статистике. Поэтому *вариационные автоэнкодеры* относят также к семейству генеративных моделей.

Энкодер выдает не один вектор размера n, а два вектора этого размера: вектор средних значений μ и вектор стандартных отклонений σ. Далее, используя полученные параметры закона распределения, генерируется случайный вектор образца, который подается на декодер для восстановления данных. Таким образом для одинаковых входных данных результат кодирования будет разным вследствие случайности выбора вектора кодирования. Каждому входному объекту соответствует уже не одна точка в скрытом пространстве, а некоторая непрерывная область. Фактически одному набору входных данных соответствует набор векторов кодирования, с которыми уже может работать декодер, что и обеспечивает вариативность модели.

Схема вариационного автоэнкодера дана на рис. 3.3.

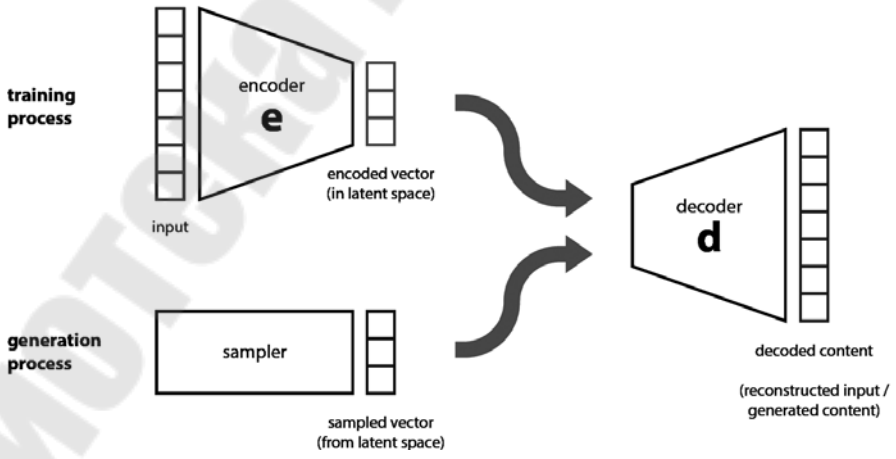


Рис. 3.3. Схема вариационного автоэнкодера

Для различимости образцов необходимо обеспечить генерацию значительно отличающихся векторов μ, а для того чтобы векторы кодирования не сильно отличались для одного образца, нужно миними-

зировать значение вектора σ для каждого образца. При этом все области в скрытом пространстве должны быть как можно ближе друг к другу, обеспечивая непрерывность всего скрытого пространства.

Для достижения этого в функцию потерь вводится так называемая *Kullback–Leibler* (*KL*-расходимость). *KL*-расходимость между двумя функциями распределения показывает, насколько сильно они отличаются друг от друга.

Минимизация *KL*-расходимости означает оптимизацию параметров распределения μ и σ таким образом, что они становятся близки к параметрам целевого распределения.

Геренация примеров вариационным автоэнкодером показана на рис. 3.4.

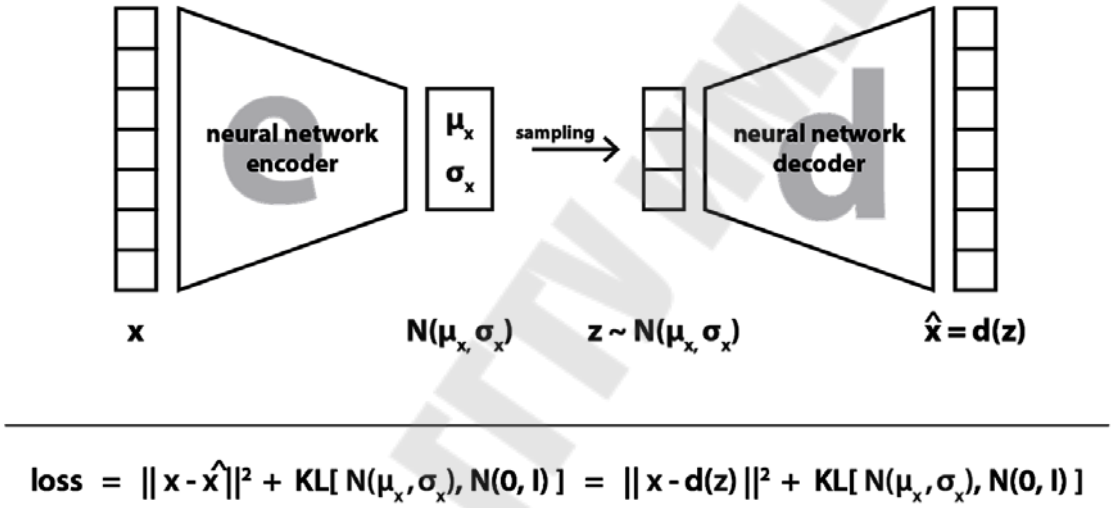


Рис. 3.4. Геренация примеров вариационным автоэнкодером

3.1.5. Глубокие сети доверия

Deep Belief Networks (*DBN*). Глубокая сеть доверия (*Deep Belief Networks – DBN*) – это композиционная архитектура нейронной сети прямого распространения, состоящая из подсетей, представляющих собой RBM и (или) AE.

В *DBN* скрытые слои каждой подсети представляют собой видимый слой следующей подсети. Обучение осуществляется без учителя, начиная с первой пары слоев (на видимый слой которой подается тренировочный набор примеров). После этого фиксируются найденные веса и берется следующий слой, и так далее по всей сети.

Схема *Deep Belief Networks* дана на рис. 3.5.

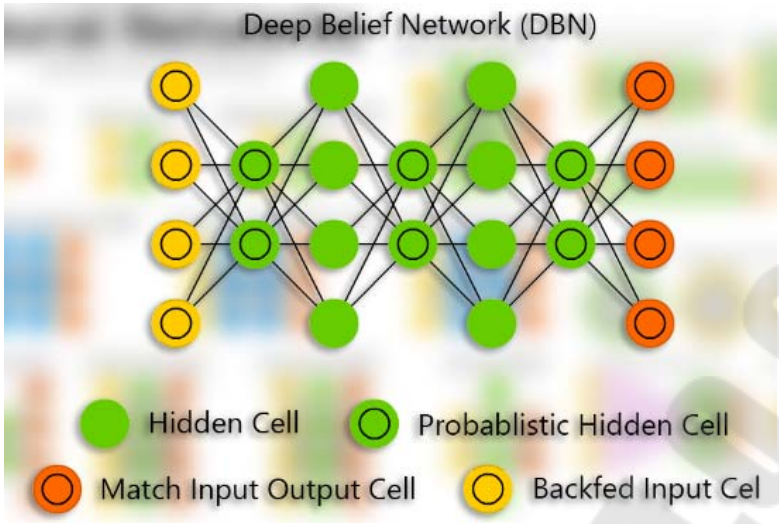


Рис. 3.5. Схема *Deep Belief Networks*

Предобучение. Можно использовать *RBM* для предварительной тонкой настройки весов сети:

• Предположим, у нас есть сеть прямого распространения следующей структуры: $x1 -> x2 -> ... -> xn$, где каждый xi – это количество нейронов в слое; xn – выходной слой.

• Обозначим за $x0$ размерность входного образа, который подается на вход, т. е. на слой $x1$. Так же у нас есть массив данных для обучения, данные (обозначим $D0$) – это пары вида «вход, ожидаемый выход», и мы хотим обучить сеть, используя алгоритм обратного распространения ошибки. Но перед этим осуществим тонкую инициализацию весов, для этого:

– последовательно обучим по одной *RBM* для каждой последовательной пары слоев, включая фиктивный слой $x0$, в итоге получим ровно n штук *RBM*;

– для начала берется первая пара $x0 <-> x1$, она обучается на множестве входных образов из $D0$ (это обучение без учителя). Затем для каждого входного образа из $D0$ делается вывод вероятностей скрытого слоя и сэмплинг, получаем множество бинарных векторов $D1$. Затем с использованием $D1$ обучается пара $x1 <-> x2$, и так далее до $x\{n-1\} <-> xn$;

– разбираем каждую *RBM* и берем вторые слои с их же весами;

– из выбранных вторых слоев каждой RBM составляем исходную сеть $x1 -> x2 -> ... -> xn$, которую и будем называть глубокой сетью доверия.

Схема обучения *DBN* показана на рис. 3.6.

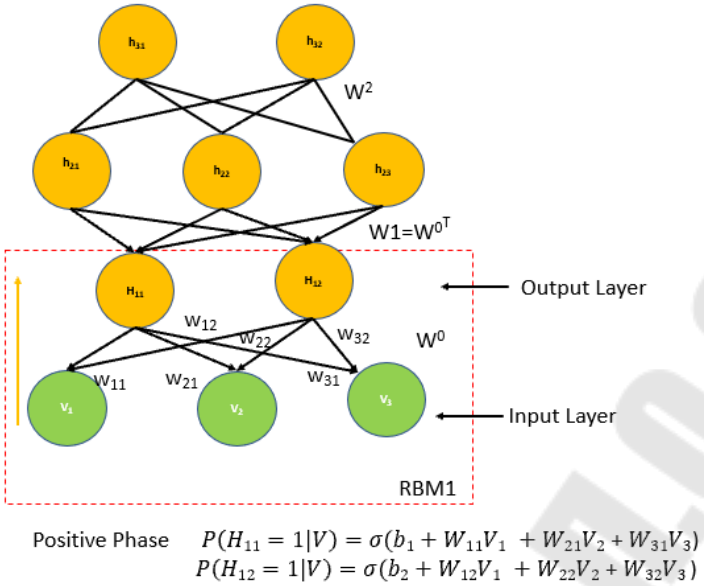


Рис. 3.6. Схема обучения *DBN*

На сегодняшний день не существует формального математического доказательства того, что предварительная тонкая настройка весов гарантирует улучшение качества работы сети, однако на практике это работает.

Схема предварительной тонкой настройки весов представлена на рис. 3.7.

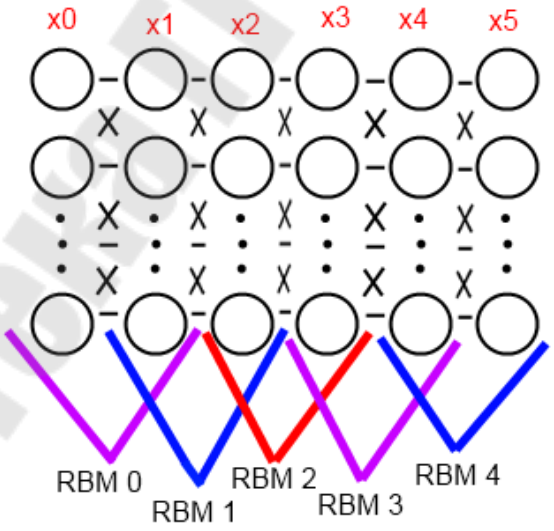


Рис. 3.7. Схема предварительной тонкой настройки весов

Объяснение этому эффекту можно дать следующее: при обучении самой первой *RBM* мы создаем модель, которая по видимым состояниям генерирует некоторые скрытые признаки, т. е. сразу помеща-

ем веса в некоторый минимум, необходимый для вычисления этих признаков; с каждым последующим слоем вычисляются признаки признаков, а веса всегда помещаются в состояние, достаточное для вычисления этих иерархических признаков; когда дело доходит до обучения с учителем, по сути, эффективно обучаться будут только 2–3 слоя от выхода, на основании тех гиперпризнаков, что были вычислены раньше, а те, в свою очередь, будут незначительно меняться в угоду задаче. Отсюда можно предположить, что необязательно таким образом инициализировать все слои, а выходной и, возможно, 1–2 скрытых можно инициализировать обычно, например, выбирая числа из нормального распределения с центром в 0 и дисперсией 0,01.

3.1.6. Сверточные нейронные сети

Сверточная нейронная сеть (*convolutional neural network – CNN*) – это архитектура нейронной сети прямого распространения, нацеленная на эффективное распознавание образов. Использует некоторые особенности зрительной коры, в которой были открыты так называемые простые клетки, реагирующие на прямые линии под разными углами, и сложные клетки, реакция которых связана с активацией определенного набора простых клеток. Таким образом, идея сверточных нейронных сетей заключается в чередовании сверточных слоев (англ. *convolution layers*) и субдискретизирующих слоев (англ. *subsampling layers* или англ. *pooling layers* – слоев подвыборки).

Структура сверточной нейронной сети дана на рис. 3.8.

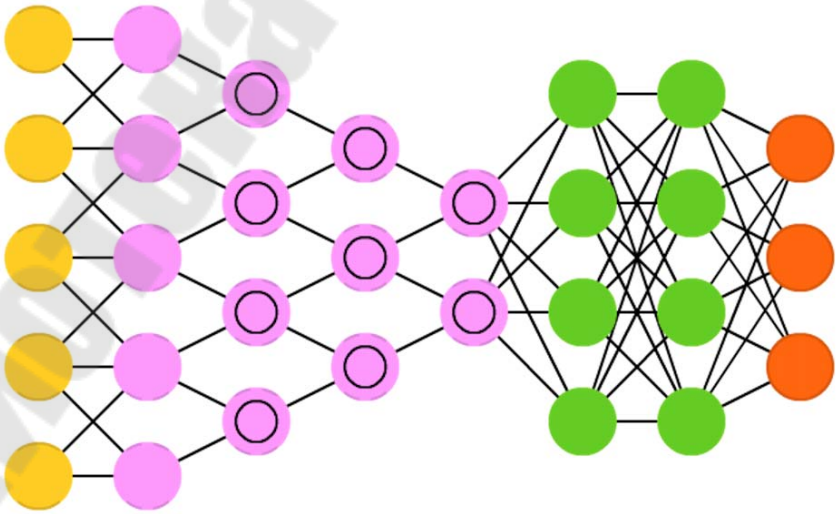


Рис. 3.8. Структура сверточной нейронной сети

Структура сети – однонаправленная (без обратных связей), принципиально многослойная. Для обучения используются стандартные методы, чаще всего метод обратного распространения ошибки. Функция активации нейронов (передаточная функция) – любая, по выбору исследователя.

Свое название архитектура сети получила из-за наличия операции свертки, суть которой в том, что каждый фрагмент изображения умножается на матрицу (ядро) свертки поэлементно, а результат суммируется и записывается в аналогичную позицию выходного изображения.

3.2. Многослойные нейронные сети: обучение и оценка

3.2.1. Функция потери (loss function)

Функция потерь используется для расчета ошибки между реальными и полученными ответами нейронной сети.

Процесс обучения нейронной сети – это процесс минимизации ошибки, т. е. поиск минимума функции потери.

Функция потерь измеряет, насколько «хороша» нейронная сеть в отношении данной обучающей выборки и ожидаемых ответов. Она также может зависеть от таких переменных, как веса и смещения.

Функция потерь одномерна и не является вектором, поскольку она оценивает, насколько хорошо нейронная сеть работает в целом:

$$L = \frac{1}{N}\sum_{i}\left(f\left(x_i, W\right), y_i\right). \tag{3.3}$$

Некоторые известные функции потерь:

– квадратичная (среднеквадратичное отклонение):

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \widehat{Y_i}\right)^2;$$

– кросс-энтропия:

$$\mathrm{H}(p,q) = -\sum_{x} p(x)\log q(x);$$

– экспоненциальная (*AdaBoost*):

$$L_{exp}(x, y) = e^{-yf(x)};$$

– расстояние Кульбака–Лейблера:

$$D_{\mathrm{KL}}(P \mid Q) = \int_X p \log \frac{p}{q} \mathrm{d}\mu.$$

При обучении нейронной сети одной из проблем является переобучение. Для борьбы с переобучением используют в функции потерь специальные «штрафы», заставляющие определенным образом выбирать веса с целью получения наиболее «простого» решения. Например, для задач классификации нужно выбирать наиболее простое разделение данных, т. е. прямую линию, а не кривую.

Функцию потерь строят состоящей из двух компонент: потерь на обучающих данных (суммы всех $\boldsymbol{L_i}$) и потерь регуляризации $\boldsymbol{R}(\boldsymbol{W})$:

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i\left(f(x_i, W), y_i\right) + \lambda R(W). \tag{3.4}$$

Регуляризация *L*1. *L*1 использует манхэттенскую норму $\boldsymbol{W}$ и тоже добавляет штраф к весовым коэффициентам. С регуляризацией *L*1 матрица параметров будет более *разреженной*:

$$R(W) = \sum_k \sum_l |W_{k,l}|. \tag{3.5}$$

Регуляризация *L*2. Применяется в машинном обучении довольно часто. Это обычная евклидова норма параметров $\boldsymbol{W}$ (иногда берется квадрат или половина квадрата нормы). Идея в том, что вы просто добавляете «штраф» к весам, чтобы их значения не оказались слишком большими. Функция $\boldsymbol{R}(\boldsymbol{W})$ выглядит следующим образом:

$$R(W) = \sum_k \sum_l W_{k,l}^2. \tag{3.6}$$

Эластичная сеть. Это комбинация регуляризаций *L*1 и *L*2:

$$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|. \tag{3.7}$$

3.2.2. Эволюционное обучение

В рамках этого метода оперируем следующей терминологией: коэффициенты матрицы весов – геном; один коэффициент – ген; «перевернутая вниз головой» функция потерь – ландшафт приспособленности (тут мы уже ищем локальный максимум, но это всего лишь условность).

Алгоритм эволюционного обучения включает следующие шаги:

1. Проинициализировать геном (матрицу весов) случайным образом в диапазоне от –1 до 1. Повторить это несколько раз, тем самым создав начальную популяцию разных, но случайных нейросетей. Размер популяции обозначим через *P – population or parents*.

2. Создать нескольких потомков, например, три-четыре клона каждого родителя, внеся небольшие изменения (мутации) в их геном. Пример: переназначить случайным образом половину весов, или добавить случайным образом к половине весов случайные значения в диапазоне от –0,1 до 0,1.

3. Оценить приспособленность каждого потомка на основе того, как он справляется с примерами из обучающей выборки (в самом простом варианте – процент верно угаданных классов, в идеале – перевернутая функция потерь). Отсортировать потомков по их приспособленности.

4. «Оставить в живых» только *P* самых приспособленных и вернуться к п. 2, повторяя этот цикл несколько раз до достижения заданной точности.

Такую нейроэволюцию можно улучшить. Например, можно ввести такие дополнительные гены-параметры, как τ – темп мутагенеза и μ – сила мутагенеза.

Теперь аддитивные мутации в матрицу весов нейронов будут вносится с вероятностью τ, добавляя каждому параметру случайное число в выбранном диапазоне (например, от –0,1 до 0,1), умноженное на μ. Эти гены тоже будут подвержены изменчивости.

В идеале отбор должен контролировать силу и темп мутагенеза на разных этапах эволюции, увеличивая эти параметры до тех пор, пока скачком возможно выбраться из локального максимума ландшафта приспособленности, либо уменьшая их, чтобы медленно и без резких скачков двигаться к глобальному максимуму. Также можно добавить в модель кроссинговер. Теперь потомки будут образовываться путем скрещивания, получив случайным образом по половине генов от двух случайных родителей. В этой схеме тоже необходимо оставить внесение случайных мутаций в геном.

3.2.3. Гиперпараметрическая оптимизация

Оптимизация гиперпараметров – задача машинного обучения по выбору набора оптимальных гиперпараметров для обучающего алгоритма.

Одни и те же виды моделей машинного обучения могут требовать различные предположения, веса или скорости обучения для различных видов данных. Эти параметры называются гиперпараметрами и их следует настраивать так, чтобы модель могла оптимально решить задачу обучения. Для этого находится кортеж гиперпараметров, который дает оптимальную модель, оптимизирующую заданную функцию потерь на заданных независимых данных. Целевая функция берет кортеж гиперпараметров и возвращает связанные с ними потери. Часто используется перекрестная проверка для оценки этой обобщающей способности.

В нейросетевых моделях можно выделить следующие гиперпараметры:

• *Скорость обучения* (*learning rate*) – это гиперпараметр, определяющий порядок того, как мы будем корректировать наши весы с учетом функции потерь в градиентном спуске. Чем ниже величина, тем медленнее мы движемся по наклонной. Хотя при использовании низкого коэффициента скорости обучения мы можем получить положительный эффект в том смысле, чтобы не пропустить ни одного локального минимума, – это также может означать, что нам придется затратить много времени на сходимость, особенно если мы попали в область плато.

Эффективность скорости обучения может быть оценена путем тренировки модели с изначально заданной низкой скоростью обучения, которая затем повышается (линейно или по экспоненте) в каждой итерации. Фиксируя значения показателей на каждой итерации, мы увидим, что по мере повышения скорости обучения будет (достигнута) точка, в которой значения функции потерь перестает уменьшаться и начинает увеличиваться. На практике наша скорость обучения в идеальном варианте должна быть где-то слева от нижней точки.

• *Параметр регуляризации* – влияет на качество процесса обучения и позволяет бороться с переобучением.

Количество скрытых слоев и нейронов – определяет размер сети и качество ее работы. Однако на начальных этапах определения гиперпараметров нейронной сети целесообразно уменьшить размеры скрытых слоев сети и выходного, что позволит использовать меньшую обучающую выборку и ускорить процесс подбора начальных гиперпараметров.

• *Количество эпох обучения* – в конце каждой эпохи обучения выполняется подсчет точности классификации на тестовых данных. Когда точность перестает улучшаться, обучение прекращается. Одна-

ко такой подход не всегда оправдан, на практике гораздо целесообразней использовать отложенную остановку, т. е. прекращать обучение, если наилучшая точность классификации не улучшается какое-то заданное количество эпох.

Размер выборки для обучения (размер *minibatch*) – выбор лучшего размера мини-пакета – это компромисс, максимизирующий скорость обучения. Слишком маленький пакет приводит к значительным погрешностям при вычислении общего градиента функции потери и не позволит в полной мере использовать возможности тензорных вычислений, что, в целом, снизит время обучения. Слишком большой пакет не даст частого обновления весов и, как следствие, так же снизит скорость обучение. Выбор размера мини-пакета, при котором скорость максимизируется, относительно независима от других гиперпараметров (кроме общей архитектуры), поэтому, чтобы найти хороший размер мини-пакета, не обязательно их оптимизировать. Следовательно, достаточно будет использовать приемлемые (не обязательно оптимальные) значения у других гиперпараметров, а потом попробовать несколько разных размеров мини-пакетов, масштабируя коэффициент скорости обучения.

Автоматизация подбора гиперпараметров реализуется посредством алгоритмов «перебора по сетке». При этом лучше использовать не равномерный поиск, а случайный (стохастический) – он чаще всего дает гораздо более удачные результаты.

Ранняя остановка (*early stopping*). Валидационное множество может служить базой для оценки гиперпараметров сети (таких, как глубина, количество нейронов/ядер, параметры регуляризации и т. д.). Представьте, что сеть прогоняется с различными комбинациями гиперпараметров, а затем решение принимается на основе их производительности на валидационном множестве. Обратите внимание, что мы не должны знать ничего о тестовом множестве до того, как окончательно определимся с гиперпараметрами, так как иначе признаки тестового множества непроизвольно вольются в процесс обучения. Этот принцип также известен как *золотое правило машинного обучения*, и нарушалось во многих ранних подходах.

Возможно, самый простой способ использования валидационного множества – настройка количества «эпох» (циклов) с помощью процедуры, известной как *ранняя остановка* – просто остановите процесс обучения, если за заданное количество эпох потери не начинают уменьшаться.

3.2.4. Методы инициализации сети

Нулевая инициализация. В общей практике смещения инициализируются 0, а веса – случайными числами. Если все веса инициализированы 0, производная по функции потерь одинакова для каждого веса. Таким образом, все веса имеют одинаковое значение на последующих итерациях. Это делает скрытые единицы симметричными и продолжается для всех n итераций, то есть установка весов в 0 не делает его лучше, чем линейная модель. Важно помнить, что смещения не имеют никакого эффекта, когда они инициализируются с 0.

Случайная инициализация. Присвоение случайных значений весам лучше, чем просто присвоение 0. Однако имеется ряд замечаний по значениям весов:

– если веса инициализируются с очень высокими значениями, то взвешенная сумма входов на нейроне становится большой, а значение функции активации близко к единице. Дисперсия входного сигнала будет иметь тенденцию быстро расти с каждым слоем. В итоге она станет большой и бесполезной, так как сигмоидальная функция полога на больших значениях. Это значит, что наша активация нейрона будет стремиться к насыщению и градиент будет приближаться к нулю;

– если веса инициализируются малыми значениями, то взвешенная сумма входов близка к нулю, дисперсия входного сигнала начнет уменьшаться во время прохождения каждого слоя в сети. В итоге входной сигнал упадет до действительного малого значения и будет неэффективным.

Эта проблема часто упоминается как исчезающий градиент.

Инициализация Xavier. Один из хороших способов назначить веса – это нормальное распределение. Очевидно, что это распределение будет иметь нулевое математическое ожидание и некоторую конечную дисперсию. На каждом слое мы хотим, чтобы дисперсия оставалась одинаковой. Это позволяет нам держать сигнал от роста на большие значения или исчезновения до нуля. Другими словами, нам надо инициализировать веса так, чтобы дисперсия оставалась неизменной.

Тогда можно брать веса из нормального распределения с нулевым математическим ожиданием и дисперсией, равной $1/N$, где N – число входных нейронов.

В оригинале авторы берут среднее между числом входных нейронов и выходных. Причина этого действия заключается в сохранении сигнала на этапе обратного распространения. Но это более сложно реализовать в вычислительном смысле. Поэтому во время практической реализации берется только число входных нейронов.

Инициализация *He*. Логика построения аналогична логике вывода инициализации Xavier для случая функции активации *ReLU*, компенсирующая тот факт, что эта функция возвращает ноль для половины области определения. Дисперсия в этом случае так же будет равна $1/N$, где N – число входных нейронов.

Ортогональная инициализация. Ортогональная инициализация (*orthogonal initialization*) – используется для инициализации весов сетей *RNN*. Начальная матрица весов представляется ортогональной матрицей.

Ортогональные матрицы обладают многими интересными свойствами, но наиболее важным для нас является то, что все собственные значения ортогональной матрицы имеют абсолютное значение 1. Это означает, что независимо от того, сколько раз мы выполняем повторное умножение матрицы, результирующая матрица не вырождается.

Когда мы выполняем повторное умножение матрицы в пределах *RNN*, и результат вырождается или наблюдается его значительный рост, то и градиенты либо сильно затухают, либо наблюдается их взрывной рост.

Если градиенты исчезают, обучение почти останавливается, так как никакая информация не передается обратно. Если градиенты начинают сильно возрастать, то обучение может никогда не сойтись, так как обновление градиента сильно колеблется.

При инициализации весовой матрицы в *RNNs* нередко используется случайная равномерная или случайная нормальная инициализация. Этот метод не дает никаких гарантий относительно собственных значений матрицы весов и, вероятно, приведет к взрывному росту или исчезновению результатов.

Если вместо этого мы используем ортогональную матрицу для инициализации весов, результирующая матрица не будет экстремально возрастать и не выродится. Это позволяет более эффективно вычислять градиенты при обратном распространении.

Одним из подходов построения ортогональной матрицы является использование разложения по сингулярным значениям.

3.2.5. Батч-нормализация

Батч-нормализация – это метод ускорения глубокого обучения, предложенный *Ioffe* и *Szegedy* в начале 2015 г. Метод решает следующую проблему, препятствующую эффективному обучению нейронных сетей: по мере распространения сигнала по сети, даже если мы

нормализовали его на входе, пройдя через внутренние слои, он может сильно исказиться как по матожиданию, так и по дисперсии (данное явление называется внутренним ковариационным сдвигом), что чревато серьезными несоответствиями между градиентами на различных уровнях. Поэтому нам приходится использовать более сильные регуляризаторы, замедляя тем самым темп обучения.

Батч-нормализация предлагает весьма простое решение данной проблемы: нормализовать входные данные таким образом, чтобы получить нулевое матожидание и единичную дисперсию.

Нормализация выполняется перед входом в каждый слой. Это значит, что во время обучения мы нормализуем *batch_size* примеров, а во время тестирования мы нормализуем статистику, полученную на основе всего обучающего множества, так как увидеть заранее тестовые данные мы не можем. А именно – мы вычисляем матожидание и дисперсию для определенного батча (пакета) $\beta = x_1, ..., x_m$ следующим образом:

$$\mu_\beta = \frac{1}{m}\sum_{i=1}^{m} x_i; \tag{3.8}$$

$$\sigma^2{}_\beta = \frac{1}{m}\sum_{i=1}^{m}\left(x_i - \mu_\beta\right)^2. \tag{3.9}$$

С помощью этих статистических характеристик мы преобразуем функцию активации таким образом, чтобы она имела нулевое матожидание и единичную дисперсию на всем батче:

$$\hat{x}_i = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta{}^2 - \in}}, \tag{3.10}$$

где $\varepsilon > 0$ – параметр, защищающий нас от деления на 0 (в случае, если среднеквадратичное отклонение батча очень мало или даже равно нулю).

Наконец, чтобы получить окончательную функцию активации *y*, надо убедиться, что во время нормализации не потеряна способность к обобщению, и так как к исходным данным применялись операции масштабирования и сдвига, можно позволить произвольные масштабирование и сдвиг нормализованных значений, получив окончательную функцию активации:

$$y_i = \gamma \hat{x}_i + \beta, \tag{3.11}$$

где β и γ – параметры батч-нормализации, которым системы можно обучить (их можно оптимизировать методом градиентного спуска на обучающих данных). Это обобщение также означает, что батч-нормализацию может быть полезно применять непосредственно к входным данным нейронной сети.

Батч-нормализация может служить отличным *регуляризатором*, позволяя не так осмотрительно выбирать темп обучения. Регуляризация здесь – это следствие того факта, что результат работы сети для определенного примера больше *не детерминирован* (он зависит от всего батча, в рамках которого данный результат получен), и это упрощает обобщение.

И хотя авторы метода рекомендуют применять батч-нормализацию до функции активации нейрона, последние исследования показывают, что, если не полезнее, то, по крайней мере, также выгодно использовать ее после активации.

3.2.6. Расширение обучающего множества (*data augmentation*)

В идеале хорошо бы уметь так обучать сеть, чтобы она оставалась устойчивой к трансформационным искажениям исходных данных, но наша модель может обучаться только на основе тех образцов, которые мы ей предоставили, при том что она проводит в некотором роде статистический анализ обучающего множества и экстраполирует его.

К счастью, для этой проблемы существует решение – простое, но эффективное, особенно на задачах по распознаванию изображений: искусственно расширить обучающие данные искаженными версиями во время обучения. Это означает следующее: перед тем, как подать пример на вход модели, мы применим к нему все трансформации, которые сочтем нужными, а потом позволим сети напрямую наблюдать, какой эффект имеет применение их к данным и, обучая ее «хорошо вести себя» и на этих примерах.

3.2.7. Ансамбли (*ensembles*)

Пример архитектуры ансамбля приведен на рис. 3.9.

Одна интересная особенность нейронных сетей, которую можно заметить, когда они используются для распределения данных на более

чем два класса, – это то, что при различных начальных условиях обучения им легче дается распределение по одним классам, в то время как другие приводят их в замешательство.

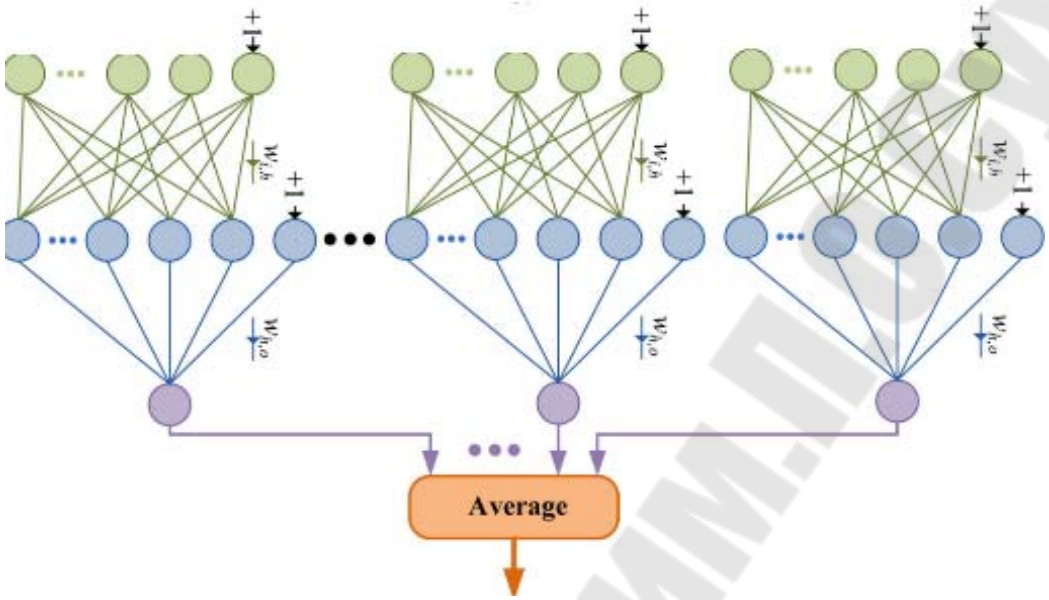


Рис. 3.9. Пример архитектуры ансамбля

С этим несоответствием можно бороться с помощью метода статистических ***ансамблей*** – вместо *одной* сети постройте несколько ее *копий* с разными начальными значениями и вычислите их средний результат на одних и тех же входных данных.

ГЛАВА 4. ПРОГРАММНЫЕ СРЕДСТВА ГЛУБОКОГО ОБУЧЕНИЯ

4.1. Фреймворки глубокого обучения

4.1.1. Понятие графа вычислений

Под глубоким обучением, как правило, понимают обучение функции, представляющей собой композицию множества нелинейных преобразований. Такая сложная функция еще называется потоком или графом вычислений.

Граф вычислений – это граф, узлами которого являются функции (обычно достаточно простые, взятые из заранее фиксированного набора), а ребра связывают функции со своими аргументами:

$$f = Wx; \quad L_i = \sum_{j \neq y_i} \max\left(0, s_j - s_{y_i} + 1\right). \tag{4.1}$$

На рис. 4.1 изображен пример графа с классификатором. Узел с операцией (*) означает умножение матриц параметров W и данных x, результатом которого является вектор весов s. Следующая вершина зависимых потерь (*hinge loss*) определяет потери данных L. Узел R вычисляет регуляризацию. В конце формируются общие потери путем суммирования регуляризации и потерь данных.

Преимущество графов в том, что они позволяют использовать метод обратного распространения ошибки (*backpropagation*). Этот алгоритм рекурсивно использует правило дифференцирования сложной функции для вычисления градиента каждой переменной в графе. Метод полезен для сложных функций, которые применяются в сверточных нейросетях.

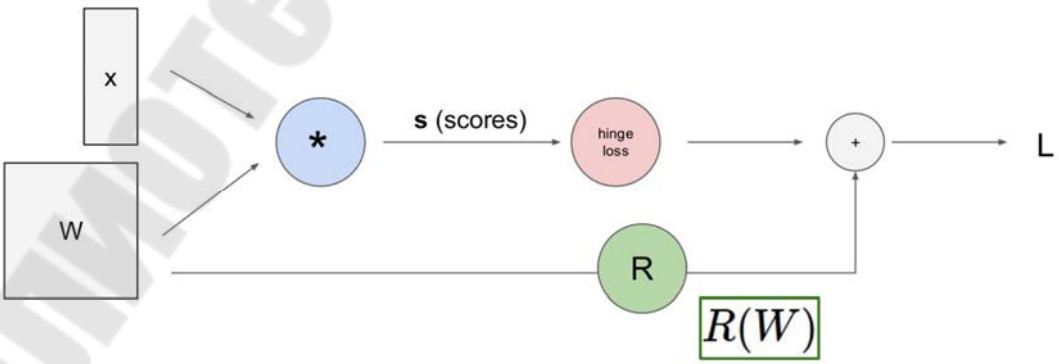


Рис. 4.1. Пример графа вычислений с классификатором

Приведем пример применения вычислительного графа: пусть имеется функция $f(x,y,z)=(x+y)z$ и необходимо найти ее градиенты по отношению к каждой из переменных. На вычислительном графе отражены операции $x+y$ и $(x+y)z$ в виде вершин сложения (+) и умножения (*). Для примера взяты значения $x=-2;\ y=5;\ \ z=-4,$ поэтому −2 + 5 = 3 (промежуточный результат после узла сложения), а $3*(-4)=-12$ (результат умножения).

$$f(x,y,z)=(x+y)z. \tag{4.2}$$

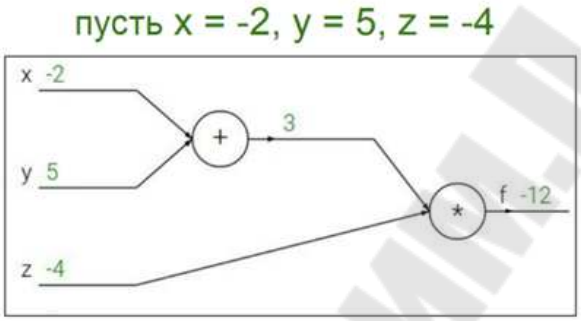


Рис. 4.2. Промежуточные этапы работы графа

Следует ввести обозначения для промежуточных значений. Пусть переменная после (+) называется q $(q=x+y),$ тогда функция f будет равна qz. Далее необходимо выписать градиенты (частные производные): градиент q зависит от переменных x и y, а градиент f – от q и z. Поскольку необходимо найти градиент исходной функции f, которая зависит от всех трех переменных x, y и z, то требуется вычислить $\frac{\partial q}{\partial x}$, $\frac{\partial q}{\partial y}$ и $\frac{\partial f}{\partial z}$.

$$q=x+y;\ \ \frac{\partial q}{\partial x}=1;\ \frac{\partial q}{\partial y}=1; \tag{4.3}$$

$$f=qz;\qquad \frac{\partial f}{\partial q}=z;\ \frac{\partial f}{\partial z}=q. \tag{4.4}$$

Частные производные $\frac{\partial q}{\partial x}$ и $\frac{\partial q}{\partial y}$ равны единице – это означает, что при любом изменении x или **у** q изменится точно так же.

Необходимо рекурсивно воспользоваться правилом дифференцирования сложной функции. Следует начать с конца вычислительного графа и двигаться к началу, по пути считая все градиенты. Самое последнее полученное значение – функция f и ее промежуточный результат –12, частная производная по которому равна единице. Далее идет переменная z. Известно, что $\frac{\partial f}{\partial z} = q = 3$. Затем – вершина (+) и переменная q: производная $\frac{\partial f}{\partial q}$ равна z, т. е. −4.

В итоге в двух ребрах, входящих в вершину умножения, градиент каждого из входов равен значению другого входа, умноженному на предыдущий градиент.

Этап процесса вычислений в графе представлен на рис. 4.3.

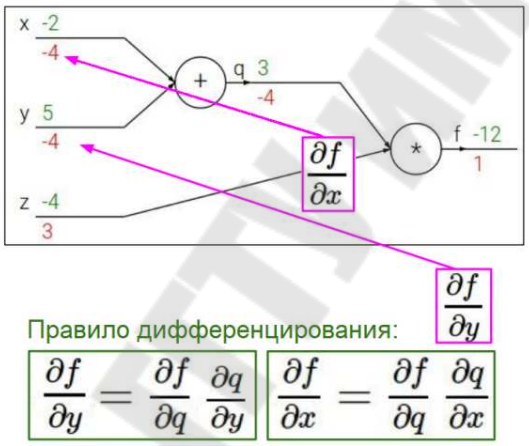


Рис. 4.3. Этап процесса вычислений в графе

Следующим этапом является производная $\frac{\partial f}{\partial y}$. По правилам дифференцирования сложной она может быть записана как $\frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y}$. Значения $\frac{\partial f}{\partial q}$ и $\frac{\partial q}{\partial y}$ известны: они равны −4 и 1 соответственно, поэтому производная dfdy равна −4. Аналогично – для $\frac{\partial f}{\partial x}$.

На данном шаге проявляется закономерность: в вершине сложения равны все три градиента – и самой вершины, и входящие в нее. Значение производной для узла, который следует за текущим, называется восходящий градиент, а для предыдущего узла – локальный градиент.

Метод обратного распространения ошибки может сильно упрощать вычисление градиентов для громоздких функций.

Однако вычислительный граф может быть оптимизирован. В предыдущем примере граф разбивался на множество простейших вершин. Но на самом деле это не всегда необходимо: вершины можно группировать друг с другом, если они образуют легко дифференцируемую функцию. Это может существенно сократить и упростить вычисления.

В качестве примера может быть рассмотрена сигмоида: ее производная легко выражается через саму функцию:

$$a(w,x)=\frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}; \tag{4.5}$$

$$\sigma(x)=\frac{1}{1+e^{-x}}; \tag{4.6}$$

$$\frac{d\sigma(x))}{dx}=\frac{e^{-x}}{\left(1+e^{-x}\right)^2}=\left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right)=(1-\sigma(x))\sigma(x). \tag{4.7}$$

Пример оптимизации графа с сигмоидой приведен на рис. 4.4.

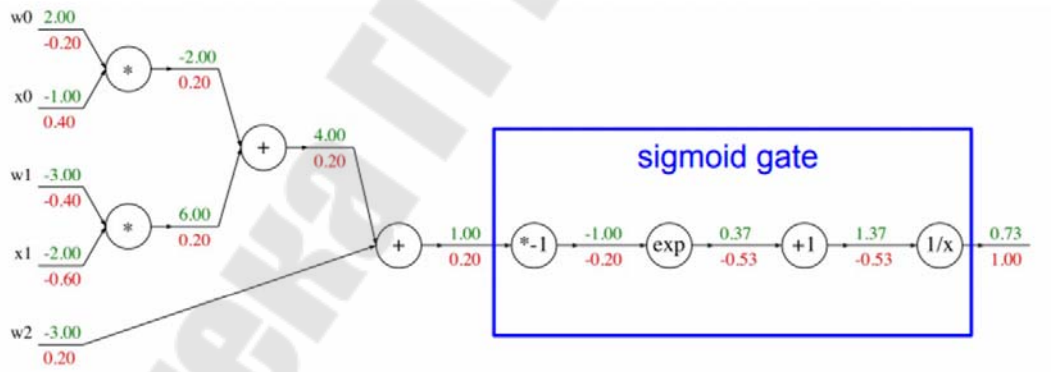


Рис. 4.4. Пример оптимизации графа с сигмоидой

Переменные степени экспоненты $w0$, $w1$, $w2$, $x0$ и $x1$ складываются в вершине сложения, которая следует сразу за $w2$. Поэтому та часть графа, которая полностью совпадает с сигмоидой, может быть объеденина в один узел. Чтобы убедиться, что локальные градиенты в этом случае окажутся одинаковыми, достаточно подставить значение функции f в выражение производной сигмоиды (рис. 4.5).

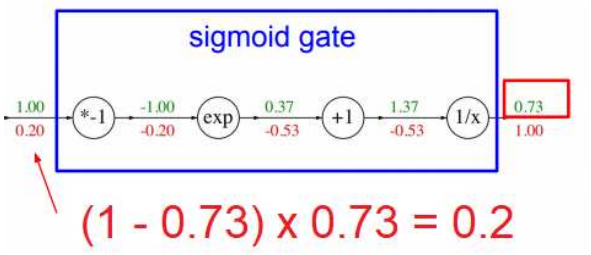


Рис. 4.5. Нахождение сигмоиды

Градиенты совпадают. Итоговый граф (результат упрощения графа с сигмоидой) представлен на рис. 4.6.

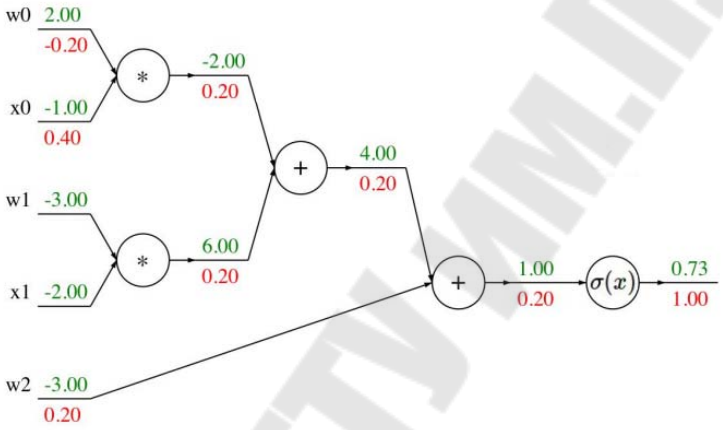


Рис. 4.6. Результат упрощения графа с сигмоидой

Все предыдущие действия над вычислительными графами делятся на два больших этапа:

1. Проход по графу в прямом направлении, подставляя исходные значения переменных и считая все промежуточные результаты.

2. Проход в обратном направлении, вычисляя градиенты и пользуясь правилом дифференцирования.

В коде они могут быть описаны функциями *forward* и *backward*, которые будут выглядеть следующим образом:

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [передаём входные данные...]
        # 2. следуем по графу в прямом направлении:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # последняя вершина графа выводит потери
    def backward():
```

```
            for gate in re-
versed(self.graph.nodes_topologically_sorted()):
                gate.backward() # немного обратного распространения
с правилом дифференцирования
            return inputs_gradients
```

При этом можно установить механику *forward* и *backward* для каждой вершины.

По типу графа вычислений все фреймворки глубокого обучения сетей можно разделить на три крупные категории:

1. Фиксированные модули. В данных фреймворках заранее определенные блоки комбинируются в граф вычислений и затем осуществлятся его запуск. Прямой и обратный проходы заложены в каждом таком блоке. Определение новых блоков гораздо сложнее использования готовых и требует дополнительных знаний и навыков. Расширяемость близка к нулю, однако при использовании готовых блоков благодаря высокой оптимизированности заранее написанного кода демонстрируется высокая скорость работы. Представители данной группы: *Caffe*, *Caffe*2, *CNTK*, *Kaldi*, *DL*4*J*.

2. Статический граф вычислений. Эти фреймворки являются более гибкими: на этапе описания возможно создать граф вычислений произвольного размера и сложности, однако после компиляции его изменения невозможны. Доступными останутся два действия: запуск графа в прямом или обратном направлениях. Данный подход комбинирует гибкость на этапе разработки и скорость в момент исполнения. Представители: *Theano*, *TensorFlow*, *MXNet*.

3. Динамический граф вычислений. Схож со статическим графом по принципу формирования, однако может перестраивать статический граф перед каждым его запуском. Граф динамически строится каждый раз при каждом прямом проходе для дальнейшей возможности осуществления обратного прохода. Подобный подход дает максимальную гибкость и расширяемость. К этому классу фреймворков относятся *Torch* и *PyTorch*.

Схематичное изображение трех основных категорий фреймворков глубокого обучения показано на рис. 4.7.

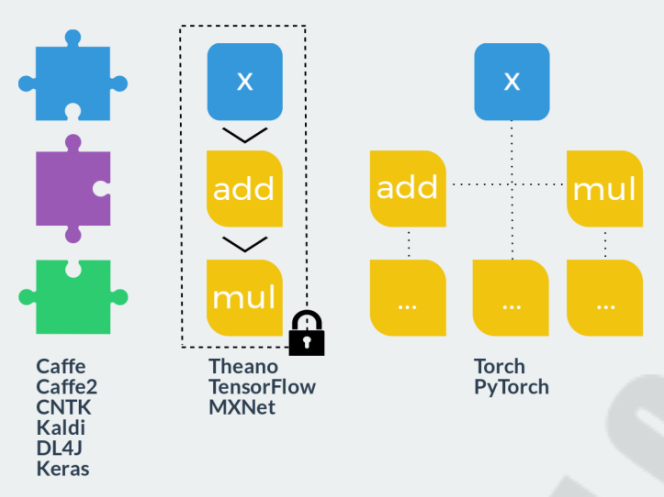


Рис. 4.7. Основные категории фреймворков глубокого обучения

Пример сравнения реализации цикла данных в статическом и динамическом вычислительных графах показан на рис. 4.8.

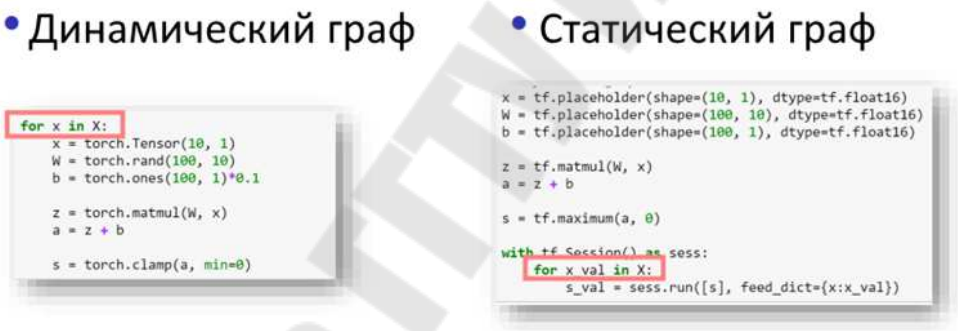


Рис. 4.8. Сравнение реализации статического и динамического графов

4.1.2. Тензорные вычисления

Основным объектом, которым манипулируют в *Tensorflow*, является тензор [18]. *TensorFlow*, как видно из названия, является платформой для определения и выполнения вычислений с использованием тензоров. Внутри *TensorFlow* тензоры представлены в виде n-мерных массивов базовых типов данных.

При написании программы *TensorFlow* основным объектом, которым вы манипулируете и передаете, является *tf.Tensor*. Программы *TensorFlow* работают, сначала создавая граф объектов *tf.Tensor* и подробно описывая, как вычисляется каждый тензор на основе других

доступных тензоров, а затем запуская части этого графа для получения результатов вычисления.

tf.Tensor обладает следующими параметрами:

– тип данных (*float*32, *int*32, или *string*, например);

– размеры (*shape*).

Все элементы тензора имеют одинаковый тип данных, и он всегда известен. Размеры (количество измерений и размер каждого измерения) могут быть известны только частично. Результатом большинства операций являются тензоры с известными размерами, если размеры на входе также полностью известны, но в некоторых случаях узнать размеры тензора можно только во время исполнения графа.

Основные виды тензоров следующие:

– *tf.Variable*;

– *tf.constant*;

– *tf.placeholder*;

– *tf.SparseTensor*.

За исключением *tf.Variable* значение тензора неизменяемо, т. е. в контексте одного выполнения тензор может иметь только одно значение. Однако вычисление одного и того же тензора дважды может вернуть различные значения – например, тот же тензор может быть результатом чтения данных с диска, или генерации случайного числа.

Ранг объекта *tf.Tensor* – это количество его измерений. Синонимами ранга являются порядок, степень, размерность. Как показывает табл. 4.1, каждый ранг в *Tensorflow* соответствует некоторой математической сущности:

```
mymat = tf.Variable([[7],[11]], tf.int16)
myxor = tf.Variable([[False, True],[True, False]], tf.bool)
linear_squares = tf.Variable([[4], [9], [16], [25]], tf.int32)
squarish_squares = tf.Variable([ [4, 9], [16, 25] ], tf.int32)
rank_of_squares = tf.rank(squarish_squares)
mymatC = tf.Variable([[7],[11]], tf.int32)
```

Таблица 4.1

Перечень рангов тензоров Tensorflow

Ранг	Математическая сущность
0	Скаляр (только величина)
1	Вектор (величина и направление)
2	Матрица (таблица чисел)
3	3-Тензор (куб чисел)
n	*n*-Тензор (идея понятна)

Ранг 0. Следующий фрагмент демонстрирует создание нескольких переменных ранга 0:

```
mammal = tf.Variable("Elephant", tf.string)
ignition = tf.Variable(451, tf.int16)
floating = tf.Variable(3.14159265359, tf.float64)
its_complicated = tf.Variable(12.3 - 4.85j, tf.complex64)
```

Строка считается единым объектом в TensorFlow, а не последовательностью символов. Возможно иметь строковые скаляры, векторы строк и т. д.

Ранг 1. Для создания объекта *tf.Tensor* ранга 1 можно передать список элементов в качестве начальных значений. Например:

```
mystr = tf.Variable(["Hello"], tf.string)
cool_numbers  = tf.Variable([3.14159, 2.71828], tf.float32)
first_primes = tf.Variable([2, 3, 5, 7, 11], tf.int32)
its_very_complicated = tf.Variable([12.3 - 4.85j, 7.5 - 6.23j],
tf.complex64)
```

Ранг 2 объекта *tf.Tensor* состоит как минимум из одной строки и одного столбца.

Тензоры более высокого ранга аналогично состоят из *n*-мерных массивов. Например, при обработке изображений используется много тензоров ранга 4, с размерностями, соответствующими номеру примера в пакете, высоте изображения, ширине изображения и цветовому каналу:

```
my_image = tf.zeros([10, 299, 299, 3])  # размер пакета x высо-
та x ширина x количество цветовых каналов
```

Для определения ранга объекта *tf.Tens*or необходимо вызвать метод *tf.rank*. Например, следующий метод программно определяет ранг *tf.Tensor*, заданного выше:

```
r = tf.rank(my_image)
# После запуска графа, r станет равным 4.
```

Ссылки на срезы *tf.Tensor*. Поскольку *tf.Tensor* – это *n*-мерный массив ячеек, для получения доступа к одной ячейке в *tf.Tensor* вам нужно указать *n* индексов.

Для тензоров ранга 0 (скаляров), индексы не нужны, поскольку это уже просто число. Для тензора ранга 1 (вектор) передача единственного индекса даст вам доступ к числу:

```
my_scalar = my_vector[2]
```

Стоит отметить, что индекс, передаваемый в [ ], может сам быть скаляром *tf.Tensor*, если необходимо динамически выбрать элемент из вектора.

Для тензоров ранга 2 или выше ситуация отличается. Для *tf.Tensor* ранга 2 передача двух чисел возвращает, как и ожидалось, скаляр:

```
my_scalar = my_matrix[1, 2]
```

Однако передача одного числа возвращает подвектор матрицы следующим образом:

```
my_row_vector = my_matrix[2]
my_column_vector = my_matrix[:, 3]
```

Нотация в синтаксисе выделения подмассива в python используется как «игнорировать данное измерение». Это полезно в тензорах высокого ранга, поскольку позволяет получить доступ к подвекторам, подматрицам и даже другим подтензорам.

Размеры тензора – это количество элементов в каждом измерении. Документация *TensorFlow* использует три условных обозначения для описания размерности тензора: ранг, размеры и количество измерений. Таблица 4.2 показывает, как они соотносятся друг с другом.

Таблица 4.2

Размеры тензоров Tensorflow

Ранг	Размеры	Количество измерений	Пример
0	[ ]	0-*D*	0-*D* тензор. Скаляр
1	[*D*0]	1-*D*	1-*D* тензор размера [5]
2	[*D*0, *D*1]	2-*D*	2-*D* тензор размера [3, 4]
3	[*D*0, *D*1, *D*2]	3-*D*	3-*D* тензор размера [1, 4, 3]
n	[*D*0, *D*1, … *Dn* – 1]	*n*-*D*	Тензор размера [*D*0, *D*1, … *Dn* – 1]

Размеры могут быть представлены в виде списков *Python*/кортежей целых чисел или с помощью *tf.TensorShape*.

Есть два способа получить размеры *tf.Tensor*. При построении графа имеет смысл выяснить, что известно о размерах тензора. Это можно сделать при помощи чтения свойства shape объекта *tf.Tensor*. Этот метод возвращает объект *TensorShape*, который является удобным способом представления частично определенных размеров (поскольку при построении графа не все размеры могут быть полностью известны).

Также можно получить *tf.Tensor*, который представляет полностью определенные размеры другого *tf.Tensor* во время выполнения. Это делается вызовом операции *tf.shape*. Этим способом возможно построить граф, который манипулирует размерами тензоров, строя другие тензоры, зависящие от динамических размеров входных *tf.Tensor*.

Например, так можно сделать вектор нулей того же размера, что и число столбцов данной матрицы:

```
zeros = tf.zeros(my_matrix.shape[1])
```

Изменение размеров *tf.Tensor*. Количество элементов тензора – это произведение всех его измерений. Количество элементов скаляра всегда равно 1. Так как множество разных размеров могут давать одно и то же число элементов, зачастую имеет смысл менять размеры *tf.Tensor*, не изменяя его элементы. Это может быть сделано с помощью *tf.reshape*.

Ниже показан пример изменения размеров тензора:

```
rank_three_tensor = tf.ones([3, 4, 5])
matrix = tf.reshape(rank_three_tensor, [6, 10])  # Преобразование существующих данных в матрицу 6x10
matrixB = tf.reshape(matrix, [3, -1])  # Преобразование существующих данных в матрицу 3x20. -1 говорит reshape что нужно посчитать размер этого измерения.
matrixAlt = tf.reshape(matrixB, [4, 3, -1])  # Преобразование существующих данных в тензор 4x3x5

# Отметим, что число элементов в преобразованных тензорах должно совпадать
# с изначальным количеством. Поэтому следующий пример породит
# ошибку поскольку нет такого значения для последнего измерения
# при котором совпадет количество элементов.
yet_another = tf.reshape(matrixAlt, [13, 2, -1])  # ERROR!
```

В дополнение к размерности у тензоров есть тип данных. У *tf.Tensor* не может быть более одного типа данных. Однако возможно сериализовать произвольные структуры данных в string и хранить их в *tf.Tensor*.

Можно преобразовать *tf.Tensor* из одного типа данных в другой, используя *tf.cast*:

```
    # Преобразование константного целочисленного тензора в тензор с
плавающей запятой.
    float_tensor = tf.cast(tf.constant([1, 2, 3]),
dtype=tf.float32)
```

Чтобы посмотреть тип данных *tf.Tensor*, необходимо использовать свойство *Tensor.dtype*.

При создании *tf.Tensor* из объекта *python* можно опционально указать тип данных. Если этого не сделать, *TensorFlow* выберет тип данных, который может представлять данные. *TensorFlow* преобразует целые числа Python в *tf.int*32, а числа с плавающей запятой в *tf.float*32. В других случаях *TensorFlow* использует те же правила что и numpy при конвертации массивов.

После завершения построения вычислительного графа становится возможным запустить вычисление, которое сгенерирует определенный *tf.Tensor*, и извлечь присвоенное ему значение. Это часто полезно для отладки, а также для работы большей части *TensorFlow*.

Самый простой способ оценить Tensor – использовать метод *Tensor.eval*. Например:

```
constant = tf.constant([1, 2, 3])
tensor = constant * constant
print(tensor.eval())
```

Метод *eval* работает только тогда, когда активна сессия по умолчанию *tf.Session*. Tensor.eval возвращает массив numpy с тем же содержанием что и тензор.

Иногда невозможно оценить *tf.Tensor* без контекста, потому что его значение может зависеть от динамической информации, которая недоступна. Например, тензоры, зависящие от *placeholder*, не могут быть оценены без предоставления значения для *placeholder*:

```
p = tf.placeholder(tf.float32)
t = p + 1.0
```

```
    t.eval()  # Это не сработает, потому что  placeholder не полу-
чил значение.
    t.eval(feed_dict={p:2.0})  # Это сработает, потому что мы пере-
дает значение в placeholder.
```

Может использоваться любой *tf.Tensor*, не только *placeholder*.

Конструкции других моделей могут усложнить оценивание *tf.Tensor*. *TensorFlow* не может оценить напрямую *tf.Tensor*, определенные внутри функций или внутри конструкций потока управления. Если *tf.Tensor* зависит от значения из очереди, оценка *tf.Tensor* сработает только когда что-то поместят в очередь; в противном случае оценка тензора зависнет. При работе с очередями не забывайте вызвать *tf.train.start_queue_runners* перед оценкой любого *tf.Tensor*.

4.1.3. Датасеты и их использование

Важным этапом работы с нейронными сетями является этап подготовки данных, результатом которого является обработанный набор очищенных данных, пригодных для обработки алгоритмами машинного обучения. Такая выборка, называемая датасет (*dataset*), нужна для тренировки, чтобы обучить систему и затем использовать ее для решения реальных задач. Однако поскольку в процессе обучения необходимо оценивать качество модели, различают несколько типов выборок.

Dataset для машинного обучения – это обработанная и структурированная информация в табличном виде. Строки такой таблицы называются объектами, а столбцы – признаками. Различают два вида признаков:

– независимые переменные – предикторы;

– зависимые переменные – целевые признаки, которые вычисляются на основе одного или нескольких предикторов.

Признаковое описание характерно для задач классификации, когда имеется выборка – конечное множество объектов, для которых известно, к каким классам они относятся. Классовая принадлежность остальных объектов неизвестна. В процессе машинного обучения строится модель, способная классифицировать произвольный объект из исходного множества.

В зависимости от варианта задачи классификации целевой признак может выглядеть по-разному:

– один столбец с двоичными значениями (1/0, *TRUE*/*FALSE* и др.): двухклассовая классификация (*binary classification*), когда каждый объект принадлежит только одному классу;

– несколько столбцов с двоичными значениями: задача классификации с пересекающимися классами (*multi-label classification*), когда один объект может принадлежать нескольким классам;

– один столбец с действительными значениями: регрессионный анализ, когда прогнозируется одна величина;

– несколько столбцов с действительными значениями: задача множественной регрессии, когда прогнозируется несколько величин.

Первичный набор исходных данных принято называть генеральной совокупностью. Процесс формирования выборок из генеральной совокупности называется «порождение данных». Выборка – это конечное подмножество элементов генеральной совокупности, изучив которое можно понять поведение исходного множества. Например, генеральная совокупность состоит из 150 тысяч посетителей сайта, а в выборку попали 250 из них.

Вероятностная модель порождения данных предполагает, что выборка из генеральной совокупности формируется случайным образом. Если все ее элементы одинаково случайно и независимо друг от друга распределены по исходному множеству (генеральной совокупности), выборка называется простой. Простая выборка является математической моделью серии независимых опытов и, как правило, используется для машинного обучения. При этом для каждого этапа машинного обучения необходим свой набор данных:

– для непосредственного обучения модели нужна обучающая выборка (*training sample*), по которой производится настройка (оптимизация параметров) алгоритма;

– для оценки качества модели используется тестовая (контрольная) выборка (*test sample*), которая, в идеальном случае, не должна зависеть от обучающей;

– для выбора наилучшей модели машинного обучения понадобится проверочная (валидационная) выборка (*validation sample*), которая также не должна пересекаться с обучающей.

Существует множество готовых датасетов, на которых проходит наладка, «обкатка» и сравнение новых архитектур нейронных сетей. Данные датасеты охватывают различные области машинного обучения и покрывают широкий спектр задач.

Далее перечислены наиболее популярные при изучении глубокого обучения датасеты.

В компьютерном зрении самые простые датасеты связаны с рукописными цифрами или буквами. Существует несколько наборов

данных с цифрами, которые изначально появились для экспериментов с моделями по распознаванию изображений.

Среди этих датасетов (рис. 4.9) перечислим следующее:

– *MNIST* – рукописные цифры, не нуждается в дополнительном представлении;

– *USPS* – рукописные цифры в низком разрешении;

– *SVHN* – номера домов с *Google Street View*;

– *Synth Numbers* – синтетические числа, как следует из названия.

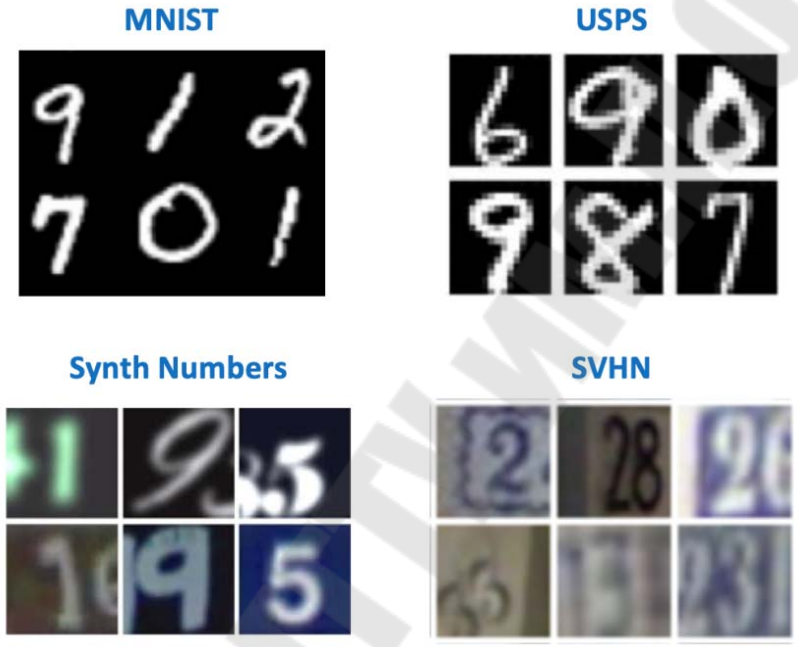


Рис. 4.9. Пример датасетов рукописных цифр

Еще одной парой датасетов для обучения модели являются дорожные знаки:

– *Synth Signs* – изображения дорожных знаков, сгенерированные так, чтобы они были похожи на настоящие знаки на улице;

– *GTSRB* – довольно известная база для распознавания, содержащая знаки с немецких дорог.

4.1.4. Доменная адаптация

Доменная адаптация (*domain adaptation*) является одной из областей машинного обучения. Суть этой адаптации заключается в обучении модели на данных из домена-источника (*source domain*) так, чтобы она показывала сравнимое качество на целевом домене (*target domain*). Например, *source domain* может представлять собой синтетические данные, которые можно «дешево» сгенерировать, а *target*

domain – фотографии пользователей. Тогда задача *domain adaptation* заключается в тренировке модели на синтетических данных, которая будет хорошо работать с «реальными» объектами.

Исследования в доменной адаптации затрагивают вопросы использования в новой задаче предыдущего накопленного нейросетью опыта. Сможет ли сеть выделить некоторые характерные особенности из домена-источника и использовать их в целевом домене?

Кроме того, доменная адаптация может помочь решить одну из фундаментальных проблем глубокого обучения: для тренировки больших сетей с высоким качеством распознавания необходимо очень большое количество данных, которые на практике не всегда доступны. Одним из решений может быть использование методов *domain adaptation* на синтетических данных, которые можно нагенерировать практически в неограниченном количестве.

Довольно часто в прикладных задачах встречается случай, когда для обучения доступны данные только из одного домена, а применять модель необходимо на другом домене. Например, сеть, определяющую эстетическое качество фотографии, можно обучить на доступной в сети базе, собранной с сайта фотолюбителей. А применять эту сеть планируется на обычных фотографиях, уровень качества которых в среднем отличается от уровня фото со специализированного фотосайта. В качестве варианта решения можно рассматривать адаптацию модели под обычные неразмеченные фотографии.

Такие теоретические и прикладные вопросы лежат в области *domain adaptation*. Главная идея *deep domain adaptation* заключается в том, чтобы обучить на домене-источнике глубокую нейронную сеть, которая будет переводить изображение в такое векторное представление (*embedding*) (обычно это последний слой сети), что при использовании его на целевом домене получится высокое качество.

Существует довольно обширная и разнообразная классификация методов доменной адаптации. Далее приводится упрощенное деление методов по их ключевым особенностям. Современные методы *deep domain adaptation* можно разделить на три большие группы:

– *Discrepancy-based*: подходы, основанные на минимизации расстояния между векторными представлениями на исходном и целевом доменах с помощью введения этого расстояния в *loss*-функцию;

– *Adversarial-Based*: эти подходы используют состязательную (*adversarial*) *loss*-функцию, появившуюся в *GAN*'ах, для обучения сети, инвариантной относительно домена. Методы этого семейства активно развиваются в последние пару лет;

– смешанные методы, которые не используют *adversarial loss*, но применяют идеи из *discrepancy-based* семейства, а также последние наработки из глубокого обучения: *self-ensembling*, новые слои, *loss*-функции и т. п.

Discrepancy-based. Когда возникает задача адаптации модели под новые данные, первое, что приходит на ум, это использование *fine-tuning*, т. е. дообучения модели на новых данных. Для этого необходимо учитывать меру несоответствия (*discrepancy*) между доменами. Такой вид доменной адаптации можно разделить на три подхода: Class Criterion, Statistical criterion и Architecture Criterion.

Class Criterion. Методы из этого семейства в основном применяются, когда нам доступны размеченные данные из целевого домена. Одним из популярных вариантов *Class Criterion* является подход *Deep transfer metric learning*. Как следует из названия, он основан на *metric learning*, суть которого заключается в обучении такого векторного представления, получаемого из нейронной сети, что представители одного класса будут близки друг к другу в этом представлении по заданной метрике (чаще всего используют L^2 или косинусную метрики). В статье *Deep transfer metric learning* (*DTML*) для реализации этого подхода используется *loss*, состоящий из суммы слагаемых:

– близость представителей одного класса друг к другу (*intraclass compactness*);

– увеличение расстояния между представителями разных классов (*interclass separability*);

– метрика *Maximum Mean Discrepancy* (*MMD*) между доменами. Эта метрика относится к семейству *statistical criterion* (см. ниже), но используется и в *class criterion*.

MMD между доменами записывается в виде

$$MMD^2\left(D_s, D_t\right) = \left\| \frac{1}{M}\sum_{i=1}^{M}\varphi\left(x_i^s\right) - \frac{1}{N}\sum_{j=1}^{N}\varphi\left(x_j^t\right)\right\|_H^2, \qquad (4.13)$$

где $\varphi(x)$ – это некоторое ядро, в нашем случае – векторное представление сети; $x_i^s, i \in 1\ldots M$ – данные из исходного домена; $x_i^t, i \in 1\ldots N$ – данные из целевого домена. Таким образом, при минимизации метрики *MMD* во время обучения подбирается такая сеть $\varphi(x)$, чтобы ее средние векторные представления на обоих доменах были близки. Основная идея *DTML* представлена на рис. 4.10.

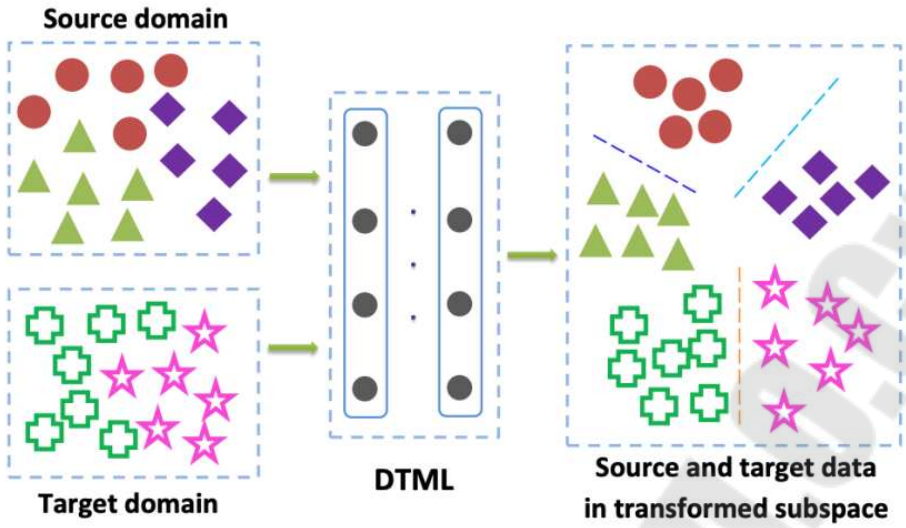


Рис. 4.10. Схема работы *DTML*

Statistical criterion. Методы, относящиеся к этому семейству, используются для решения задачи *unsupervised domain adaptation*. Случай, когда целевой домен не размечен, встречается во многих задачах, и все методы доменной адаптации, которые будут рассмотрены далее, решают именно такую задачу.

Подходы, основанные на статистическом критерии, пытаются измерить разницу между распределениями векторного представления сети, получаемыми из данных исходного и целевого доменов. Затем они используют вычисленную разницу для сближения этих двух распределений.

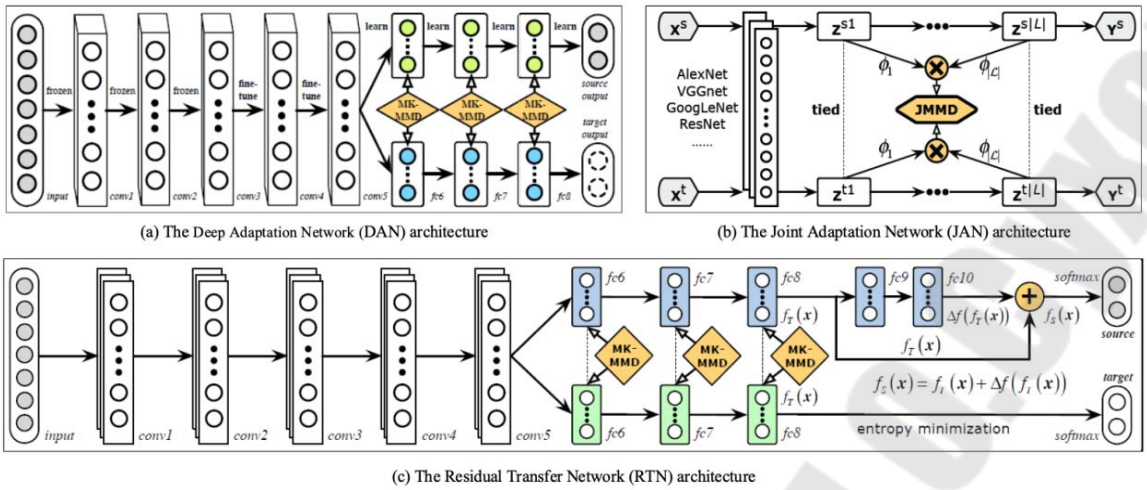
Одним из таких критериев является уже описанный выше *Maximum Mean Discrepancy* (*MMD*). Его варианты используются в нескольких методах:

– *Deep adaptation network* (*DAN*);

– *Joint adaptation network* (*JAN*);

– *Residual transfer network* (*RTN*). *RTN* показывает неплохие результаты для пары *MNIST -> SVHN*: 90,66 % точности на целевом домене.

Схемы этих трех методов представлены ниже. В них варианты *MMD* используются для определения разницы между распределениями на слоях сверточной нейронной сети, примененной к исходному и целевому доменам. Каждый из них использует модификацию *MMD* в качестве *loss*'а между слоями сверточных сетей (желтые фигуры на рис. 4.11).



(a) The Deep Adaptation Network (DAN) architecture

(b) The Joint Adaptation Network (JAN) architecture

(c) The Residual Transfer Network (RTN) architecture

Рис. 4.11. Схема вариантов метода доменной адаптации *MMD*

Критерий *CORAL* (*CORrelation ALignment*) и его расширение с помощью глубоких сетей *Deep CORAL* направлены на то, чтобы выучить такое представление данных, при котором максимально совпадали бы между собой статистики второго порядка между доменами. Для этого используются ковариационные матрицы векторных представлений сети. Сближение статистик второго порядка на обоих доменах в некоторых случаях позволяет получить лучшие, чем для *MMD*, результаты адаптации.

$$L_{CORAL} = \frac{1}{4d^2}\left\|C_S - C_T\right\|_F^2, \tag{4.14}$$

где $\left\|C_S - C_T\right\|_F^2$ – квадрат матричной нормы Фробениуса; C_S и C_T – ковариационные матрицы данных из исходного и целевого доменов соответственно; d – размерность векторного представления.

На датасете *Office* среднее качество адаптации с использованием *Deep CORAL* для пар доменов *Amazon* и *Webcam*: 72,1 %. На доменах дорожных знаков *Synth Signs -> GTSRB* результат также весьма средний: 86,9 % точности на *target domain*.

Развитием идей *MMD* и *CORAL* является критерий *Central Moment Discrepancy* (*CMD*), который сравнивает центральные моменты данных из исходного и целевого доменов всех порядков до K включительно (K – параметр алгоритма). На датасете *Office* среднее качество адаптации *CMD* для пар доменов *Amazon* и *Webcam* составляет 77,0 %.

Architecture Criterion. Алгоритмы этого типа строятся на предположении, что основная информация, которая отвечает за адаптацию на новый домен, заложена в параметрах нейронной сети.

В ряде работ при обучении сетей для исходного и целевого доменов с помощью *loss*-функций для каждой пары слоев изучается на весах этих слоев информация, инвариантная относительно домена. Пример таких архитектур приведен на рис. 4.12.

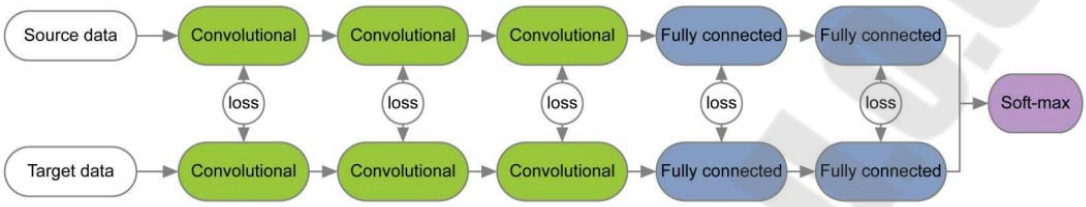


Рис. 4.12. Схема архитектуры *Architecture Criterion*

Предполагается, что в весах сети заложена информация, связанная с классами, на которых учится сеть, а доменная информация заложена в статистиках (среднем и стандартном отклонении) слоев *Batch Normalization* (*BN*). Следовательно, для адаптации необходимо пересчитать эти статистики на данных из целевого домена. Использование этого приема вместе с *CORAL* способно улучшить качество адаптации на датасете *Office* для пар доменов *Amazon* и *Webcam* до 75,0 %. Затем было показано, что использование слоя *Instance Normalization* (*IN*) вместо *BN* еще больше улучшает качество адаптации. В отличие от *BN*, который нормализует входной тензор по батчам, *IN* вычисляет статистику для нормализации по каналам и, следовательно, не зависит от батча.

Adversarial-Based Approaches. В последние 1–2 года большинство результатов в *deep domain adaptation* связаны с *adversarial-based* подходом. Это во многом обусловлено стремительным развитием и ростом популярности Генеративно-состязательных сетей (*Generative Adversarial Networks*, *GAN*), потому что *adversarial-based* подход к доменной адаптации использует ту же состязательную (*adversarial*) целевую функцию при обучении, что и *GAN*. Оптимизируя ее, такие методы *deep domain adaptation* минимизируют расстояние между эмпирическими распределениями векторных представлений данных на исходном и целевом доменах. Обучая сеть таким образом, ее стараются сделать инвариантной относительно домена.

Существует два больших подхода в *adversarial-based domain adaptation*, которые отличаются тем, используется или нет генератор *G*.

Non-Generative Models. Ключевой особенностью методов из этого семейства является обучение нейронной сети с инвариантным по отношению к исходному и целевому доменам векторным представлением. Тогда обученную на размеченном *source domain* сеть можно будет использовать на *target domain*, в идеале – практически без потери качества классификации.

Представленный в 2015 г. алгоритм *Domain-Adversarial Training of Neural Networks* (*DANN*) состоит из трех частей:

1) основной сети, с помощью которой получается векторное представление (*feature extractor*) (зеленая часть на рис. 4.13);

2) «головы», отвечающей за классификацию на исходном домене (синяя часть);

3) «головы», которая обучается отличать данные из исходного домена от целевого (красная часть).

При обучении с помощью градиентного спуска (*SGD*) (стрелки к *input* на рисунке) минимизируются классификационный и доменный *loss*'ы. К тому же при обратном распространении ошибки в обучении для «головы», отвечающей за домены, используется слой *Gradient reversal layer* (черная часть на рис. 4.13), который умножает проходящий через него градиент на негативную константу, увеличивая доменный *loss*. Этим добиваются того, что распределения векторных представлений на обоих доменах становятся близки.

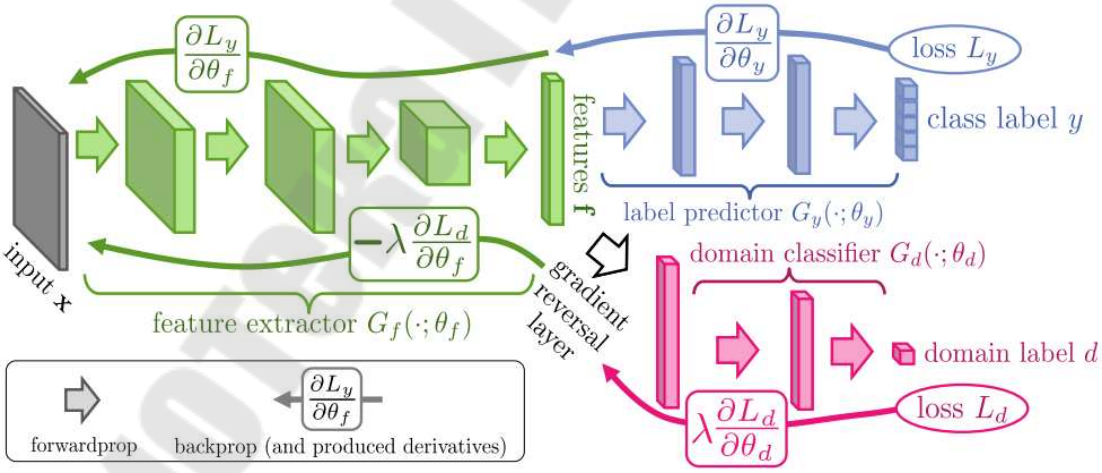


Рис. 4.13. Схема согласования доменов

Следующим важным представителем семейства *non-generative models* является метод *Adversarial Discriminative Domain Adaptation* (*ADDA*), который подразумевает разделение сети для исходного домена и сети для целевого домена. Алгоритм состоит из следующих шагов:

1. Сначала классифицирующую сеть обучаем на исходном домене. Ее векторное представление обозначим M_s, а X_s – исходный домен.

2. Теперь инициализируем нейронную сеть для целевого домена с помощью обученной сети из предыдущего шага. Обозначим ее M_t, а X_t – целевой домен.

3. Перейдем к *adversarial*-тренировке: будем обучать дискриминатор при фиксированных и с помощью следующей целевой функции:

$$\min_D L_{adv_D}(X_s, X_t, M_s, M_t) =$$

$$= \left(-\mathbb{E}_{x_s \sim X_s}\right) - \mathbb{E}_{x_t \sim X_t}\left[\log\left(1 - D\left(M_t(x_t)\right)\right)\right]. \quad (4.15)$$

4. Заморозим дискриминатор и дообучим на целевом домене:

$$\log D\left(M_s(x_s)\right) \min_{M_s, M_t} L_{adv_M}(X_s, X_t, D) = -\mathbb{E}_{x_t \sim X_t}\left[\log D\left(M_t(x_t)\right)\right]. \quad (4.16)$$

Шаги 3 и 4 повторяются несколько раз. Суть *ADDA* заключается в том, что мы сначала обучаем хороший классификатор на размеченном исходном домене, а затем с помощью *adversarial*-обучения адаптируем так, чтобы векторные представления классификатора на обоих доменах были близки. Графически алгоритм можно представить следующим образом (рис. 4.14).

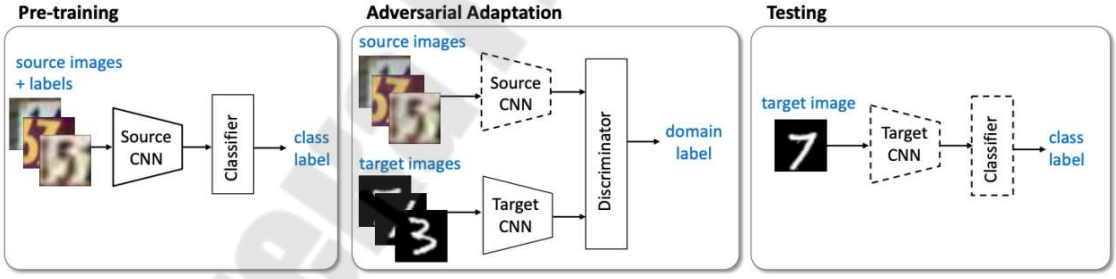


Рис. 4.14. Схема метода *Adversarial Discriminative Domain Adaptation*

4.2. Обзор библиотек глубокого обучения

4.2.1. TensorFlow

TensorFlow – фреймворк для глубокого машинного обучения, разрабатываемый в *Google Brain* [18].

Наиболее удобным клиентским языком работы с *TensorFlow* является *Python*, но доступны и экспериментальные интерфейсы

на *JavaScript*, *C++*, *Java* и *Go*. Сообщество *open source* разработало также решения для *C#* и *Julia*.

TensorFlow оперирует статическим вычислительным графом. То есть вначале мы определяем граф, далее запускаем вычисления и, если необходимо внести изменения в архитектуру, заново обучаем модель. Такой подход выбран ради эффективности, но многие современные нейросетевые инструменты умеют учитывать уточнения в процессе обучения без существенной потери скорости обучения [17].

Работа с *TensorFlow* организуется вокруг построения и выполнения графа вычислений. Граф вычислений – это конструкция, которая описывает то, каким образом будут проводиться вычисления. Основа *TensorFlow* – это создание структуры, задающей порядок вычислений. Программы естественным образом структурируются на две части – составление графа вычислений и выполнение вычислений в созданных структурах.

В *TensorFlow* граф состоит из плейсхолдеров, переменных и операций. Из этих элементов можно собрать граф, в котором будут вычисляться тензоры. Тензоры – многомерные массивы, они служат «топливом» для графа. Тензором может быть как отдельное число, вектор признаков из решаемой задачи или изображение, так и целый батч описаний объектов или массив из изображений. Вместо одного объекта можно передать в граф массив объектов и для него будет вычислен массив ответов. Работа *TensorFlow* с тензорами похожа на то, как обрабатывает массивы *numpy*, в функциях которого можно указать ось массива, относительно которой будет выполняться вычисление [18].

Сессии. Вычислительные графы выполняются в сессиях. Объект сессии (*tf.Session*) скрывает в себе контекст выполнения графа – необходимые ресурсы, вспомогательные классы, адресные пространства.

Существует два типа сессий – обычные, которые реализованы в *tf.Session*, и интерактивные (*tf.InteractiveSession*). Разница между ними в том, что интерактивная сессия больше подходит для выполнения в консоли и сразу определяет себя как сессия по умолчанию. Основной эффект – объект сессии не нужно передавать в функции вычисления как параметр.

Тензорная структура данных. Тензорные элементы используются в качестве основных структур данных в языке *TensorFlow*. Тензорные элементы представляют соединительные ребра в любой блок-схеме, называемой графиком потока данных. Тензоры определяются как многомерный массив или список.

4.2.2. PyTorch

Torch – библиотека для научных вычислений с широкой поддержкой алгоритмов машинного обучения. Разрабатывается *Idiap Research Institute*, *New York University* и *NEC Laboratories America*, начиная с 2000 г., распространяется под лицензией *BSD* [19], [20].

Библиотека реализована на языке *Lua* с использованием *C* и *CUDA*. Быстрый скриптовый язык *Lua* в совокупности с технологиями *SSE*, *OpenMP*, *CUDA* позволяют *Torch* показывать неплохую скорость по сравнению с другими библиотеками. На данный момент поддерживаются операционные системы *Linux*, *FreeBSD*, *Mac OS X*. Основные модули также работают и на *Windows*. В зависимостях *Torch* находятся пакеты *imagemagick*, *gnuplot*, *nodejs*, *npm* и др. [21].

Библиотека состоит из набора модулей, каждый из которых отвечает за различные стадии работы с нейросетями. Так, например, модуль *nn* обеспечивает конфигурирование нейросети (определение слоев и их параметров), модуль *optim* содержит реализации различных методов оптимизации, применяемых для обучения, а *gnuplot* предоставляет возможность визуализации данных (построение графиков, показ изображений и т. д.). Установка дополнительных модулей позволяет расширить функционал библиотеки.

Torch позволяет создавать сложные нейросети с помощью механизма контейнеров [20]. Контейнер – это класс, объединяющий объявленные компоненты нейросети в одну общую конфигурацию, которая в дальнейшем может быть передана в процедуру обучения. Компонентом нейросети могут быть не только полносвязные или сверточные слои, но и функции активации или ошибки, а также готовые контейнеры. *Torch* позволяет создавать следующие слои:

– полносвязный слой (*Linear*);

– функции активации: гиперболический тангенс (*Tanh*), выбор минимального (*Min*) или максимального (*Max*), *softmax*-функция (*SoftMax*) и др.;

– сверточные слои: свертка (*Convolution*), прореживание (*SubSamp-ling*), пространственное объединение (*MaxPooling*, *AveragePooling*, *LPPoolin*g), разностная нормализация (*SubtractiveNormalization*).

Функции ошибки: средне-квадратичная ошибка (*MSE*), кроссэнтропия (*CrossEntropy*) и т. д.

При обучении могут использоваться следующие методы оптимизации:

– стохастический градиентный спуск (*SGD*);

– усредненный стохастический градиентный спуск (*Averaged SGD*);

– алгоритм Бройдена–Флетчер –Гольдфарба–Шанно (*L-BFGS*) [10];

– метод сопряженных градиентов (*Conjugate Gradient*, *CG*).

4.2.3. Keras

Keras – открытая нейросетевая библиотека, написанная на языке *Python* [23]. Она представляет собой надстройку над фреймворками *Deeplearning*4*j*, *TensorFlow* и *Theano* [24]. Нацелена на оперативную работу с сетями глубинного обучения, при этом спроектирована так, чтобы быть компактной, модульной и расширяемой. Она была создана как часть исследовательских усилий проекта *ONEIROS* (*Open-ended Neuro-Electronic Intelligent Robot Operating System*), а ее основным автором и поддерживающим является Франсуа Шолле (фр. *François Chollet*), инженер *Google*.

Keras – рекомендуемая библиотека для глубокого изучения *Python*, особенно для начинающих. Его минималистичный, модульный подход позволяет с легкостью построить и запустить глубокие нейронные сети [25].

Типичные рабочие процессы *Keras* выглядят так:

– определите ваши тренировочные данные: входной тензор и целевой тензор;

– определите сеть слоев (или модель), которая отображает входные данные для наших целей;

– настройте процесс обучения, выбрав функцию потерь, оптимизатор и некоторые показатели для мониторинга;

– повторяйте данные тренировки, вызывая метод *fit*( ) вашей модели.

В *Keras* доступно два основных типа моделей: последовательная модель и класс *Model*, используемый с функциональным *API*.

Эти модели имеют ряд общих методов и атрибутов:

– *model.layers* – это плоский список слоев, составляющих модель;

– *model.inputs* – список входных тензоров модели;

– *model.outputs* – список выходных тензоров модели;

– *model.summary*( ) – печатает краткое представление вашей модели;

– *model.get_config*( ) – возвращает словарь, содержащий конфигурацию модели;

– *model.get_weights*( ) – возвращает список всех весовых тензоров в модели в виде массивов *Numpy*;

– *model.set_weights*(*weights*) – устанавливает значения весов модели из списка массивов *Numpy*. Массивы в списке должны иметь ту же форму, что и возвращаемые *get_weights*( );

– *model.to_json*( ) – возвращает представление модели в виде строки *JSON*. Обратите внимание, что представление не включает веса, только архитектуру;

– *model.to_yaml*( ) – возвращает представление модели в виде строки *YAML*. Обратите внимание, что представление не включает веса, только архитектуру;

– *model.save_weights*(*filepath*) – сохраняет вес модели в виде файла *HDF*5;

– *model.load_weights*(*filepath*, *by_name=False*) – загружает вес модели из файла *HDF*5 (созданного *save_weights*). По умолчанию ожидается, что архитектура не изменится.

Методы *API* последовательной модели. Компиляция – *Compile* – настраивает модель для обучения.

Аргументы:

– *optimizer*: строка (имя оптимизатора) или экземпляр оптимизатора;

– *loss* (потеря): строка (имя целевой функции), или целевая функция или *loss* экземпляр. Смотрите потери. Если модель имеет несколько выходов, вы можете использовать разные потери на каждом выходе, передав словарь или список потерь. Значение потерь, которое будет минимизировано моделью, станет тогда суммой всех индивидуальных потерь;

– *metrics*: список метрик, которые будут оцениваться моделью во время обучения и тестирования. Как правило, вы будете использовать *metrics* = ['*accuracy*']. Чтобы указать разные метрики для разных выходов модели с несколькими выходами, вы также можете передать словарь, например *metrics* = {'*output_a*': '*accuracy*', '*output_b*': ['*accuracy*', '*mse*']}. Вы также можете передать список (*len* = *len* (выводы)) списков метрик, таких как *metrics* = [['*accuracy*'], ['*accuracy*', '*mse*']] или *metrics* = ['*accuracy*', ['*accuracy*', '*mse*']];

– *loss_weights*: необязательный список или словарь, задающий скалярные коэффициенты (числа *Python*) для взвешивания вкладов по-

терь в различные выходные данные модели. Значение потерь, которое будет минимизировано моделью, затем станет взвешенной суммой всех индивидуальных потерь, взвешенных по *loss_weights* коэффициентам. Если это список, ожидается, что он будет иметь соотношение 1 : 1 к выходам модели. Если диктат, ожидается, что выходные имена (строки) будут сопоставлены скалярным коэффициентам;

– *sample_weight_mode*: Если вам нужно сделать взвешивание выборки по временным шагам (2*D* веса), установите это значение «*temporal*». *None* по умолчанию используются веса выборки (1*D*). Если модель имеет несколько выходов, вы можете использовать разные *sample_weight_mode* на каждом выходе, передав словарь или список режимов;

– *weighted_metrics*: список метрик, которые будут оцениваться и взвешиваться по *sample_weight* или *class_weight* во время обучения и тестирования;

– *target_tensors*: по умолчанию *Keras* создаст заполнители для цели модели, которые будут снабжены целевыми данными во время обучения. Если вместо этого вы хотите использовать свои собственные целевые тензоры (в свою очередь, *Keras* не будет ожидать внешних данных *Numpy* для этих целей во время обучения), вы можете указать их с помощью *target_tensors*-аргумента. Это может быть один тензор (для модели с одним выходом), список тензоров или точные сопоставления выходных имен с целевыми тензорами;

– ***kwargs*: при использовании бэкэндов *Theano*/*CNTK* эти аргументы передаются в *K.function*. При использовании бэкэнда *TensorFlow* эти аргументы передаются в *tf.Session.run*;

– *Fit* – обучает модель для фиксированного числа эпох (итераций в наборе данных).

Аргументы:

– *x*: входные данные. Это может быть:

– массив *Numpy* (или похожий на массив) или список массивов (в случае, если модель имеет несколько входов);

– диктовое отображение (*dict mapping*) входных имен в соответствующий массив/тензоры, если модель имеет именованные входы;

– генератор или *keras.utils.Sequence* – возвращение (*inputs*, *targets*) или (*inputs*, *targets*, *sample weights*);

– *None* (*default*) – нет (по умолчанию) при подаче из тензоров, встроенных в каркас (например, тензоры данных *TensorFlow*);

– *y*: целевые данные. Как и входные данные *x*, это могут быть либо массив(ы) *Numpy*, тензор(ы), встроенные в платформу, список массивов *Numpy* (если модель имеет несколько выходных данных), либо *None* (по умолчанию), если они поступают из тензоров, встроенных в платформу (например, *TensorFlow*) тензоры данных). Если выходным слоям в модели присвоены имена, вы также можете передать словарь, отображающий выходные имена в массивы *Numpy*. *If x* является генератором или *keras.utils.Sequence* экземпляром, *y* указывать не следует (поскольку цели будут получены из *x*);

– *batch_size*: целое число или *None*. Количество образцов на обновление градиента. Если *batch_siz*e не указан, по умолчанию будет 32. Не указывайте *batch_size*, если ваши данные представлены в виде символических тензоров, генераторов или *Sequence* экземпляров (так как они генерируют пакеты);

– *epochs*: целочисленные. Количество эпох для обучения модели. Эпоха – это итерация по всему *x* и *y* предоставленным данным. Обратите внимание, что в сочетании с *initial_epoch*, *epochs* следует понимать как «конечную эпоху». Модель не обучается для ряда итераций, заданных *epochs*, а просто до тех пор, пока не будет достигнута эпоха индекса;

– *verbose*: *Integer*. 0, 1 или 2. Режим многословия. 0 = тихий, 1 = индикатор выполнения, 2 = одна строка за эпоху;

– *callbacks*: список *keras.callbacks*. *Callback* экземпляров. Список обратных вызовов, применяемых во время обучения и проверки (*if*). Смотрите обратные вызовы;

– *validation_split*: с плавающей точкой от 0 до 1. Доля данных обучения, которые будут использоваться в качестве данных проверки. Модель выделит эту часть обучающих данных, не будет обучаться им и будет оценивать потери и любые метрики модели на этих данных в конце каждой эпохи. Данные проверки выбираются из последних выборок *x* и *y* предоставленных данных перед перетасовкой. Этот аргумент не поддерживается, когда является генератором или *Sequence* экземпляром;

– *validation_data*: данные для оценки потерь и любые метрики модели в конце каждой эпохи. Модель не будет обучаться на этих данных. *validation_data* перекроет *validation_split*. *validation_data* может быть: [кортеж (*x_val*, *y_val*) массивов или тензоров (*x_val*, *y_val*, *val_sample_weights*)] [*Numpy* – кортеж массивов] [*Numpy* – набор данных или итератор набора данных]. Для первых двух случаев

batch_size должны быть предоставлены. Для последнего случая, *validation_steps* должны быть предоставлены;

– *shuffle*: *Boolean* (следует ли перемешивать данные тренировки перед каждой эпохой) или *str* (для «партии»). «Пакетная» – это специальная опция для работы с ограничениями данных *HDF*5; он тасуется кусками размером с партию. Не имеет эффекта, когда *steps_per_epoch* нет *None*;

– *class_weight*: необязательный словарь, отображающий индексы класса (целые числа) на значение веса (с плавающей запятой), используемое для взвешивания функции потерь (только во время обучения). Это может быть полезно для того, чтобы сказать модели «уделять больше внимания» выборкам из недопредставленного класса;

– *sample_weight*: необязательный массив весов *Numpy* для обучающих выборок, используемый для взвешивания функции потерь (только во время обучения). Вы можете либо передать плоский (1*D*) массив *Numpy* такой же длины, что и входные выборки (отображение весов и выборок 1 : 1), либо в случае временных данных вы можете передать двумерный массив с формой (*samples*, *sequence_length*), чтобы применить разный вес для каждого временного шага каждого образца. В этом случае вы должны обязательно указать *sample_weight_mode* = "*temporal*" в *compile*( ). Этот аргумент не поддерживается, когда применяется *x*-генератор или *Sequence* экземпляр, вместо этого предоставляют *sample_weights* в качестве третьего элемента *x*;

– *initial_epoch*: целое число. Эпоха, с которой начинается тренировка (полезно для возобновления предыдущего тренировочного заезда);

– *steps_per_epoch*: целое число или *None*. Общее количество шагов (партий образцов) до объявления одной эпохи законченной и начала следующей эпохи. При обучении с использованием входных тензоров, таких как тензоры данных *TensorFlow*, значение по умолчанию *None* равно числу выборок в вашем наборе данных, деленному на размер пакета, или 1, если это невозможно определить;

– *validation_steps*: только релевантно, если *steps_per_epoch* указано. Общее количество шагов (партий образцов) для проверки перед остановкой;

– *validation_steps*: релевантно, только если *validation_data* предоставлено и является генератором. Общее количество шагов (партий образцов), которые нужно нарисовать перед остановкой при выполнении проверки в конце каждой эпохи;

– *validation_freq*: уместно, только если предоставлены данные проверки. Целое число или список/кортеж/набор. Если целое число, указывает, сколько тренировочных эпох должно быть выполнено до того, как будет выполнен новый прогон проверки, например, *validation_freq* = 2 выполняет проверку каждые две эпохи. Если в списке, кортеже или наборе указываются эпохи, в которых нужно выполнять проверку, например, *validation_freq* = [1, 2, 10] выполняет проверку в конце 1-, 2- и 10-й эпох;

– *max_queue_size*: целое число. Используется только для генератора или *keras.utils.Sequence* входа. Максимальный размер очереди генератора. Если не указано, то *max_queue_size* по умолчанию будет 10;

– *workers*: целое число. Используется только для генератора или *keras.utils.Sequence* входа. Максимальное количество процессов, которые могут ускоряться при использовании потоков на основе процессов. Если не указано, то *workers* по умолчанию будет 1. Если 0, будет запускать генератор в основном потоке;

– *use_multiprocessing*: *Boolean*. Используется только для генератора или *keras.utils.Sequence* входа. Если *True*, используйте процессные потоки. Если не указано, то *use_multiprocessing* по умолчанию *False*. Обратите внимание, что, поскольку эта реализация опирается на многопроцессорность, вы не должны передавать невыгружаемые аргументы генератору, так как они не могут быть легко переданы дочерним процессам;

– ***kwargs*: используется для обратной совместимости.

4.2.4. MXNet

Это фреймворк для глубокого обучения, созданный *Apache*, который поддерживает изобилие языков, например, *Python*, *Julia*, *C++*, *R* или *JavaScript*. Он применяется в *Microsoft*, *Intel* и веб-сервисах *Amazon*.

Фреймворк *MXNet* известен своей высокой масштабируемостью, поэтому он используется большими компаниями в основном для распознавания речи и почерка, *NLP* и прогнозирования.

В его основе лежит динамический планировщик зависимостей, который автоматически распараллеливает как символьные, так и императивные операции «на лету». Слой оптимизации графов поверх него делает символьное исполнение быстрым и эффективным с точки зрения памяти. Фреймворк является легковесным и может масштабироваться на несколько *GPU* или несколько машин.

4.2.5. Microsoft Cognitive Toolkit

Microsoft Cognitive Toolkit представляет из себя открытый фреймворк для глубокого обучения, созданный для работы с действительно большими датасетами с поддержкой *Python*, *C++*, *C#* и *Java* [24].

CNTK обеспечивает эффективное обучение моделей для распознавания голоса, рукописного текста и картинок, также поддерживает *CNN* и *RNN*. Она используется в *Skype*, *Xbox* и *Cortana*.

Как и всегда, эксперты уже оценили его преимущества:

– показывает хорошую производительность и масштабируемость;

– предоставляет множество хорошо оптимизированных компонентов;

– предлагает поддержку *Apache Spark*;

– эффективен при использовании ресурсов.

4.2.6. Caffe

Разработка *Caffe* ведется с сентября 2013 г. Начало разработки положил *Yangqing Jia* во время его обучения в Калифорнийском университете в Беркли. С указанного момента *Caffe* активно поддерживается Центром Зрения и Обучения Беркли (*The Berkeley Vision and Learning Center*, *BVLC*) и сообществом разработчиков на *GitHub*. Библиотека распространяется под лицензией *BSD* 2-*Clause*.

Caffe реализована с использованием языка программирования *C++*, имеются обертки на *Python* и *MATLAB*. Официально поддерживаемые операционные системы – *Linux* и *OS X*, также имеется неофициальный порт на *Windows*. *Caffe* использует библиотеку *BLAS* (*ATLAS*, *Intel MKL*, *OpenBLAS*) для векторных и матричных вычислений. Наряду с этим в число внешних зависимостей входят *glog*, *gflags*, *OpenCV*, *protoBuf*, *boost*, *leveldb*, *nappy*, *hdf*5, *lmdb*. Для ускорения вычислений *Caffe* может быть запущена на *GPU* с использованием базовых возможностей технологии *CUDA* или библиотеки примитивов глубокого обучения *cuDNN*.

Разработчики *Caffe* поддерживают возможности создания, обучения и тестирования полностью связанных и сверточных нейросетей. Входные данные и преобразования описываются понятием слоя. В зависимости от формата хранения могут использоваться следующие типы слоев исходных данных:

– *DATA* – определяет слой данных в формате *leveldb* и *lmdb*;

– *HDF5_DATA* – слой данных в формате *hdf*5;

– *IMAGE_DATA* – простой формат, который предполагает, что в файле приведен список изображений с указанием метки класса.

Преобразования могут быть заданы с помощью слоев:

– *INNER_PRODUCT* – полностью связанный слой;

– *CONVOLUTION* – сверточный слой;

– *POOLING* – слой пространственного объединения;

– *Local Response Normalization* (*LRN*) – слой локальной нормализации.

Вместе с тем при формировании преобразований могут использоваться различные функции активации:

– положительная часть (*Rectified-Linear Unit*, *ReLU*);

– сигмоидальная функция (*SIGMOID*);

– гиперболический тангенс (*TANH*);

– абсолютное значение (*ABSVAL*);

– возведение в степень (*POWER*).

Функция биноминального нормального логарифмического правдоподобия (*binomial normal log likelihood*, *BNLL*).

Последний слой нейросетевой модели должен содержать функцию ошибки. В библиотеке имеются следующие функции:

– среднеквадратичная ошибка (*Mean-Square Error*, *MSE*);

– краевая ошибка (*Hinge loss*);

– логистическая функция ошибки (*Logistic loss*);

– функция прироста информации (*Info gain loss*);

– сигмоидальная кросс-энтропия (*Sigmoid cross entropy loss*);

– *Softmax*-функция. Обобщает сигмоидальную кросс-энтропию на случай количества классов больше двух.

В процессе обучения моделей применяются различные методы оптимизации. Разработчики *Caffe* предоставляют реализацию ряда методов:

– стохастический градиентный спуск (*Stochastic Gradient Descent*, *SGD*);

– алгоритм с адаптивной скоростью обучения (*Adaptive gradient learning rate algorithm*, *AdaGrad*);

– ускоренный градиентный спуск Нестерова (*Nesterov's Accelerated Gradient Descent*, *NAG*).

В библиотеке *Caffe* топология нейросетей, исходные данные и способ обучения задаются с помощью конфигурационных файлов в формате *prototxt*. Файл содержит описание входных данных (тренировочных и тестовых) и слоев нейронной сети.

4.2.7. Deeplearning4j

Deeplearning4j – библиотека программ на языке *Java*, используемая как фреймворк для глубокого обучения. Включает реализацию ограниченной машины Больцмана, глубокой сети доверия, глубокого автокодировщика, стекового автокодировщика с фильтрацией шума, рекурсивной тензорной нейронной сети, *word2vec*, *doc2vec*, *GloVe*. Эти алгоритмы включены также в версии библиотеки, поддерживающие распределенные вычисления, интегрированные с архитектурами *Apache Hadoop* и *Spark*.

Является открытым программным обеспечением, распространяется под лицензией *Apache* 2.0.

Deeplearning4j реализована на языке *Java* и выполняется в среде, при этом совместима с *Clojure* и включает интерфейс (*API*) для языка *Scala*. Дополнительная библиотека *ND4J* открытого доступа обеспечивает вычисления на графических процессорах с поддержкой *CUDA*. Кроме того, имеются средства для работы с библиотекой на языке *Python* через фреймворк *Keras*.

Фреймворк позволяет комбинировать компоненты, объединяя обычные нейронные сети с машинами Больцмана, сверточными нейронными сетями, автокодировщиками и рекуррентными сетями в одну систему. Также поддерживаются расширенные средства визуализации. Обучение проводится как с помощью обычных многослойных нейронных сетей, так и для сложных сетей, в которых определен граф вычислений.

Распределенное обучение в *Deeplearning4j* осуществляется через кластеры. Нейронные сети обучаются параллельно по итерациям, процесс поддерживается архитектурами *Hadoop-YARN* и *Spark*. *Deeplearning4j* осуществляет также интеграцию с ядром архитектуры *CUDA* для осуществления чистых операций с *GPU* и распределения операций на графических процессорах. Для обработки больших объемов текстов с использованием мощности параллельных графических процессоров *Deeplearning4j* привлекает инструментарий векторного и тематического моделирования на языке *Java*.

В библиотеку входят реализации частотной инверсии (*TF-IDF*), глубинное обучение, алгоритм Миколова *word2vec*, *doc2vec* и *GloVe*, которые оптимизированы на *Java*. При этом используется принцип стохастического встраивания соседей с распределением Стьюдента (*t-SNE*) для реализации облака слов.

Сопоставление производительности показывает, что *Deeplearning4j* сопоставим с *Caffe* в задачах нетривиального распознавания образов с привлечением параллельных графических процессоров.

ГЛАВА 5. НЕЙРОННЫЕ СЕТИ ГЛУБОКОГО ОБУЧЕНИЯ

5.1. Сверточные нейронные сети

5.1.1. Архитектура сверточной нейронной сети

Сверточная нейросеть (*ConvNet*/*CNN*) – это алгоритм глубокого обучения, способный выявлять самостоятельно скрытые признаки в исходных данных [18].

Сверточные нейронные сети состоят из нейронов и синапсов (рис. 5.1). В результате обучения следует определить веса синапсов и смещения на нейронах.

Последний слой, как правило, является полносвязным. На нем вводится функция потерь (например, *Softmax*), минимизируя которую находят значения весов и смещений (рис. 5.1).

Архитектура сверточных нейронных сетей явно предполагает получение на входе изображений, что позволяет учесть определенные свойства входных данных в самой архитектуре сети [18]. Эти свойства позволяют реализовать функцию прямого распространения эффективнее и сильно уменьшают общее количество параметров в сети. В частности, в отличие от обычных нейронных сетей слои в сверточной нейронной сети располагают нейроны в трех измерениях – ширине, высоте, глубине (слово «глубина» относится к третьему измерению активационных нейронов, а не глубине самой нейронной сети, измеряемой в количестве слоев).

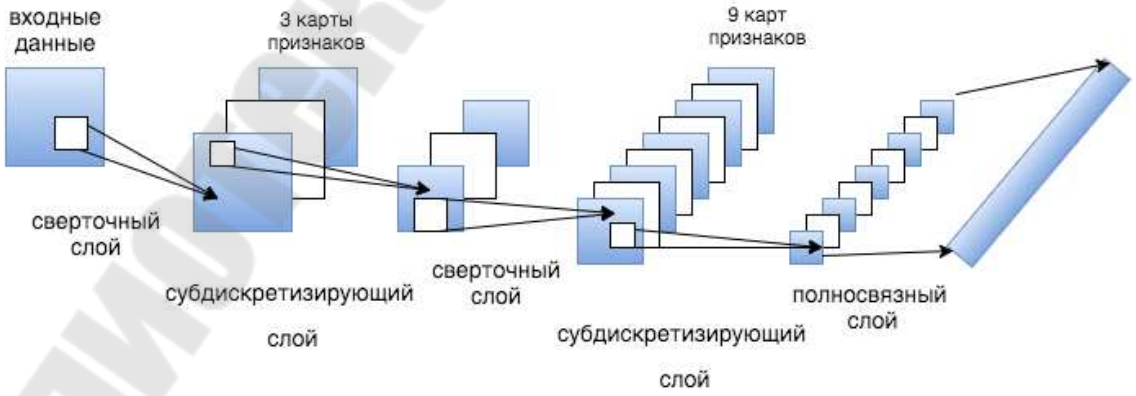


Рис. 5.1. Схема архитектуры сверточной нейронной сети

Каждый слой сверточной нейронной сети преобразует 3*D*-представление входных данных в 3*D*-представление выходных данных в виде нейронов активации.

Для изображений размеры входного слоя равны размерам изображения, а глубина будет равна трем (три канала – *red*, *green*, *blue*).

5.1.2. Виды слоев сверточных сетей

Входной слой (*input*) – преобразует исходное растровое изображение в 3*D*-представление входных данных (три слоя для каждой компоненты цвета: *red*, *green*, *blue*) (рис. 5.2).

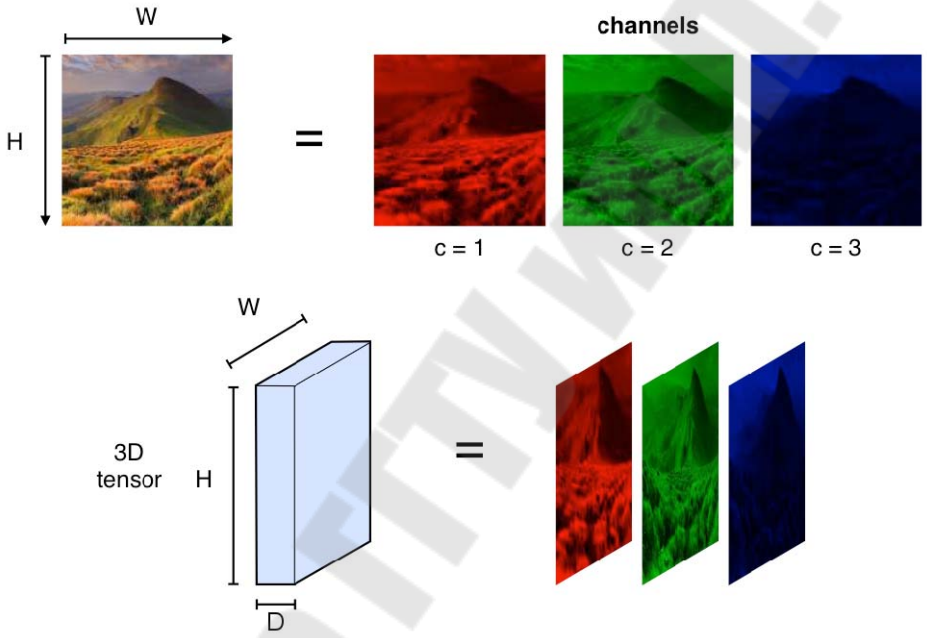


Рис. 5.2. Пример 3*D*-представления входных данных

***Слой подвыборки* (*pool*)** – данный слой позволяет уменьшить пространство признаков, сохраняя наиболее важную информацию. Существует несколько разных версий слоя пулинга, среди которых максимальный пулинг, средний пулинг и пулинг суммы. Слою подвыборки требуется всего один гиперпараметр – шаг пулинга, т. е. число раз, в которое нужно сократить пространственные размерности. Наиболее часто используется слой макспулинга с уменьшением размера входного тензора в два раза (рис. 5.3). Некоторые библиотеки позволяют задавать раздельные параметры уменьшения по высоте и ширине, однако чаще всего эти параметры совпадают [17].

Рис. 5.3. Схема слоя подвыборки

Исходные данные:
– представление размером $\boldsymbol{W_1 \times H_1 \times D_1}$;
– два параметра: размер рецептивного поля $\boldsymbol{F}$, шаг $\boldsymbol{S}$.
Результат – представление размером $\boldsymbol{W_2 \times H_2 \times D_2}$, где

$$W_2 = (W_1 - F)/S + 1; \;\; H_2 = (H_1 - F)/S + 1; \;\; D_2 = D_1. \qquad (5.1)$$

Слой активации – представляет из себя некоторую функцию, которая применяется к каждому числу входного изображения. Наиболее часто используются такие функции активации, как *ReLU*, *Sigmoid*, *Tanh*, *LeakyReLU*. Обычно активационный слой ставится сразу после слоя свертки, из-за чего некоторые библиотеки даже встраивают *ReLU* функцию прямо в сверточный слой.

***Полносвязный слой* (*FC – fully connected*)** – слой, в котором каждый нейрон соединен со всеми нейронами на предыдущем слое, причем каждая связь имеет свой вес, а каждый нейрон – свое смещение. Фактически данный слой ничем не отличается от такого же слоя в обыкновенной полносвязной сети. Единственным гиперпараметром слоя является количество выходных значений. При этом результатом применения слоя является вектор или тензор, у которого матрицы в каждом канале имеют размер 1 × 1.

Слой нормализации – используется для согласования слоев и может размещаться перед входом в каждый слой. Много типов слоев нормализации было предложено для использования в сверточных нейронных сетях, иногда намеренно применяя такие архитектурные решения, которые наблюдаются в природе и биологическом мозге. Однако эти слои со временем перестали использоваться, так как на практике выяснилась их минимальная эффективность и влияние на результат.

Bottleneck-слой – уменьшает количество свойств (а значит и операций) в каждом слое, так что скорость получения результата можно сохранить на высоком уровне. Прежде чем передавать данные в «дорогие» сверточные модули, количество свойств уменьшается, предположим, в четыре раза. Это сильно сокращает объем вычислений.

***Dropout-слой* (*dropout* регуляризация)** – временно исключает из обучения часть нейронов последующего слоя. Используется для борьбы с переобучением в нейронных сетях. Главная идея *Dropout* – вместо обучения одной нейронной сети обучить ансамбль нескольких сетей, а затем усреднить полученные результаты (рис. 5.4).

Сети для обучения получаются с помощью ***исключения*** из сети (*dropping out*) нейронов с вероятностью p, таким образом, вероятность того, что нейрон останется в сети, составляет $q = 1 - p$. «Исключение» нейрона означает, что при любых входных данных или параметрах он возвращает 0.

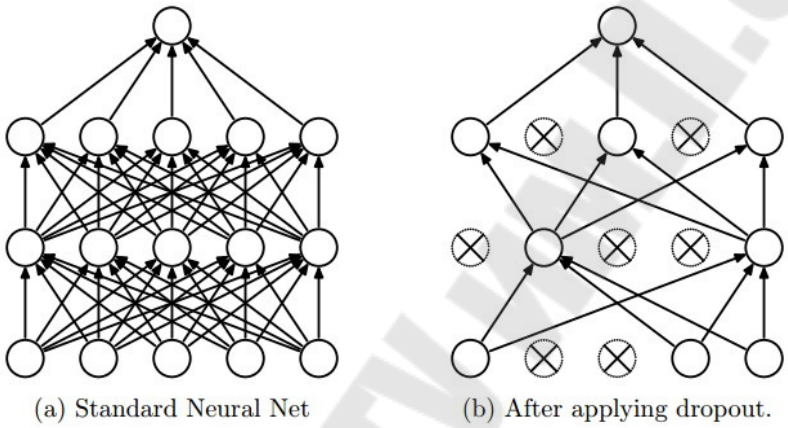


Рис. 5.4. Схема слоя *Dropout*

Слой свертки – в сверточном слое нейрон соединен с ограниченным числом нейронов предыдущего слоя, т. е. сверточный слой аналогичен применению операции свертки, где используется матрица весов небольшого размера (ядро свертки или фильтр), которая перемещается с заданным шагом по всему слою. В итоге получается матрица меньшей размерности (рис. 5.5).

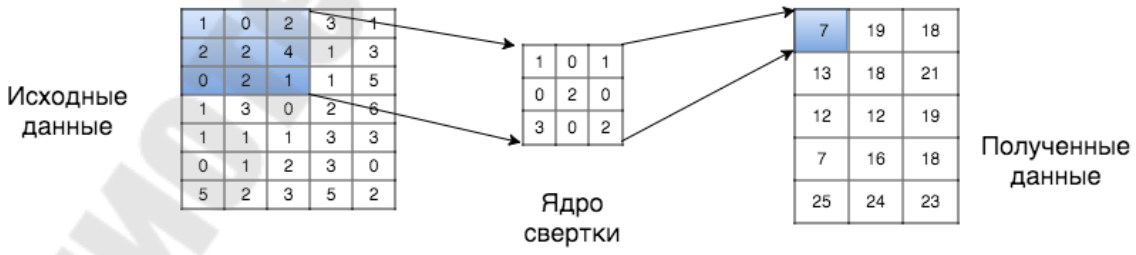


Рис. 5.5. Схема слоя свертки

Такая процедура применяется ко всем слоям исходных данных (*red*, *green*, *blue*). В итоге будет получено три матрицы, значения элементов которых суммируются в единую матрицу. После этого к каж-

дому значению матрицы добавляется одинаковое число – значение смещения данного ядра (фильтра). Полученная матрица составляет один канал выходной карты признаков. После того, как будут получены каналы для каждого из ядер, матрицы объединяются в единый тензор, благодаря чему на выходе снова получается изображение с другим числом каналов и другим размером.

Параметры сверточного слоя

***Число признаков* (*filters count*, *fc*)** – это количество фильтров, которые есть в слое.

***Размер фильтров* (*filter size*, *fs*)** – это высота и ширина тензора фильтров. Обычно является нечетным числом, наиболее часто используются фильтры размером 3 или 5.

***Шаг свертки* (*stride*, *S*)** – это количество пикселей, на которое перемещается матрица фильтра по входному изображению. Когда шаг равен 1, фильтры перемещаются по одному пикселю за раз. Когда шаг равен 2, тогда фильтры перескакивают на два пикселя за раз. Чем больше шаг, тем меньшего размера карты признаков получаются на выходе.

***Дополнения нулями* (*padding*, *P*)** – количество пикселей, которые добавляются с каждого края изображения. Это позволяет избежать уменьшения изображения на размер фильтра, поскольку фильтр может накладываться лишь в тех местах, в которых под каждым значением фильтра будет значение входного изображения.

Таким образом, входными параметрами сверточного слоя являются:

– тензор размером $W_1 \times H_1 \times D_1$;

– четыре гиперпараметра: fc, fs, S, P.

Выходным параметром слоя является тензор размером $W_2 \times H_2 \times D_2$, где

$$W_2 = (W_1 - fs + 2P)/S + 1, \quad H_2 = (H_1 - fs + 2P)/S + 1, \quad D_2 = fc. \qquad (5.2)$$

5.1.3. Преобразование полностью связанных слоев в сверточные слои

Единственной разницей между полносвязным слоем и сверточным слоем является то, что нейроны в сверточном слое связаны только с локальной областью входного представления и совместно используют параметры (веса и смещения).

Однако нейроны в обоих слоях по-прежнему вычисляют скалярное произведение, а это значит, что их функциональная структура

идентична. Таким образом становится очевидно, что возможно преобразование между полносвязным слоем и сверточным слоем:

– для любого сверточного слоя существует полносвязный слой, который реализует ту же функцию прямого распространения. Матрица весов, по большей части, будет заполнена нулями, кроме той области, которая является отображением локальных значений входного представления и значения в которой будут одинаковы в связи с тем фактом, что слои совместно используют веса;

– любой полносвязный слой может быть преобразован в сверточный слой. Например, полносвязный слой с $K = 4096$ (количество фильтров), находящийся над областью размером $7 \times 7 \times 12$, может быть эквивалентно представлен в виде сверточного слоя с гиперпараметрами $F = 7$, $P = 0$, $S = 1$, $K = 4096$. Другими словами, размеры фильтра устанавливаются таким образом, чтобы он соответствовал размерам входного представления, и в таком случае выходное представление будет размерами $1 \times 1 \times 4096$, давая идентичный результат начального полносвязного слоя.

5.1.4. Шаблоны слоев

Входной слой (содержащий изображение) должен делиться на 2 множество раз. Стандартные значения включают 32 (например, *CIFAR*-10), 64, 96 (например, *STL*-10) или 224 (например, *ImageNet*), 384 и 512.

В ***сверточном слое*** рекомендуется использовать фильтры небольшого размера (например, 3×3 или, максимум, 5×5), с использованием шага $\boldsymbol{S = 1}$ и, что не менее важно, размером выравнивания таким образом, чтобы сверточный слой не изменял пространственные размеры входного представления. Таким образом, $\boldsymbol{F = 3}$ и при $P = 1$ будет возможность сохранить исходные пространственные размеры входного представления. При $F = 5$ $P = 2$. Для обобщенного размера F может быть определено, что при $P = (F – 1)/2$ размеры исходного представления будут сохранены. Если по каким-то причинам необходимо использовать фильтры большего размера (такие как 7×7), т. е. смысл использовать их только на первом слое сверточной нейронной сети при работе с исходным изображением.

Слой подвыборки отвечает за снижение размерности входного представления. Наиболее частой конфигурацией является использование слоя подвыборки по максимальному значению размером 2×2 ($F = 2$) и шагом 2 ($S = 2$). Нужно учесть, что подобная конфигурация

исключает 75 % активаций входного представления (из-за сэмплирования в два раза по ширине и высоте). Другой, менее распространенной конфигурацией, является конфигурация с параметрами 3×3 (размер фильтра) и 2 (размер шага). Изредка можно увидеть размеры фильтра больше 3×3 для слоя подвыборки, потому что подвыборка представляет собой очень агрессивную функцию, при применении которой теряется информация по признакам. Такой подход обычно приводит к плохой производительности.

Представленные выше рекомендации выглядят удобными для работы, потому что все сверточные слои сохраняют пространственные размеры входных представлений, а за уменьшение размеров представления отвечает слой подвыборки. При альтернативных значениях, где мы используем шаг больше 1 или не выполняем операцию выравнивания входных представлений, мы должны очень внимательно следить за размером входных представлений по мере прохождения через слои и убедиться в том, что все работает как надо и архитектура сверточной нейронной сети выровнена и симметрична.

Почему используют размер шага равный 1 в сверточном слое? На практике меньшего размера шаги работают лучше. Как мы отмечали выше, размер шага равный 1 позволяет делегировать сэмплирование входного представления слою подвыборки (изменение размеров входного представления по ширине и высоте), сохраняя за сверточным слоем только преобразование в глубину.

Зачем использовать выравнивание? В дополнение к приведенному выше аргументу по сохранению размерности выходного представления после прохождения через сверточный слой выравнивание позволяет повысить производительность. Если бы сверточные слои не использовали выравнивание входных представлений, а лишь выполняли операцию свертки, тогда размер выходных представлений уменьшался бы каждый раз на небольшое количество пикселей, а информация на краях входного представления исчезала бы слишком быстро.

Компромиссы, основанные на ограничениях памяти. В некоторых ситуациях (особенно на ранних стадиях развития архитектур сверточных нейронных сетей) количество требуемой памяти может очень быстро увеличиваться, если следовать всем рекомендациям выше. Например, применяя три сверточных слоя по 64 фильтра каждый с размером 3×3 и шагом 1 к изображению $224 \times 224 \times 3$, создается на выходе активационное представление размером $224 \times 224 \times 64$. Это примерно равно 10 млн активаций, или 72 Мб памяти (на изображение как для активаций, так и для градиентов). Так как «бутылочным

горлышком» в процессе вычислений является размер доступной памяти *GPU*, то приходится идти на компромиссы. На практике одним из компромиссов может быть использование первого сверточного слоя с размером фильтра 7 × 7 и шагом 2. Другой компромисс, как это сделано в *AlexNet*, – использовать фильтр размером 11 × 11 и шагом 4.

5.1.5. Примеры архитектур сверточных нейронных сетей

***LeNet*5.** В 1994 г. была разработана одна из первых сверточных нейросетей, положившая начало глубокому обучению. Эта пионерская работа Яна Лекуна (*Yann LeCun*) после многих успешных итераций, начиная с 1988 г., получила название *LeNet*5.

Архитектура *LeNet*5 стала фундаментальной для глубокого обучения, особенно с точки зрения распределения свойств изображения по всей картинке. Свертки с обучаемыми параметрами позволяли с помощью нескольких параметров эффективно извлекать одинаковые свойства из разных мест. В *LeNet*5 в первом слое пиксели не используются, потому что изображения сильно коррелированы пространственно, так что использование отдельных пикселей в качестве входных свойств не позволит воспользоваться преимуществами этих корреляций (рис. 5.6).

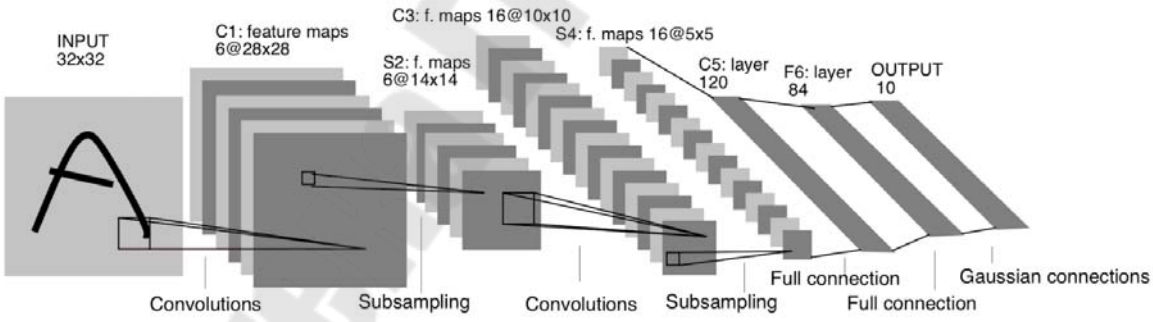


Рис. 5.6. Архитектура модели *LeNet*5

По классической схеме на вход этой сети подается изображение размером $32 \times 32 \times 1$ пикселя. Изображение черно-белое, поэтому слой всего один, однако никто не мешает использовать цветное изображение с тремя слоями.

Стоит уточнить, что за прошедшие годы терминология более или менее устоялась, и то, что на схеме называется слоем *Subsampling*, в современной терминологии чаще всего зовется *Pooling*.

Также стоит обратить внимание на то, что на схеме каждый *convolutional* и *pooling* слои считаются отдельными слоями, однако есть немало приверженцев иного подхода, при котором *convolutional* + + *pooling* слои вместе считаются частью одного слоя *CNN*.

Особенности *LeNet*5 (приложение 1):

– сверточная нейросеть, использующая последовательность из трех слоев: слои свертки (*convolution*), слои группирования (*pooling*) и слои нелинейности (*non-linearity*) → с момента публикации работы Лекуна это, пожалуй, одна из главных особенностей глубокого обучения применительно к изображениям;

– использует свертку для извлечения пространственных свойств;

– подвыборка с использованием пространственного усреднения карт;

– нелинейность в виде гиперболического тангенса или сигмоид;

– финальный классификатор в виде многослойной нейросети (*MLP*);

– разреженная матрица связности между слоями позволяет уменьшить объем вычислений.

AlexNet. В 2012 г. Алексей Крижевский опубликовал *AlexNet* (приложение 2), углубленную и расширенную версию *LeNet*, которая с большим отрывом победила в сложном соревновании *ImageNet* [20]. Ее архитектура состоит из пяти сверточных слоев, между которыми располагаются *pooling*-слои и слои нормализации, а завершают нейросеть три полносвязных слоя (рис. 5.7).

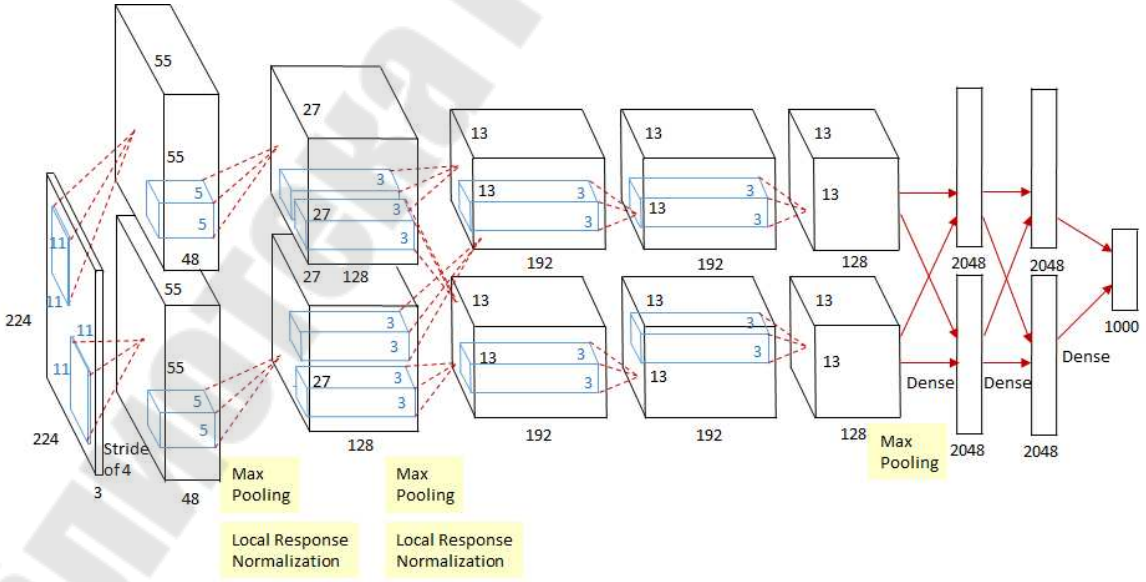


Рис. 5.7. Архитектура модели *AlexNet*

На схеме архитектуры все выходные изображения делятся на два одинаковых участка – это связано с тем, что нейросеть обучалась на старых *GPU GTX*580, у которых было всего 3 ГБ видеопамяти. Для обработки использовались две видеокарты, чтобы параллельно выполнять операции над двумя частями изображения.

AlexNet содержит пять сверточных слоев и три полносвязных слоя. *Relu* применяется после каждого сверточного и полносвязного слоя. Дропаут применяется перед первым и вторым полносвязными слоями. Сеть содержит 62,3 млн параметров и затрачивает 1,1 млрд вычислений при прямом проходе. Сверточные слои, на которые приходится 6 % всех параметров, производят 95 % вычислений.

Изначально на вход подается фотография размером 227 × 227 × 3, и размер сверточных фильтров первого слоя – 11 × 11. Всего применяется 96 фильтров с шагом 4.

Особенности *AlexNet*:

– как функция активации используется *Relu* вместо арктангенса для добавления в модель нелинейности. За счет этого при одинаковой точности метода скорость становится в шесть раз быстрее;

– использование дропаута вместо регуляризации решает проблему переобучения. Однако время обучения удваивается с показателем дропаута 0,5;

– производится перекрытие объединений для уменьшения размера сети. За счет этого уровень ошибок первого и пятого уровней снижаются до 0,4 и 0,3 % соответственно.

VGGNet. *VGG*16 (приложение 3) – модель сверточной нейронной сети, предложенная *K. Simonyan* и *A. Zisserman* из Оксфордского университета. *VGG*16 является улучшенной версией *AlexNet* (рис. 5.8). Ее особенностью является применение в каждом сверточном слое фильтров 3 × 3 и их объединение в последовательности сверток, что впоследствии фактически стало стандартом. Большим преимуществом *VGG* стала находка, что несколько сверток 3 × 3, объединенных в последовательность, могут эмулировать более крупные рецептивные поля, например, 5 × 5 или 7 × 7. Эти идеи позднее будут использованы в архитектурах *Inception* и *ResNet*.

Сети *VGG* для представления сложных свойств используют многочисленные сверточные слои 3 × 3. Для извлечения более сложных свойств и их комбинирования применяются последовательности фильтров 3 × 3, что равносильно большим сверточным классификаторам. Это дает огромное количество параметров и прекрасные способ-

ности к обучению. Но учить такие сети было сложно, приходится разбивать их на более мелкие, добавляя слои один за другим. Причина заключается в отсутствии эффективных способов регуляризации моделей или каких-то методов ограничения большого пространства поиска, которому способствует множество параметров.

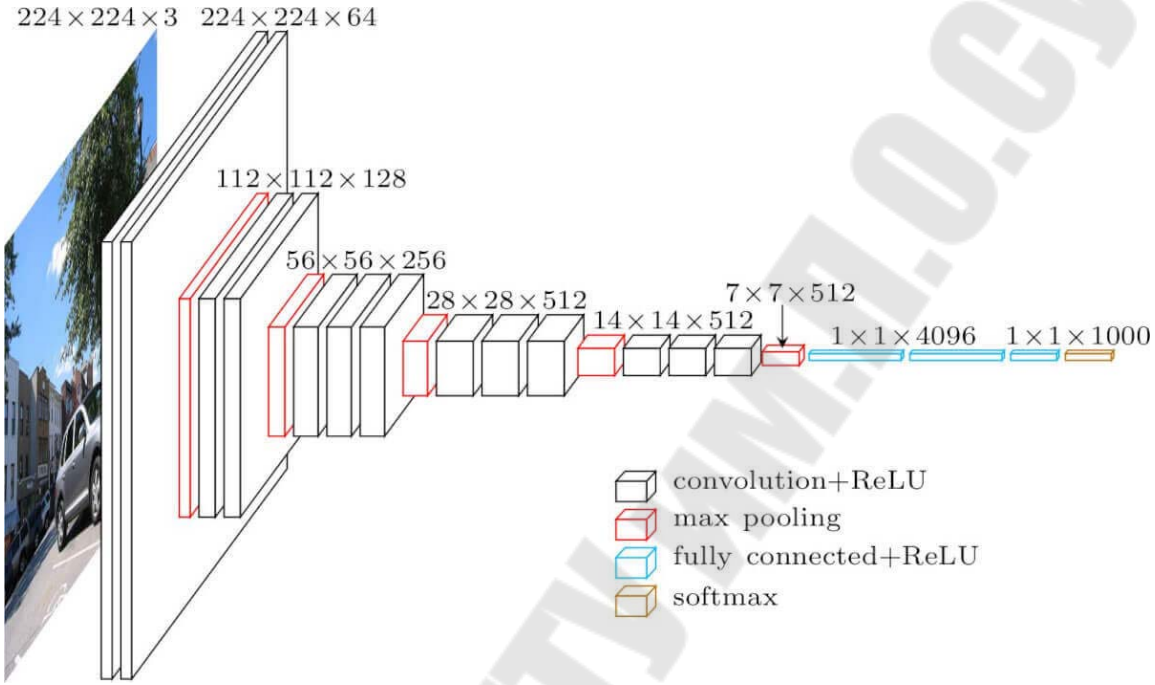


Рис. 5.8. Архитектура модели *VGG*16

Особенности *VGGNet*:

– нейросеть заняла второе место в задаче классификации и первое место в локализации на соревновании *ImageNet* (при локализации необходимо не только классифицировать объект, но и обвести его в ограничивающие рамки);

– процедура обучения такая же, как у *AlexNet*;

– слои нормализации отсутствуют;

– *VGG* имеет два серьезных недостатка:

1) очень медленная скорость обучения;

2) сама архитектура сети весит слишком много (появляются проблемы с диском и пропускной способностью).

Сравнение архитектуры *VGG* и *AlexNet* приведено на рис. 5.9.

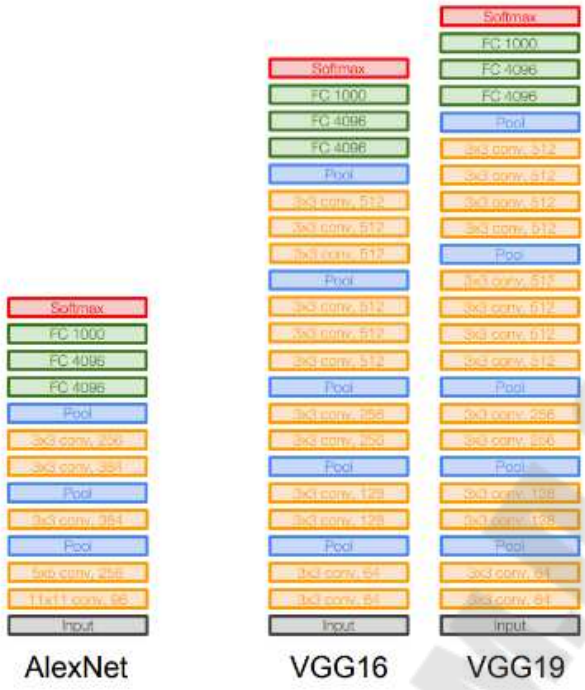


Рис. 5.9. Сравнение архитектуры *VGG* и *AlexNet*

Network-in-Network. В основе архитектуры *Network-in-Network* (*NiN*) лежит простая идея: использование сверток 1 × 1 для увеличения комбинаторности свойств в сверточных слоях.

В *NiN* после каждой свертки применяются пространственные *MLP*-слои, чтобы лучше скомбинировать свойства перед подачей в следующий слой (рис. 5.10).

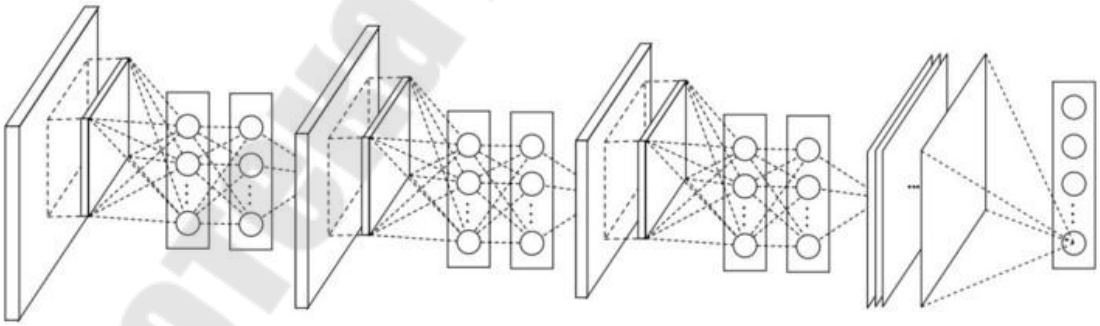


Рис. 5.10. Архитектура *Network-in-Network*

Может показаться, что использование сверток 1 × 1 противоречит исходным принципам *LeNet*, но на самом деле это позволяет комбинировать свойства лучше, чем просто набивая больше сверточных слоев. Этот подход отличается от использования голых пикселей в качестве входных данных для следующего слоя.

В данном случае свертки 1 × 1 применяются для пространственного комбинирования свойств после свертки в рамках карт свойств, так что можно использовать гораздо меньше параметров, которые являются общими для всех пикселей этих свойств.

MLP позволяют сильно повысить эффективность отдельных сверточных слоев посредством их комбинирования в более сложные группы. Эта идея позднее была использована в других архитектурах, таких как *ResNet*, *Inception* и их вариантах.

GoogleNet. Кристиан Жегеди (*Christian Szegedy*) из *Google* озаботился снижением объема вычислений в глубоких нейросетях. Для этого был предложен так называемый модуль *Inception* – вся архитектура состоит из множества таких модулей, следующих друг за другом (рис. 5.11).

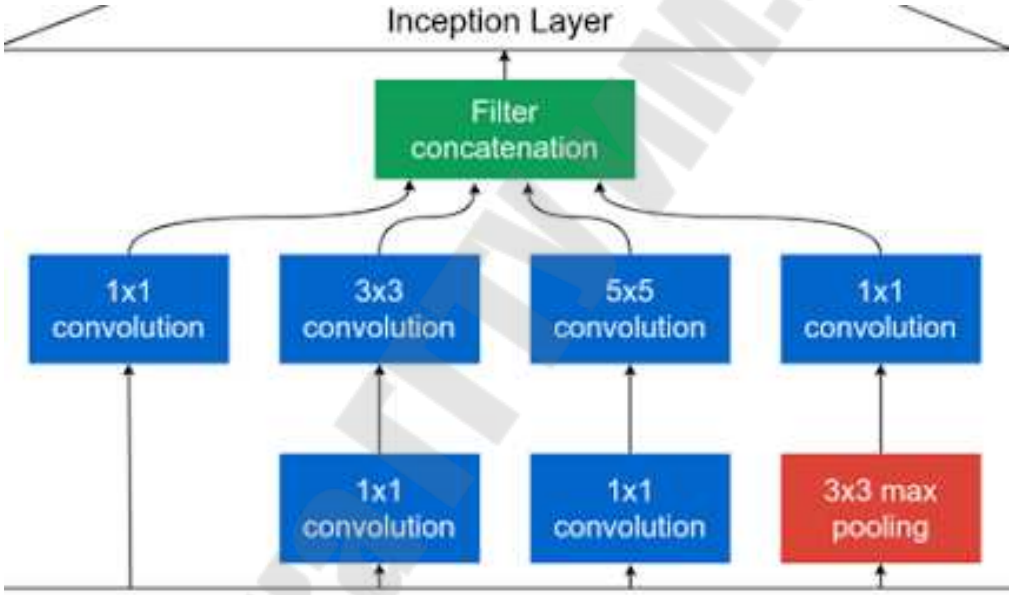


Рис. 5.11. Архитектура *GoogleNet*

Идея основного модуля *Inception* заключается в том, что он сам по себе является небольшой локальной сетью. Вся его работа состоит в параллельном применении нескольких фильтров на исходное изображение. Данные фильтров объединяются, и создается выходной сигнал, который переходит на следующий слой.

В *GoogleNet* в качестве начального слоя применяется *stem* без *Inception*-модулей – *Stem Network* (рис. 5.12). Она состоит из трех сверточных слоев с двумя *pooling*-слоями и *softmax*-классификатором, аналогичным *NiN*.

В *GoogleNet* нет полносвязных слоев, и она содержит всего 5 млн параметров – в 12 раз меньше, чем у *AlexNet*.

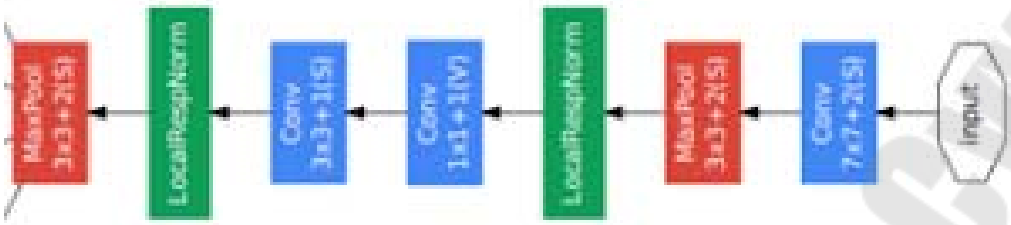


Рис. 5.12. Схема *Stem Network*

Inception *v*3 (*v*2). В 2015 г. была предложена пакетная нормализация (*batch-normalization*), которая вычисляла среднее и среднеквадратичное отклонение всех карт распределения свойств в выходном слое, и нормализовала их отклики с этими значениями. Это соответствует «отбеливанию» данных, т. е. отклики всех нейронных карт лежат в одном диапазоне и с нулевым средним. Такой подход облегчает обучение, потому что последующий слой не обязан запоминать смещения (*offsets*) входных данных и может заниматься только поиском лучших комбинаций свойств.

Основные идеи:

– максимизация потока информации в сети за счет аккуратного баланса между ее глубиной и шириной. Перед каждым *pooling*-ом увеличиваются карты свойств;

– с увеличением глубины также систематически увеличивается количество свойств или ширина слоя;

– ширина каждого слоя увеличивается ради увеличения комбинации свойств перед следующим слоем;

– по мере возможности используются только свертки 3 × 3, учитывая, что фильтры 5 × 5 и 7 × 7 можно декомпозировать с помощью нескольких 3 × 3;

– фильтры можно декомпозировать с помощью сглаженных сверток в более сложные модули.

Inception-модули могут с помощью *pooling*-а в ходе *Inception*-вычислений уменьшать размер данных. Это аналогично выполнению свертки со страйдами параллельно с простым *pooling*-слоем. В качестве финального классификатора *Inception* использует *pooling*-слой с *softmax* (рис. 5.13).

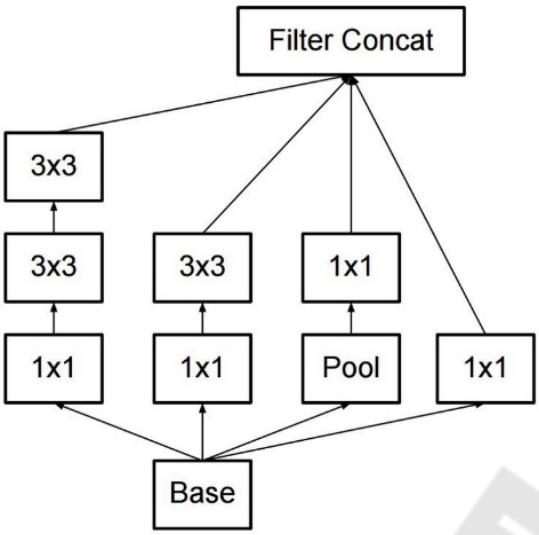


Рис. 5.13. Новый *Inception*-модуль

ResNet. В 2015 г. *ResNet* произвела настоящую революцию глубины нейросетей. Она состояла из 152 слоев и снизила процент ошибок до 3,57 % в соревновании классификации *ImageNet*. Это сделало ее почти в два раза эффективнее *GoogleNet*.

Увеличение слоев в нейронной сети не приводит к улучшению качества классификации нейронной сети. Создатели *ResNet* предположили, что загвоздка кроется в оптимизации – более глубокие модели гораздо хуже поддаются настройке. Тогда они решили не складывать слои друг на друга для изучения отображения нужной функции напрямую, а использовать остаточные блоки, которые пытаются «подогнать» это отображение. Так *ResNet* стала первой остаточной нейронной сетью.

ResNet «перепрыгивает» через некоторые слои. Они больше не содержат признаков и используются для нахождения остаточной функции $H(x) = F(x) + x$ вместо того, чтобы искать $H(x)$ напрямую. Выходные данные двух успешных сверточных слоев подаются на вход, обходя входные данные для следующего слоя. Фактически это сеть в сети, которая реализует «небольшой» классификатор (рис. 5.14).

Inception v4 и Inception-ResNet. В силу того, что некоторые модули предыдущих версий *Inception* были более сложные, чем необходимо, и при этом требовалось добиться большего единообразия, авторами были предложены новые модели *Inception*.

В *Inception v*4 был изменен *stem*-слой. Основная идея заключается в наборе операций, выполняемых перед введением начальных блоков (рис. 5.15).

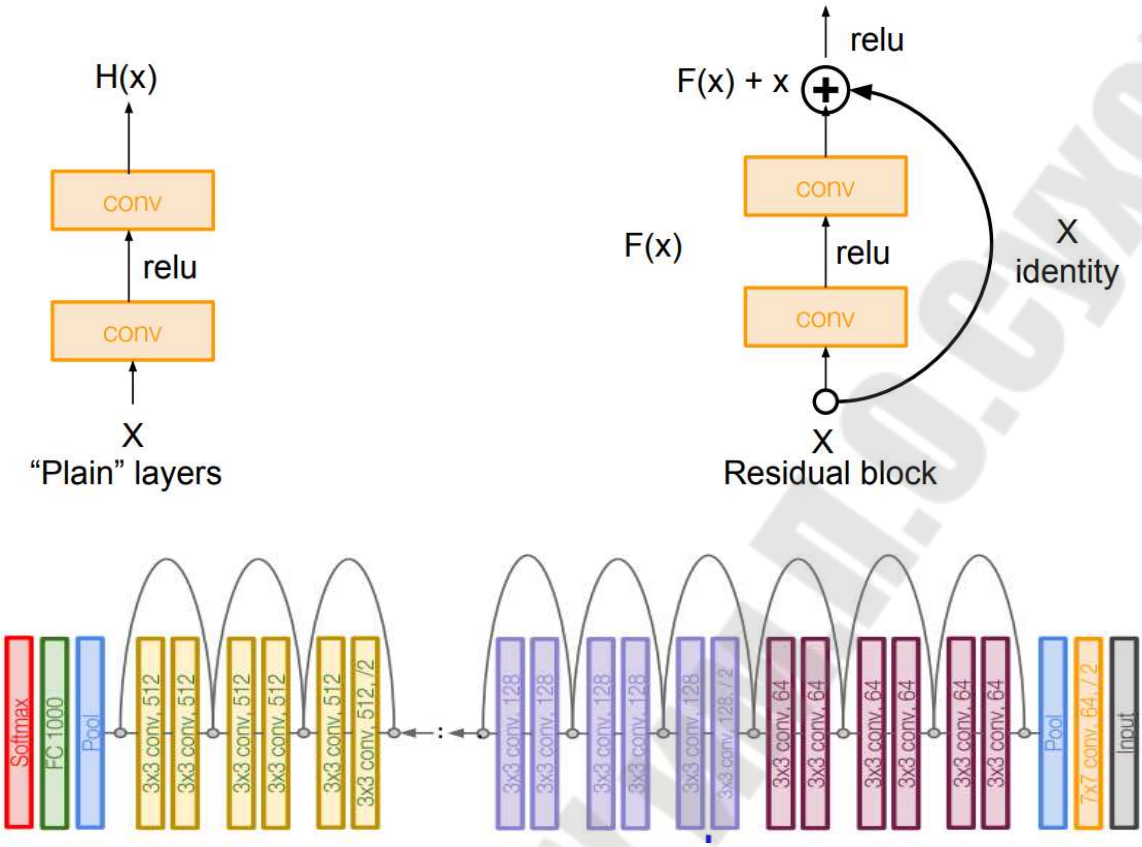


Рис. 5.14. Схема модели *ResNet*

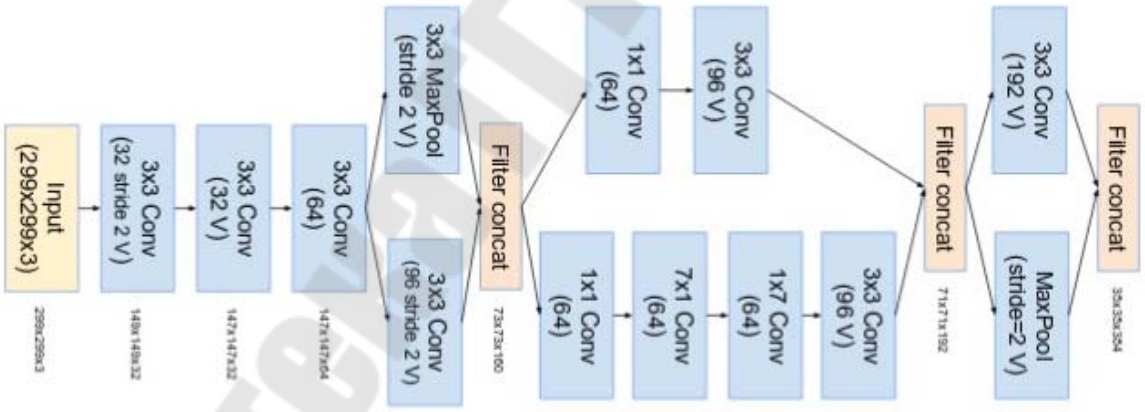


Рис. 5.15. Схема модели *Inception v*4

Было предложено три основных начальных модуля, названных *A*, *B* и *C*. Они очень похожи на свои аналоги в *Inception v*2 (или *v*3), только скомбинированы с *ResNet*-модулем (рис. 5.16).

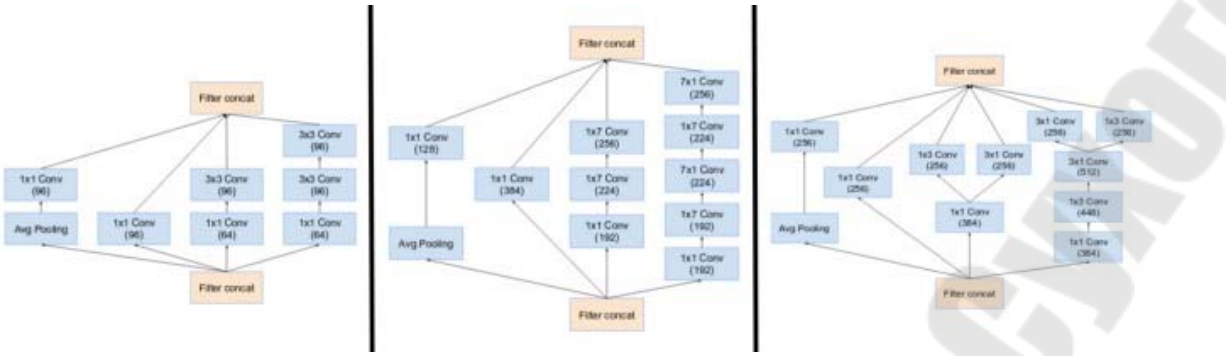


Рис. 5.16. Начальные модули *Inception v*4

В *Inception v*4 были реализованы специализированные «Редукционные блоки», которые используются для изменения ширины и высоты сетки. В более ранних версиях явно не было блоков сокращения, но подобная функциональность была также реализована.

ENet. *ENet* является попыткой скомбинировать особенности известных архитектур с целью построения компактной сети, использующей совсем мало параметров и вычислительных мощностей, но при этом дающую превосходные результаты.

Архитектура предложена Адамом Пазке (*Adam Paszke*). Основная идея – использовать кодировщик и декодер. Кодировщик построен по обычной схеме *CNN* для категоризации, а декодер представляет собой сеть с повышением дискретизации (*upsampling netowrk*), предназначенную для сегментирования посредством распространения категорий обратно в изображение исходного размера. Для сегментации изображений использовались только нейросети, никаких других алгоритмов.

Инициирующий блок *ENet* имеет вид, представленный на рис. 5.17.

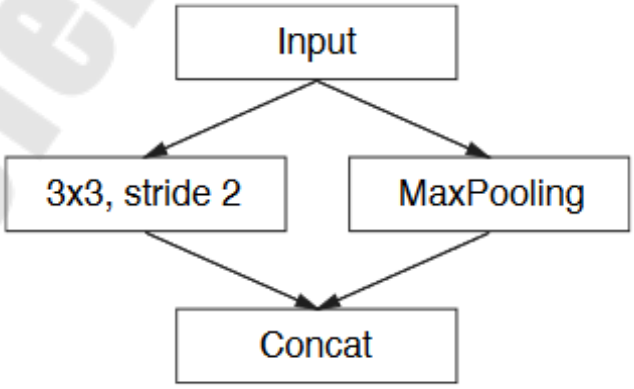


Рис. 5.17. Инициирующий блок *ENet*

Редукционный блок показан на рис. 5.18.

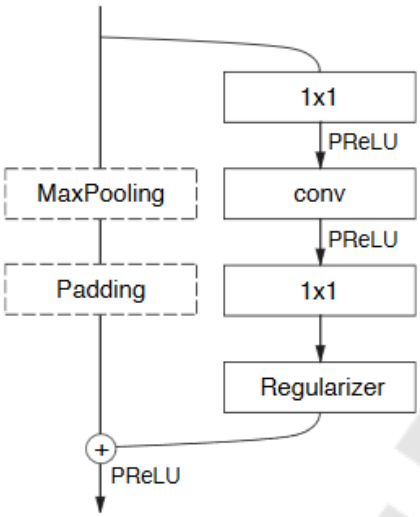


Рис. 5.18. Редукционный блок

ENet создавалась из расчета, чтобы с самого начала использовать как можно меньше ресурсов. В результате кодировщик и декодер вместе занимают всего 0,7 Мб с точностью *fp*16. И при таком крохотном размере *ENet* по точности сегментирования не уступает или превосходит прочие чисто нейросетевые решения.

ResNeXt. *ResNeXt* – архитектура, предложенная авторами *ResNet* и предлагающая «умное» расширение *Res*-блока. Основная идея заключается в разложении числа каналов в *Res*-блоке на несколько параллельных потоков, например 32 блока по 4 канала = 128 каналов вместо оригинальных 64 из обычного *Res*-блока (рис. 5.19).

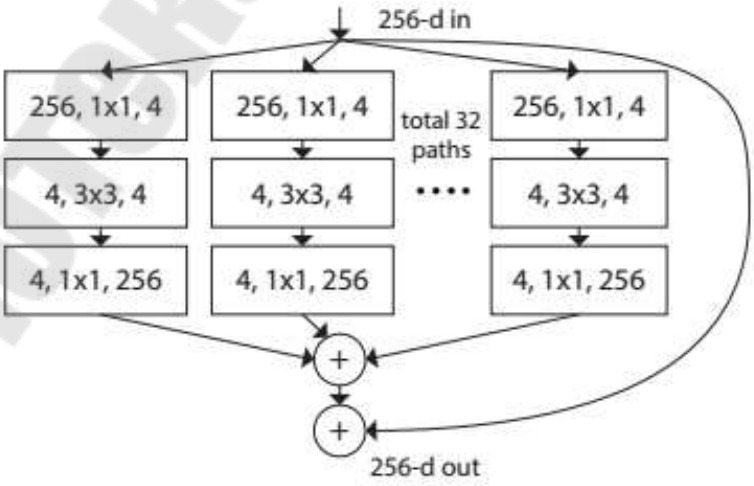


Рис. 5.19. Блок сети *ResNeXt*

Такой подход похож на *Inception*, но все параллельные блоки одинаковы. При той же сложности модели, что и у *ResNet*, получаем улучшение качества.

EfficientNets и FixEfficientNet. В данных сетях предлагается метод, основанный на некоторой оптимизации структуры сверточной нейронной сети (рис. 5.20). Предыдущие подходы произвольно масштабировали размерность нейросети (например, количество слоев и параметров). Предложенный метод равномерно масштабирует части нейросети с фиксированными коэффициентами масштабирования. Оптимизированные сети (*EfficientNets*) обходят *state-of-the-art* подходы по точности при увеличении эффективности в 10 раз (меньше и быстрее).

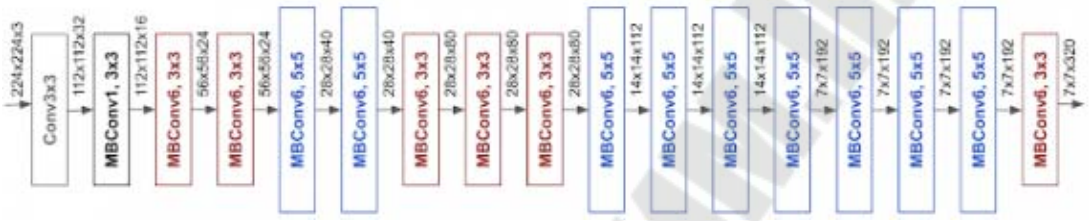


Рис. 5.20. Архитектура *EfficientNets-B*0

При обучении нейронных сетей для классификации изображений часто используется аугментация данных. Было замечено (*Hugo Touvron* и др.), что это приводит к значительному расхождению между размерами объектов, видимых классификатором во время обучения и во время тестирования: фактически более низкое разрешение обучающей выборки улучшает классификацию во время тестирования. В сетях *FixEfficientNet* предлагается простая стратегия для оптимизации производительности классификатора, которая использует различные разрешения для обучения и тестирования.

Данная стратегия основана на «дешевой» настройке сети в тестовом разрешении. Это позволяет тренировать сильные классификаторы с использованием небольших тренировочных наборов и, следовательно, значительно сократить время обучения.

5.2. Рекуррентные и рекурсивные сети

5.2.1. Архитектура рекуррентных нейронных сетей (*RNN*)

Рекуррентные нейронные сети (*Recurrent Neural Networks*, *RNNs*) – это сети, содержащие обратные связи и позволяющие сохранять информацию.

В рекуррентных нейросетях нейроны обмениваются информацией между собой: например, вдобавок к новому кусочку входящих данных нейрон также получает некоторую информацию о предыдущем состоянии сети. Таким образом в сети реализуется «память», что принципиально меняет характер ее работы и позволяет анализировать любые последовательности данных, в которых важно, в каком порядке идут значения [11].

Благодаря этому появляется возможность обрабатывать серии событий во времени или последовательные пространственные цепочки. В отличие от многослойных персептронов, рекуррентные сети могут использовать свою внутреннюю память для обработки последовательностей произвольной длины. Поэтому сети *RNN* применимы в таких задачах, где нечто целостное разбито на части (рис. 5.21).

Теоретически *RNN* могут использовать информацию в произвольно длинных последовательностях, но на практике они ограничены лишь несколькими шагами [21].

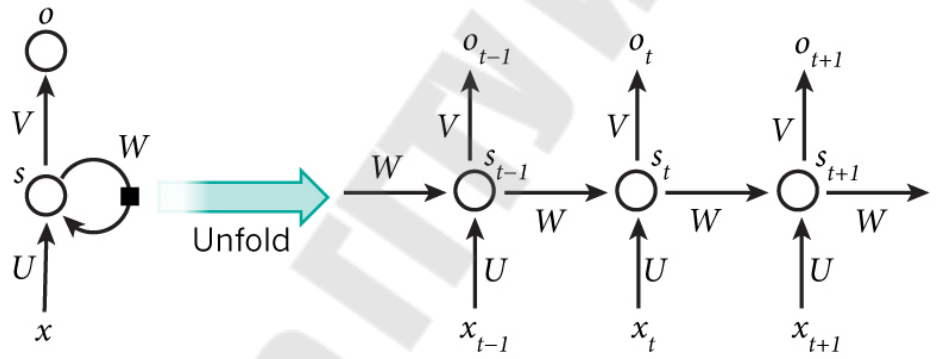


Рис. 5.21. Схема работы «памяти» рекурентной сети

На рис. 5.21 показано, как *RNN* разворачивается в полную сеть. Развернуто выписываем сеть для полной последовательности. Например, если последовательность представляет собой предложение из пяти слов, развертка будет состоять из пяти слоев, по слою на каждое слово. Формулы, задающие вычисления в *RNN*, следующие:

– x_t – вход на временном шаге t. Например x_1 может быть вектором с одним горячим состоянием (*one-hot vector*), соответствующим второму слову предложения;

– s_t – это скрытое состояние на шаге t. Это «память» сети. S_t зависит, как функция, от предыдущих состояний и текущего входа x_t: $s_t = f(Ux_t + Ws_{t-1})$. Функция f обычно нелинейная, например *tanh*

или *ReLU*. s_{-1}, которое требуется для вычисления первого скрытого состояния, обычно инициализируется нулем (нулевым вектором);

– o_t – выход на шаге *t*. Например, если мы хотим предсказать слово в предложении, выход может быть вектором вероятностей в нашем словаре. $o_t = softmax(Vs_t)$.

Можно интерпретировать s_t как память сети. s_t содержит информацию о том, что произошло на предыдущих шагах времени. Выход o_t вычисляется исключительно на основе «памяти» s_t.

В отличие от традиционной глубокой нейронной сети, которая использует разные параметры на каждом слое, *RNN* имеет одинаковые (*U*, *V*, *W*) на всех этапах. Это отражает тот факт, что выполняется одна и та же операция на каждом шаге, используя только разные входы. Это значительно уменьшает общее количество параметров, которые нам нужно подобрать.

Диаграмма выше имеет выходы на каждом шаге, но, в зависимости от задачи, они могут не потребоваться. Например при определении эмоциональной окраски предложения целесообразно заботиться только о конечном результате, а не об окраске после каждого слова. Аналогично, может не потребоваться ввод данных на каждом шаге. Основной особенностью *RNN* является скрытое состояние, которое содержит некоторую информацию о последовательности.

Слой *RNN* циклически обрабатывает упорядоченную (например, по времени) входную последовательность, храня при этом во внутреннем состоянии закодированную информацию о шагах, которые он уже видел.

Таким образом можно выделить типичный элемент рекуррентной нейронной сети (рис. 5.22).

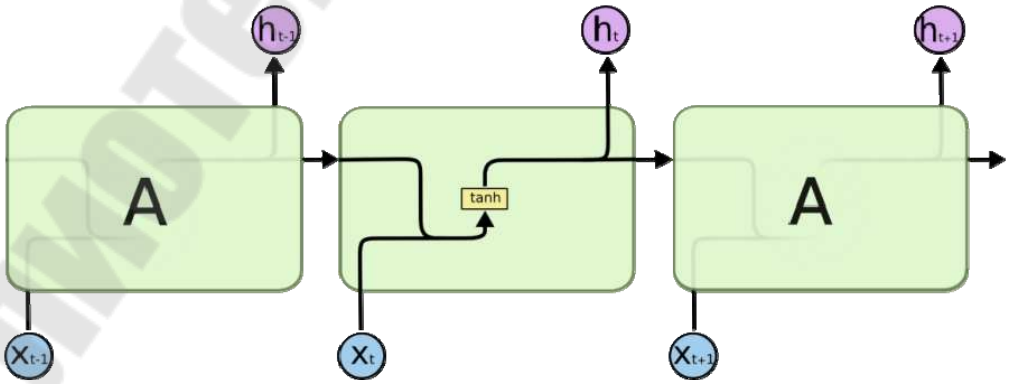


Рис. 5.22. Схема элемента рекурентной сети

RNN делят на несколько разновидностей: «*one to one*», «*one to many*», «*many to one*» и «*many to many*» (рис. 5.23).

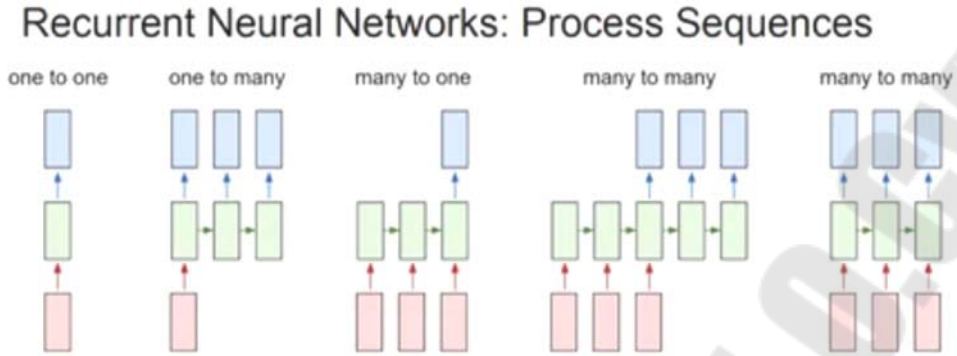


Рис. 5.23. Схемы разновидностей рекурентных сетей

К архитектурам «*one to one*» можно отнести модели с определенным размером входных и выходных данных.

В случае «*one to many*» при заранее заданном типе и размере входного объекта можно получить вывод разной длины. Такой подход применяется в популярной задаче описания изображений (*image captioning*).

Вариант «*many to one*» работает ровно наоборот – на вход подаются данные нефиксированного размера и получаются их четко определенные характеристики. Так, например, можно по фрагменту видео определять вид активностей или действия, которые в нем происходят.

И, наконец, «*many to many*» архитектуры имеют варьирующиеся размеры как входных, так и выходных данных. К решаемым ими задачам относятся машинный перевод (исходная и переведенная фразы могут быть разной длины) и покадровая классификация видео.

5.2.2. Обучение RNN

Обучение *RNN* аналогично обучению обычной нейронной сети. Используется алгоритм обратного распространения ошибки (*backpropagation*), но с небольшим изменением.

Поскольку одни и те же параметры используются на всех временных этапах в сети, градиент на каждом выходе зависит не только от расчетов текущего шага, но и от предыдущих временных шагов. Например, чтобы вычислить градиент при $t = 4$, нам нужно было бы «распространить ошибку» на три шага и суммировать градиенты. Этот алгоритм называется «алгоритмом обратного распространения ошибки сквозь время» (*Backpropagation Through Time*, *BPTT*).

Однако рекуррентные нейронные сети, прошедшие обучение с *BPTT*, испытывают трудности с изучением долгосрочных зависимостей (например, зависимость между шагами, которые находятся далеко друг от друга) из-за затухания/взрывания градиента. Чтобы обойти эти проблемы, существует определенный механизм, были разработаны специальные архитектуры *RNN* (например *LSTM*) [25].

5.2.3. Архитектуры рекуррентных нейронных сетей

Двунаправленные рекуррентные нейронные сети (*Bidirectional RNNs*) основаны на той идее, что выход в момент времени *t* может зависеть не только от предыдущих элементов в последовательности, но и от будущих. Например, если вы хотите предсказать недостающее слово в последовательности, учитывая как левый, так и правый контекст. Двунаправленные рекуррентные нейронные сети довольно просты. Это всего лишь два *RNN*, уложенных друг на друга. Затем выход вычисляется на основе скрытого состояния обеих *RNN* (рис. 5.24).

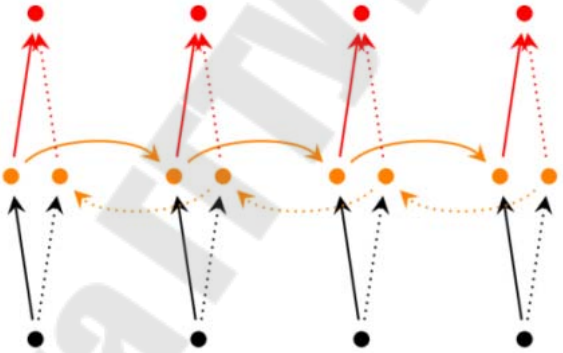


Рис. 5.24. Общая схема работы двунаправленной рекурентной сети

Глубинные рекуррентные нейронные сети похожи на двунаправленные *RNN*, только теперь у нас есть несколько уровней на каждый шаг времени. На практике это даст более высокий потенциал, но нам также потребуется много данных для обучения.

Сети LSTM. Сеть долгой краткосрочной памяти (*Long Short-Term Memory*, *LSTM*) – разновидность архитектуры рекуррентной нейросети, созданная для более точного моделирования временных последовательностей и их долгосрочных зависимостей, чем традиционная рекуррентная сеть (предложена в 1997 г. Зеппом Хохрайтером и Юргеном Шмидхубером (*Jürgen Schmidhuber*)).

Любая рекуррентная нейронная сеть имеет форму цепочки повторяющихся модулей нейронной сети. В обычной *RNN* структура одного такого модуля очень проста, например, он может представлять собой один слой с функцией активации *tanh* (гиперболический тангенс). Но при этом классическая *RNN* не способна учитывать долговременные зависимости. *LSTM* разработаны специально, чтобы избежать этой проблемы. Запоминание информации на долгие периоды времени – это их обычное поведение.

LSTM-сеть не использует функцию активации в рекуррентных компонентах, сохраненные значения не модифицируются, а градиент не стремится исчезнуть во время тренировки. Часто *LSTM* применяется в блоках по несколько элементов. Эти блоки состоят из трех или четырех затворов (например, входного, выходного и гейта забывания), которые контролируют построение информационного потока по логистической функции.

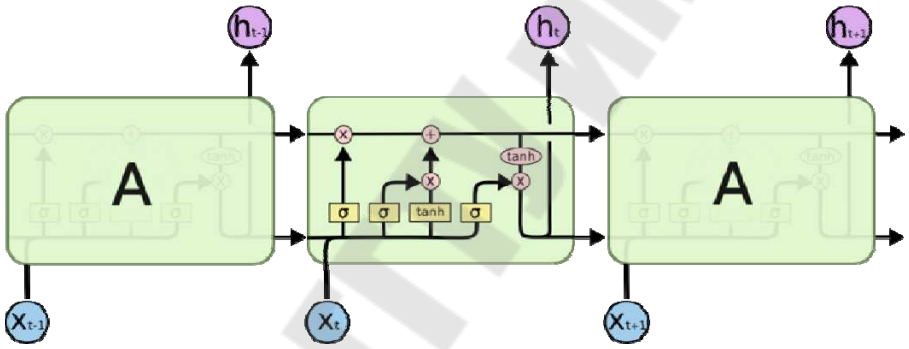


Рис. 5.25. Схема элемента *LSTM*-сети

Ключевой компонент *LSTM* – это состояние ячейки (*cell state*), которое напоминает конвейерную ленту. Она проходит напрямую через всю цепочку, участвуя лишь в нескольких линейных преобразованиях. Информация может легко течь по ней, не подвергаясь изменениям.

Тем не менее *LSTM* может удалять информацию из состояния ячейки; этот процесс регулируется структурами, называемыми фильтрами (*gates*).

Фильтры позволяют пропускать информацию на основании некоторых условий. Они состоят из слоя сигмоидальной нейронной сети и операции поточечного умножения.

Сигмоидальный слой возвращает числа от нуля до единицы, которые обозначают, какую долю каждого блока информации следует пропустить дальше по сети. Ноль в данном случае означает «не про-

пускать ничего», единица – «пропустить все». В *LSTM* три таких фильтра, позволяющих защищать и контролировать состояние ячейки.

GRU. *GRU* (*Gated Recurrent Unit*) – это тип фильтра в рекуррентных нейронных сетях, введенный в 2014 г. *Kyunghyun Cho et al*. Сама формулировка *Gated Recurrent Unit* является более обобщающей и подразумевает включение в себя *LSTM* как частного случая [24].

Схема элементов *GRU*-сети приведена на рис. 5.26.

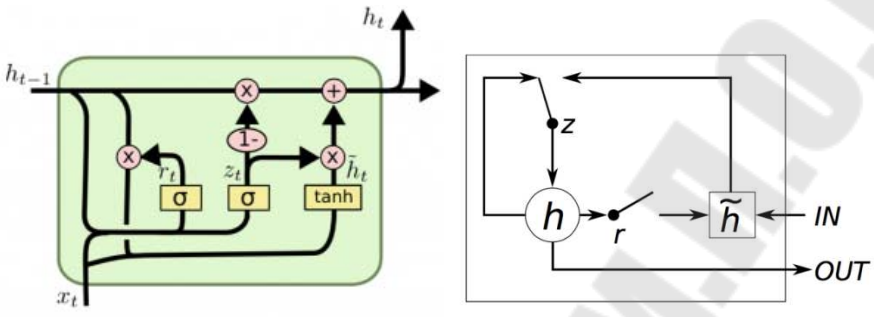


Рис. 5.26. Схема элементов *GRU*-сети

GRU имеет два фильтра: сброса *r* и обновления *z*. Интуитивно понятно, что фильтр сброса определяет, как объединить новый вход с предыдущей памятью, а фильтр обновления определяет, сколько из предыдущей памяти нужно хранить. Если мы установим сброс на все 1 и обновим фильтр на все 0, то получим обычную модель *RNN*.

GRU похож на долговременную кратковременную память (*LSTM*) с фильтром забывания, но имеет меньше параметров, чем *LSTM*, так как в нем отсутствует выходной фильтр. Кроме того, имеется несколько ключевых отличий:

– *GRU* имеет два фильтра, а *LSTM* – три фильтра;

– *GRU* не обладает внутренней памятью и не имеет выходного шлюза, который присутствует в *LSTM*;

– фильтры ввода и забывания соединены фильтром обновления *z*, а фильтр сброса *r* применяется непосредственно к предыдущему скрытому состоянию. Таким образом, ответственность фильтра сброса в *LSTM* действительно разделена на *r* и *z*.

Sequence-to-sequence. *Sequence-to-sequence* модели состоят из двух рекуррентных сетей: кодировщика и декодировщика (рис. 5.27). Одна *RNN* кодирует последовательность символов в векторное представление фиксированной длины, а другая кодирует представление в другую последовательность символов. Кодировщик и декодировщик совместно обучаются для максимизации условной вероятности целевой последовательности при заданной исходной последовательности.

Опытным путем было установлено, что производительность системы статистического машинного перевода улучшается благодаря использованию условных вероятностей пар фраз, вычисленных кодером-декодером *RNN*, в качестве дополнительной функции в существующей лог-линейной модели.

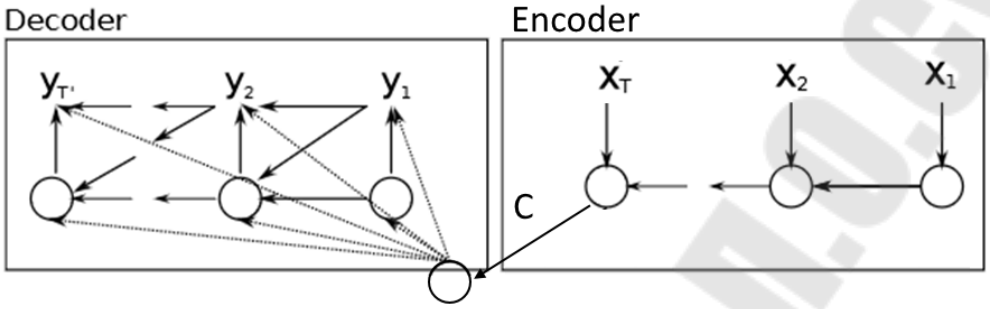


Рис. 5.27. Схема работы кодера-декодера *RNN*

Скрытая активационная функция представлена на рис. 5.28.

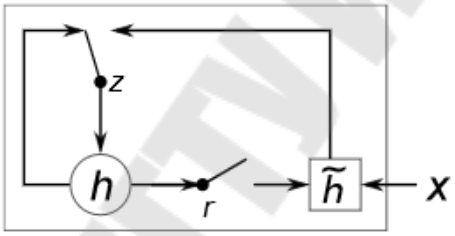


Рис. 5.28. Схема скрытой функции активации

Фильтр обновления *z* выбирает, когда скрытое состояние будет обновлено новым скрытым состоянием. Фильтр сброса решает, игнорируется ли предыдущее скрытое состояние.

5.2.4. Архитектура рекурсивных нейронных сетей (*RvNN*)

Рекурсивные нейронные сети (*Recursive neural network*, *RvNN*) – вид нейронных сетей, работающих с данными переменной длины. Модели рекурсивных сетей используют иерархические структуры образцов при обучении.

Рекурсивная структура обычно встречается в разных модальностях. Известно, что синтаксические правила естественного языка являются рекурсивными, с существительными фразами, содержащими относительные предложения, которые сами по себе содержат слово-

сочетания. Точно так же можно найти вложенное иерархическое структурирование в изображениях сцены, которые фиксируют отношения как части, так и близости (рис. 5.29).

Восстановление этой структуры в исходных данных помогает в понимании и классификации. Выявление структуры сцены и ее деконструкция – нетривиальная задача. При этом необходимо как идентифицировать отдельные объекты, так и всю структуру сцены.

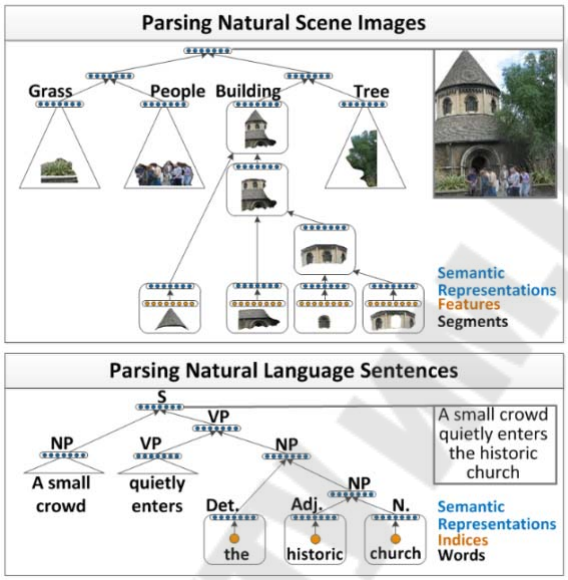


Рис. 5.29. Пример различных рекурсивных структур

В рекурсивных сетях нейроны с одинаковыми весами активируются рекурсивно в соответствии со структурой сети. В процессе работы рекурсивной сети вырабатывается модель для предсказания для структур переменной размерности, так и скалярных структур через активацию структуры в соответствии с топологией (рис. 5.30).

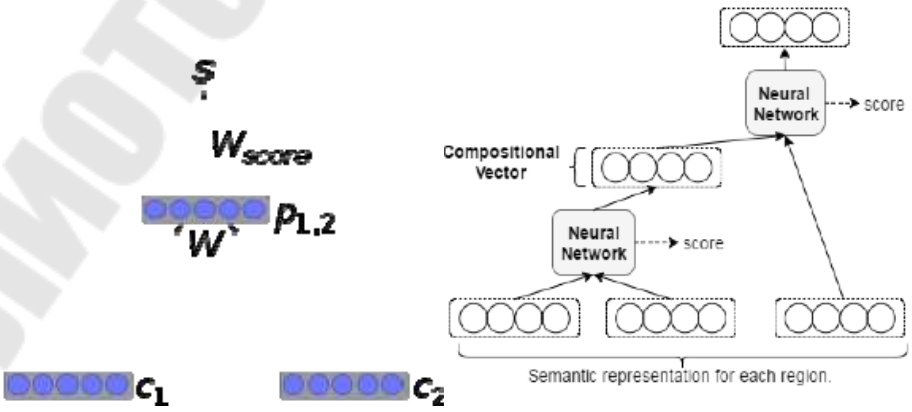


Рис. 5.30. Пример структуры рекурсивной сети

5.3. Генеративно-состязательные нейросети

5.3.1. Архитектура GAN

Генеративное моделирование – это задача машинного обучения без учителя (неконтролируемое обучение), которая заключается в автоматическом обнаружении закономерностей и зависимостей во входных данных, которые можно было бы использовать для генерации на выходе новых примеров, способных непротиворечиво/правдоподобно присутствовать в оригинальном (исходном) наборе данных [16].

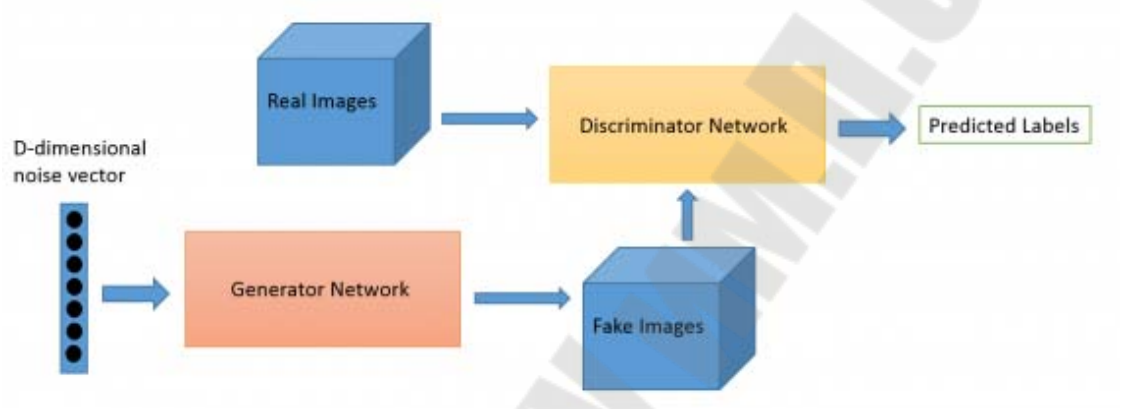


Рис. 5.31. Прицип работы генеративно-состязательной сети

GAN – генеративно-состязательная нейросеть (*Generative adversarial network*, *GAN*) – один из алгоритмов классического машинного обучения, обучения без учителя [26]. Суть идеи в комбинации двух нейросетей, при которой одновременно работает два алгоритма: «генератор» и «дискриминатор». Задача генератора – генерировать образы заданной категории. Задача дискриминатора – пытаться распознать как созданный образ, так и реальный. Две модели обучаются вместе в состязательной игре с нулевой суммой (антагонистической игре), до тех пор, пока модель дискриминатора не начнет «обманываться» примерно в половине случаев, что означает, что модель генератора генерирует правдоподобные примеры.

5.3.2. Модель генератора

Модель генератора принимает случайный вектор фиксированной длины в качестве входных данных и генерирует выборку (рис. 5.32).

Вектор взят случайным образом из гауссовского распределения, и он используется для начального процесса генерации. После обучения точки в этом многомерном векторном пространстве будут соот-

ветствовать точкам в исходной области, образуя сжатое представление распределения данных.

Это векторное пространство называется скрытым пространством или векторным пространством, состоящим из скрытых переменных. Скрытые переменные – это те переменные, которые возможны для наблюдения в исходной области, но не наблюдаются напрямую.

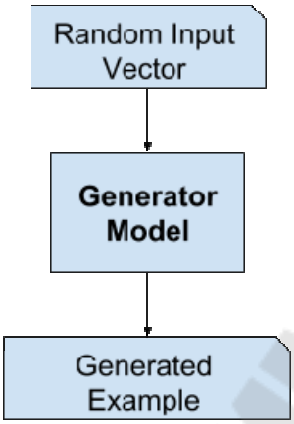


Рис. 5.32. Схема работы модели генератора

Мы часто называем скрытые переменные или скрытое пространство проекцией или сжатым распределением данных. То есть, скрытое пространство обеспечивает сжатие. В случае *GAN* модель генератора выбирает значение из точек в скрытом пространстве так, что новые точки, взятые из скрытого пространства, могут быть предоставлены модели генератора в качестве входных данных и использованы для генерации новых и различных выходных примеров.

После обучения модель генератора сохраняется и используется для генерации новых примеров.

5.3.3. Модель дискриминатора

Модель дискриминатора берет пример из области входных данных (действительный или сгенерированный) и предсказывает двоичную метку класса реального или поддельного (сгенерированного) (рис. 5.33).

Реальный пример берется из учебного набора данных. Сгенерированные примеры выводятся моделью генератора.

Дискриминатор – это нормальная (и хорошо понятная) классификационная модель.

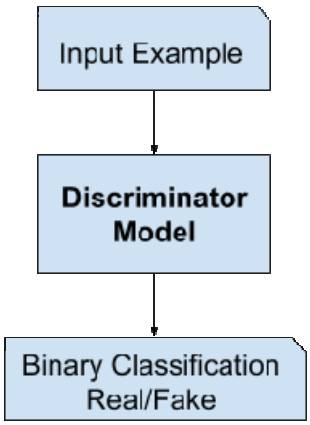


Рис. 5.33. Схема работы модели дискриминатора

Иногда генератор можно изменить, поскольку он научился эффективно извлекать примеры из данных исходной области. Некоторые или все слои модели для извлечения признаков могут применяться в приложениях с использованием таких же или аналогичных входных данных.

5.3.4. Обучение GAN

Две модели, генератор и дискриминатор, обучаются вместе. Генератор генерирует серию выборок, и они, наряду с реальными примерами из исходной области, предоставляются дискриминатору, который классифицирует их как реальные или поддельные.

Затем дискриминатор обновляется, чтобы улучшить распознавание реальных и поддельных выборок в следующем раунде, и, что важно, генератор обновляется на основе того, насколько хорошо или нет сгенерированные выборки обманули дискриминатор.

Таким образом, две модели конкурируют друг с другом, они соперничают в смысле теории игр и играют в игру с нулевой суммой (антагонистическая игра).

Поскольку инфраструктуру *GAN* можно естественно проанализировать с помощью инструментов теории игр, мы называем *GAN* «состязательными» (рис. 5.34).

В этом случае нулевая сумма означает, что, когда дискриминатор успешно идентифицирует реальные и поддельные выборки, он вознаграждается или не требуется никаких изменений для параметров модели, тогда как генератор штрафуется большими обновлениями параметров модели.

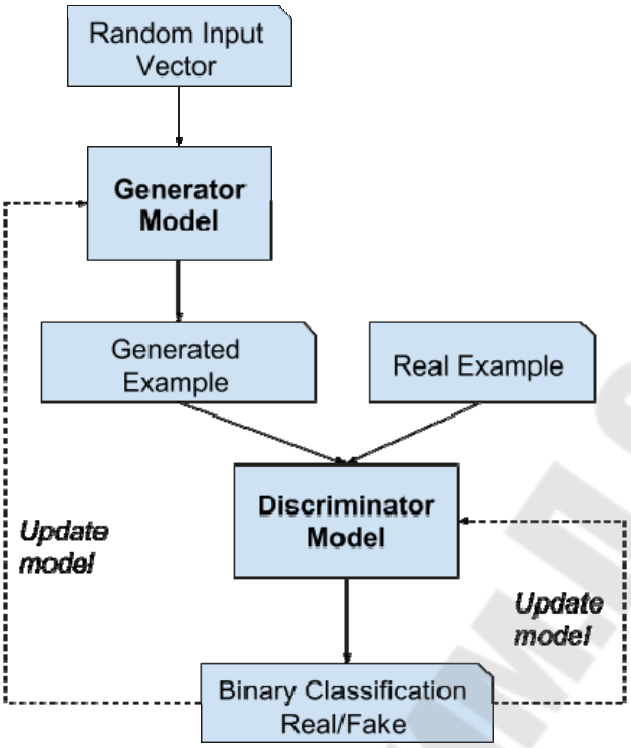


Рис. 5.34. Схема алгоритма обучения генеративно-состязательной сети

С другой стороны, когда генератор вводит в заблуждение дискриминатор, он вознаграждается, или не требует никаких изменений параметров модели, а вот дискриминатор штрафуется, и его параметры модели обновляются.

В конце генератор каждый раз создает точные реплики из входной области, а дискриминатор не может определить разницу и предсказывает «неуверенно» (например, 50 % для реального и поддельного) в каждом случае. Это просто пример идеализированного случая; нам не нужно добираться до этой точки, чтобы прийти к полезной модели генератора.

5.3.5. Условные GAN

Важным расширением *GAN* является их использование для условного генерирования выходных данных.

Генеративные состязательные сети могут быть расширены до условной модели, если и генератор, и дискриминатор обусловлены некоторой дополнительной информацией – это может быть любой вспомогательной информацией, такой как метки классов или данные из других областей (рис. 5.35).

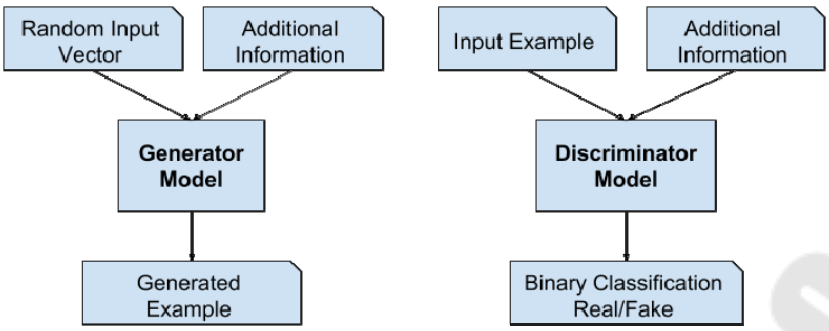


Рис. 5.35. Схема принципа работы условных *GAN*-сетей

Дискриминатор также становится условным, что означает, что он снабжен как реальным или фальшивым входным изображением, так и дополнительным вводом.

В случае условного ввода типа метки классификации дискриминатор ожидает, что этот ввод будет принадлежать этому классу, в свою очередь обучая генератор генерировать примеры этого класса, чтобы обмануть дискриминатор.

Таким образом, условная *GAN* может использоваться для генерации примеров из области данного типа.

В случае условных *GAN* для преобразования изображения в изображение, таких как преобразование дня в ночь, дискриминатор предоставляет примеры реальных и сгенерированных ночных фотографий, а также (обусловленных) реальных дневных фотографий в качестве входных данных. Генератор снабжен случайным вектором из скрытого пространства, а также (обусловленными) реальными дневными фотографиями в качестве входных данных.

ГЛАВА 6. ПРИМЕНЕНИЕ НЕЙРОННЫХ СЕТЕЙ ГЛУБОКОГО ОБУЧЕНИЯ

6.1. Примеры распознавания образов средствами многослойных нейронных сетей глубокого обучения

6.1.1. Распознавание цифр (*MNIST*)

Датасет *MNIST* – набор данных, разработанный Янн ЛеКун, Коринной Кортес и Кристофером Берджесом для оценки моделей машинного обучения по задаче классификации рукописных цифр [19].

Набор данных составлен из нескольких наборов данных различных отсканированных документов, которые нормализованы по размеру и центрированы. За счет этого он является отличным набором данных для оценки моделей.

Каждое изображение представляет собой квадрат 28 на 28 пикселей (всего 784 пикселя). Стандартный набор данных, используемый для оценки и сравнения моделей, содержит 70000 изображений рукописных цифр, где 60000 изображений используются для обучения модели, а отдельный набор из 10000 изображений – для ее проверки.

Задача распознавания цифр – это задача с 10 классами для прогнозирования (от 0 до 9). Результаты сообщаются с использованием ошибки прогнозирования.

Библиотека глубокого обучения *Keras* предоставляет удобный метод загрузки набора данных *MNIST*. Набор данных загружается автоматически при первом вызове функции и сохраняется в домашнем каталоге в виде файла *~/.keras/datasets /mnist.pkl.gz*.

```
# импорт набора данных MNIST
from keras.datasets import mnist
```

После успешного импорта данных требуется построить сверточную нейронную сеть или модель *CNN*. *Keras* предоставляет много возможностей для создания сверточных нейронных сетей.

Построение сверточной нейронной сети стоит начать с импортирования всех необходимых библиотек и функций.

```
# Модель
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, MaxPooling2D,
Dropout
from keras.optimizers import RMSprop
from keras.utils import to_categorical
%matplotlib inline
import matplotlib.pyplot as plt
```

На данном шаге импортируется класс последовательности модели (*Sequential*) и необходимые слои: *Dense* (обычный персептрон), *Flatten* (преобразователь матриц в вектор), *Dropout* (один из методов регуляризации для противостояния переобучению), *Conv2D* (сверточный слой) и *MaxPooling2D* – уменьшает размер исходной матрицы.

Следующим шагом является загрузка импортированного набора данных для дальнейшей работы с ним. Сделать это в *Keras* можно при помощи следующего фрагмента кода:

```
X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train.shape, X_test.shape
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(20, 10))
axes = axes.flat
for i in range(9):
    axes[i].imshow(X_train[i])
    axes[i].axis('off')
```

Полученные данные хранятся в виде изображений 28×28 (рис. 6.1).

Рис. 6.1. Пример данных

Для использования данных изображений в модели *Keras* необходимо произвести преобразование и обработку векторов с целевыми значениями.

```
X_train = X_train.reshape((60000,28, 28, 1)).astype('float32')/ 255
X_test = X_test.reshape((10000, 28, 28, 1)).astype('float32') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

Тип массива изменяется на *float*32 для его дальнейшей нормализации путем деления на 255. А конечные цели сети преобразуются в категории от 0 до 9.

Далее требуется задать архитектуру модели сверточной нейронной сети:

1. Первые два слоя – это сверточные слои, называемые *Convolution*2*D*. Это входные слои, ожидающие изображения.

2. Затем при помощи слоя *MaxPooling* осуществляется понижение размера исходной матрицы.

3. Следующий уровень – это уровень регуляризации с использованием *dropout*, называемый *Dropout*. Он настроен на случайное исключение 40 % нейронов в слое, чтобы уменьшить переоснащение.

4. Далее идет слой, который преобразует данные 2*D*-матрицы в вектор с именем *Flatten*. Это позволяет обрабатывать вывод стандартными, полностью связанными слоями.

5. Затем – полностью подключенный слой со 256 нейронами и функцией активации.

6. Последний, выходной, слой имеет 10 нейронов для 10 классов и функцию активации *softmax* для вывода вероятностных прогнозов для каждого класса.

```
# Архитектура модели сверточной сети
def get_model():
    model = Sequential()
        model.add(Conv2D(64, kernel_size=(3, 3), activation='relu',
input_shape=(28, 28, 1)))
    model.add(Conv2D(128, (3,3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.4))
    model.add(Flatten())
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.4))
    model.add(Dense(10, activation='softmax'))
    model.compile(optimizer=RMSprop(lr=0.01),
loss='categorical_crossentropy',metrics=['accuracy'])
    return model
```

Так как данные уже подготовлены, остается обучить саму модель.

```
    # Обучение модели нейронной сети
    history = model.fit(X_train,
                        y_train,
                        epochs=15,
                        batch_size=500,
                        validation_data=(X_test, y_test))
```

По завершении обучения полученные результаты могут быть визуализированы при помощи следующих команд.

```
# Обучение модели нейронной сети
x = range(15)
plt.grid(True)
plt.plot(x,
         history.history['acc'],
         'bo-',
         label='Train accuracy')
plt.plot(x,
         history.history['val_acc'],
         'ro-',
         label='Validation accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.grid(True)
plt.plot(x,
         history.history['loss'],
         'bo-',
         label='Train losses')
plt.plot(x,
         history.history['val_loss'],
         'ro-',
         label='Validation losses')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
```

Выполнение данного фрагмента кода позволяет получить графики точности и функции потерь итоговой модели (рис. 6.2).

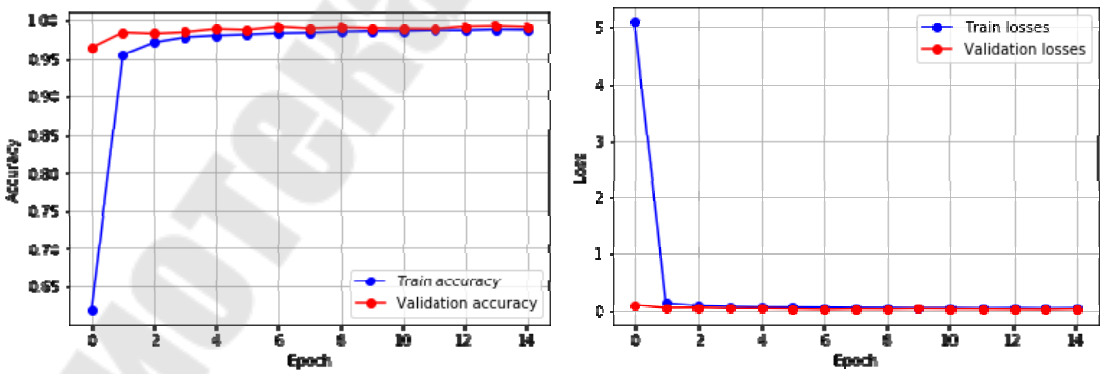


Рис. 6.2. Графики точности и функции потерь модели

Далее рассматривается реализация решения задачи распознавания рукописных цифр из набора изображений *MNIST* с использованием различных фреймворков глубокого обучения.

TensorFlow

Шаг 1. Включить необходимые модули для *TensorFlow* и модули набора данных, которые требуются для вычисления модели *CNN*.

```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
```

Шаг 2. Объявить функцию *run_cnn* ( ), которая включает в себя различные параметры и переменные оптимизации с объявлением заполнителей данных. Эти переменные оптимизации объявят схему обучения.

```
def run_cnn():
  mnist = input_data.read_data_sets("MNIST_data/", one_hot = True)
        learning_rate = 0.0001
        epochs = 10
        batch_size = 50
```

Шаг 3. На этом шаге объявляются заполнители обучающих данных с входными параметрами – для 28 × 28 пикселей = 784. Это сглаженные данные изображения, которые извлекаются из *mnist.train.nextbatch*( ).

Также можно изменить форму тензора в соответствии с необходимыми требованиями. Первое значение (–1) указывает функции динамически формировать это измерение на основе объема данных, переданных ему. Два средних размера установлены на размер изображения (т. е. 28 × 28).

```
x = tf.placeholder(tf.float32, [None, 784])
x_shaped = tf.reshape(x, [-1, 28, 28, 1])
y = tf.placeholder(tf.float32, [None, 10])
```

Шаг 4. Необходимо создать несколько сверточных слоев

```
layer1 = create_new_conv_layer(x_shaped, 1, 32, [5, 5], [2, 2],
name = 'layer1')
layer2 = create_new_conv_layer(layer1, 32, 64, [5, 5], [2, 2],
name = 'layer2')
```

Шаг 5. Стоит осуществить сглаживание выхода, готового для полностью подключенного выходного каскада – после объединения двух слоев шага 2 с размерами 28 × 28, до размера 14 × 14 или мини-

мум 7 × 7 *x*, *y* координаты, но с 64 выходными каналами. Чтобы создать полностью связанный с «плотным» слоем, новая форма должна быть [–1, 7 × 7 × 64]. Можно установить некоторые веса и значения смещения для этого слоя, а затем активировать с помощью *ReLU*.

```
    flattened = tf.reshape(layer2, [-1, 7 * 7 * 64])

    wd1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1000],
stddev = 0.03), name = 'wd1')
    bd1 = tf.Variable(tf.truncated_normal([1000], stddev = 0.01),
name = 'bd1')

    dense_layer1 = tf.matmul(flattened, wd1) + bd1
    dense_layer1 = tf.nn.relu(dense_layer1)
```

Шаг 6. Другой уровень с определенными активациями *softmax* с требуемым оптимизатором задает оценку точности, которая выполняет настройку оператора инициализации.

```
    wd2 = tf.Variable(tf.truncated_normal([1000, 10], stddev = 0.03),
name = 'wd2')
    bd2 = tf.Variable(tf.truncated_normal([10], stddev = 0.01),
name = 'bd2')

    dense_layer2 = tf.matmul(dense_layer1, wd2) + bd2
    y_ = tf.nn.softmax(dense_layer2)

    cross_entropy = tf.reduce_mean(
       tf.nn.softmax_cross_entropy_with_logits(logits =
dense_layer2, labels = y))

    optimiser = tf.train.AdamOptimizer(learning_rate = learn-
ing_rate).minimize(cross_entropy)

    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    init_op = tf.global_variables_initializer()
```

Шаг 7. Далее следует установить переменные записи. Это добавляет сводку для хранения точности данных.

```
  tf.summary.scalar('accuracy', accuracy)
    merged = tf.summary.merge_all()
    writer = tf.summary.FileWriter('E:\TensorFlowProject')
    with tf.Session() as sess:
       sess.run(init_op)
       total_batch = int(len(mnist.train.labels) / batch_size)
       for epoch in range(epochs):
```

```
            avg_cost = 0
          for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size =
batch_size)
              _, c = sess.run([optimiser, cross_entropy], feed_dict = {
               x:batch_x, y: batch_y})
               avg_cost += c / total_batch
            test_acc = sess.run(accuracy, feed_dict = {x:
mnist.test.images, y:
               mnist.test.labels})
               summary = sess.run(merged, feed_dict = {x:
mnist.test.images, y:
               mnist.test.labels})
            writer.add_summary(summary, epoch)
       print("\nTraining complete!")
       writer.add_graph(sess.graph)
       print(sess.run(accuracy, feed_dict = {x: mnist.test.images, y:
          mnist.test.labels}))
    def create_new_conv_layer(
       input_data, num_input_channels, num_filters,filter_shape,
pool_shape, name):
       conv_filt_shape = [
          filter_shape[0], filter_shape[1], num_input_channels,
num_filters]
       weights = tf.Variable(
          tf.truncated_normal(conv_filt_shape, stddev = 0.03), name
= name+'_W')
       bias = tf.Variable(tf.truncated_normal([num_filters]), name =
name+'_b')
    #Out layer defines the output
       out_layer =
          tf.nn.conv2d(input_data, weights, [1, 1, 1, 1], padding =
'SAME')
       out_layer += bias
       out_layer = tf.nn.relu(out_layer)
       ksize = [1, pool_shape[0], pool_shape[1], 1]
       strides = [1, 2, 2, 1]
       out_layer = tf.nn.max_pool(
          out_layer, ksize = ksize, strides = strides, padding =
'SAME')
       return out_layer
    if __name__ == "__main__":
    run_cnn()
```

PyTorch

Импорт *PyTorch* и необходимых библиотек.

```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
```

Инициализация гиперпараметров. Гиперпараметры – это аргументы с предварительной настройкой, которые не будут обновляться в ходе изучения нейронной сети.

```
    input_size = 784       # Размеры изображения = 28 x 28 = 784
    hidden_size = 500      # Количество узлов на скрытом слое
    num_classes = 10       # Число классов на выходе. В этом случае
от 0 до 9
    num_epochs = 5         # Количество тренировок всего набора данных
    batch_size = 100       # Размер входных данных для одной итерации
    learning_rate = 0.001  # Скорость конвергенции
```

Импорт набора данных *MNIST* – базы данных рукописных чисел от 0 до 9.

```
train_dataset = dsets.MNIST(
    root='./data',
    train=True,
    transform=transforms.ToTensor(),
    download=True
)

test_dataset = dsets.MNIST(
    root='./data',
    train=False,
    transform=transforms.ToTensor()
)
```

После загрузки *MNIST* необходимо загрузить набор данных в код:

```
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True
)

test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    shuffle=False
)
```

Обратите внимание: *train_dataset* перемешивается в процессе загрузки, чтобы процесс обучения не зависел от порядка данных, однако порядок *test_loader* остается неизменным.

Наборы данных готовы. Можно приступить к созданию нейронной сети.

Структура модели нейросети. Нейронная сеть включает в себя два полностью соединенных слоя (т. е. *fc*1 и *fc*2) и нелинейный слой *ReLU* между ними.

Запустив следующий код, указанные изображения (*x*) могут пройти через нейронную сеть и сгенерировать вывод (*out*), показывая, как именно соответствие принадлежит каждому из 10 классов.

```
  class Net(nn.Module):
      def __init__(self, input_size, hidden_size, num_classes):
          super(Net, self).__init__()
          self.fc1 = nn.Linear(input_size, hidden_size)   # 1й свя-
занный слой: 784 (данные входа) -> 500 (скрытый узел)
          self.relu = nn.ReLU()
          self.fc2 = nn.Linear(hidden_size, num_classes) # 2й свя-
занный слой: 500 (скрытый узел) -> 10 (класс вывода)

      def forward(self, x):
          out = self.fc1(x)
          out = self.relu(out)
          out = self.fc2(out)
          return out
```

Далее инициализируется нейронная сеть и активируется поддержка *CUDA*.

```
net = Net(input_size, hidden_size, num_classes)
net.cuda()
```

Функция потерь и оптимизатор:

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(),
lr=learning_rate)
```

Тренировка нейросети. Подробные инструкции находятся в комментариях (после #) в следующих примерах.

```
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):   #
Загрузка партии изображений с индексом, данными, классом
        images = Variable(images.view(-1, 28*28))
        labels = Variable(labels)
        optimizer.zero_grad()
        outputs = net(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        if (i+1) % 100 == 0:
```

```
                print('Epoch [%d/%d], Step [%d/%d], Loss: %.4f'
                      %(epoch+1, num_epochs, i+1,
len(train_dataset)//batch_size, loss.data[0]))
```

Проверка модели нейронной сети. Также как и с тренировкой нейронной сети, нужно загрузить порцию тестируемых изображений и собрать выходные данные.

```
correct = 0
total = 0
for images, labels in test_loader:
    images = Variable(images.view(-1, 28*28))
    outputs = net(images)
    _, predicted = torch.max(outputs.data, 1)  # Выбор лучшего клас-
са из выходных данных: класс с лучшим счетом
    total += labels.size(0)                    # Увеличиваем суммар-
ный счет
    correct += (predicted == labels).sum()     # Увеличиваем
корректный счет


print('Accuracy of the network on the 10K test images: %d %%' % (100
* correct / total))
```

Keras

Импорт библиотек и модулей для проекта. В первую очередь осуществляется импорт *numpy* и установка начального числа для генератора псевдослучайных чисел.

```
import numpy as np
np.random.seed(123)  # for reproducibility
```

Далее импортируется тип модели *Sequential* из *Keras*. Это просто линейный набор слоев нейронной сети, и он идеально подходит для сети типа *CNN* с прямой связью. После этого подключаются «основные» слои из *Keras*, слои *CNN* и вспомогательные утилиты.

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils
```

Затем необходимо провести загрузку набора данных *MNIST*. Так как библиотека *Keras* имеет их в своем составе, это легко сделать следующими командами:

```
from keras.datasets import mnist
```

```
# Load pre-shuffled MNIST data into train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Также имеет смысл проверить полученные данные и вывести их на экран.

```
# Форма набора данных
print(X_train.shape)
# Вывод изображения
plt.imshow(X_train[0])
```

Предварительная обработка входных данных для Keras. При использовании бэкэнда *Theano* вы должны явно объявить размер для глубины входного изображения.

Изображения *MNIST* имеют глубину только 1, и необходимо явно объявить это.

Другими словами, требуется преобразовать набор данных из формы (*n*, ширина, высота) в (*n*, глубина, ширина, высота).

```
# Преобразование набора данных из формы (n, ширина, высота) в
(n, глубина, ширина, высота)
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)

# Преобразование типа данных в float32
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# Нормализация значений данных в диапазоне [0, 1]
X_train /= 255
X_test /= 255
```

Предварительная обработка меток классов для Keras. Метод *np_utils.to_categorical* – преобразует вектор класса (целые числа) в двоичную матрицу классов.

```
# Преобразование одномерных массивов классов в 10-мерные матри-
цы классов
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
```

Задание архитектуры модели нейронной сети. Для начала объявляетс последовательная модель, и затем задается входной слой.

```
model = Sequential()
```

```
model.add(Conv2D(32,(3, 3), activation = 'relu', input_shape=(1,28,28), data_format='channels_first'))
```

Входной параметр *shape* должен иметь форму первого образца. В этом случае это то же самое (1, 28, 28), которое соответствует шаблону (глубина, ширина, высота) каждому изображению цифры.

Перые три параметра соответствуют количеству используемых фильтров свертки, количеству строк в каждом ядре свертки и количеству столбцов в каждом ядре свертки соответственно.

Затем следует добавить остальные слои.

```
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))
```

Важно выделить слой *Dropout*. Это метод регуляризации модели с целью предотвращения переоснащения.

MaxPooling2D – это способ уменьшить количество параметров в модели, переместив фильтр пула 2 × 2 по предыдущему слою и взяв максимум четыре значения в фильтре 2 × 2.

Чтобы завершить архитектуру модели, требуется добавить полностью связанный слой, а затем выходной слой:

```
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax')
```

Для плотных слоев первым параметром является выходной размер слоя. *Keras* автоматически обрабатывает связи между слоями.

Обратите внимание, что конечный слой имеет выходной размер 10, соответствующий 10 классам цифр.

Веса из слоев *Convolution* должны быть сделаны одномерными перед передачей их в полностью связанный плотный слой.

Компиляция модели. При компиляции модели объявляется функция потерь и оптимизатор (*SGD*, *Adam* и т. д.).

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

Для обучения модели достаточно объявить размер батча и количество эпох для обучения, а затем передать данные модели.

```
model.fit(X_train, Y_train,
          batch_size=32, epochs=10, verbose=1)
```

Caffe

Рассмотрим этапы построения примера сети «логистическая регрессия». Далее будем считать, что файл называется *linear_regression.prototxt*, и он размещается в директории *examples/mnist* (рис. 6.3).

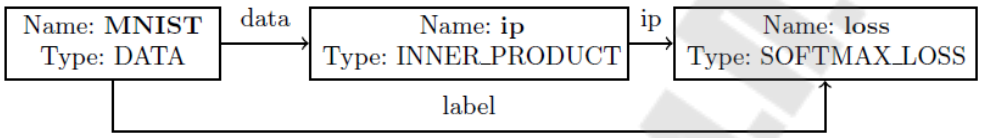


Рис. 6.3. Схема работы сети

Задание имени сети:

```
name: "LinearRegression"
```

В качестве обучающего множества используется база данных *MNIST*, хранящаяся в формате *lmdb*. Для работы с форматами *lmdb* или *leveldb* используется слой типа *DATA*, в котором необходимо указать некоторые параметры, описывающие входные данные (*data_param*): путь до данных на жестком диске (*source*), тип данных (*backend*), размер выборки (*batch_size*). Также с данными можно производить различные преобразования (*transform_param*). В параметре *top* указывается одно или несколько имен, которые будут использованы для идентификации выхода слоя. В данном примере это обработанные изображения (*data*) и метки классов, которым принадлежат изображения (*label*).

```
layers {
  name: "mnist"
  type: DATA
  top: "data"
  top: "label"
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    backend: LMDB
    batch_size: 64
  }
```

```
  transform_param {
    scale: 0.00390625
  }
}
```

Для начала требуется определить полносвязный слой. Полносвязный слой в библиотеке *Caffe* задается с помощью слоя типа *INNER_PRODUCT*. Имя входных данных указывается с помощью параметра *bottom*. В данном слое входными данными являются обработанные изображения (*data*). Количество нейронов в слое определяется автоматически (по количеству выходов в предыдущем слое), а количество выходных нейронов указывается с помощью параметра *num_output*. Результат работы слоя можно расположить по тому же имени, что и имя слоя (*ip*).

```
layers {
  name: "ip"
  type: INNER_PRODUCT
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 10
  }
}
```

В конце добавляется слой, вычисляющий функцию ошибки. Он принимает на вход результат предыдущего полносвязного слоя (*ip*) и номера классов для каждого изображения (*label*). После вычислений к результатам работы данного слоя можно обратиться по имени *loss*.

```
layers {
  name: "loss"
  type: SOFTMAX_LOSS
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
```

Конфигурация сети готова. Далее необходимо определить параметры процедуры обучения в файле формата *prototxt* (в данном примере – *solver.prototxt*). К числу параметров обучения относятся путь к файлу с конфигурацией сети (*net*), периодичность тестирования во время обучения (*test_interval*), параметры стохастического градиентного спуска (*base_lr, weight_decay* и др.), максимальное количество итераций (*max_iter*), архитектура, на которой будут проводиться вы-

числения (*solver_mode*), путь для сохранения обученной сети (*snapshot_prefix*).

```
net: "examples/mnist/linear_regression.prototxt"
test_iter: 100
test_interval: 500
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
lr_policy: "inv"
gamma: 0.0001
power: 0.75
display: 100
max_iter: 10000
snapshot: 5000
snapshot_prefix: "examples/mnist/linear_regression"
solver_mode: GPU
```

Обучение выполняется с использованием основного приложения библиотеки. При этом передается определенный набор ключей, в частности название файла, содержащего описание параметров процедуры обучения.

```
caffe train --solver=solver.prototxt
```

После обучения полученную модель можно использовать для классификации изображений, например, с помощью оберток на *Python* [26] путем следующих действий:

1. Подключение библиотеки *Caffe*. Необходимо установить режим тестирования и указать архитектуру для выполнения вычислений (*CPU* или *GPU*).

```
import caffe
caffe.set_phase_test()
caffe.set_mode_cpu()
```

2. Создать нейронную сеть, указывая следующие параметры: *MODEL_FILE* – конфигурация сети в формате *prototxt*, *PRETRAINED* – обученная сеть в формате *caffemodel*, *IMAGE_MEAN* – среднее изображение (вычисляется по набору входных изображений и используется для последующей нормализации интенсивности), *channel_swap* задает цветовую модель, *raw_scale* – максимальное значение интенсивности, *image_dims* – разрешение изображения. После чего загружаем изображение для классификации (*IMAGE_FILE*).

```
net = caffe.Classifier(MODEL_FILE, PRETRAINED, IMAGE_MEAN,
channel_swap=(0,1,2), raw_scale=255, image_dims=(28, 28))
input_image = caffe.io.load_image(IMAGE_FILE)
```

Получить ответ нейросети для выбранного изображения и вывести результаты на экран.

```
prediction = net.predict([input_image])
print 'prediction shape:', prediction[0].shape
print 'predicted class:', prediction[0].argmax()
```

Таким образом, путем несложных действий можно получить первые результаты работы нейронной сети на основе *Caffe*.

Deeplearning4j

Как и в большинстве языков программирования, необходимо явно импортировать в область видимости классы, которые будут использоваться. Ниже импортируются обычные *Deeplearning*4*j* классы, которые помогут настроить и обучить нейронную сеть. Приведенный ниже код написан на *Scala*.

Импортируются методы из класса *JavaConversions* в *Scala*, потому что это позволит использовать родные коллекции *Scala*, сохраняя при этом совместимость с коллекциями *Java* в *Deeplearning*4*j*.

```
import scala.collection.JavaConversions._

import org.deeplearning4j.datasets.iterator._
import org.deeplearning4j.datasets.iterator.impl._
import org.deeplearning4j.nn.api._
import org.deeplearning4j.nn.multilayer._
import org.deeplearning4j.nn.graph._
import org.deeplearning4j.nn.conf._
import org.deeplearning4j.nn.conf.inputs._
import org.deeplearning4j.nn.conf.layers._
import org.deeplearning4j.nn.weights._
import org.deeplearning4j.optimize.listeners._

import
org.deeplearning4j.datasets.datavec.RecordReaderMultiDataSetIterator
import org.nd4j.evaluation.classification._
import org.nd4j.linalg.learning.config._ // for different updaters
like Adam, Nesterovs, etc.
import org.nd4j.linalg.activations.Activation // defines different
activation functions like RELU, SOFTMAX, etc.
import org.nd4j.linalg.lossfunctions.LossFunctions // mean squared
error, multiclass cross entropy, etc.
```

Итераторы набора данных – важные части кода, которые помогают объединять и повторять итерации по всему набору данных для обучения и получения результатов нейронной сетью. *Deeplearning*4*j* поставляется со встроенной реализацией итератора баз данных *BaseDatasetIterator* для *MNIST*, известного как *EmnistDataSetIterator*. Этот итератор является удобной утилитой, которая обрабатывает загрузку и подготовку данных.

Ниже создается два различных итератора, один из которых предназначен для тренировочных данных, а другой – для оценки точности модели после тренировки. Последний булевый параметр в конструкторе указывает на то, запускается ли обучение нейронной сети или ее проверка.

```
    import
org.deeplearning4j.datasets.iterator.impl.EmnistDataSetIterator

    val batchSize = 128 // how many examples to simultaneously
train in the network
    val emnistSet = EmnistDataSetIterator.Set.BALANCED
    val emnistTrain = new EmnistDataSetIterator(emnistSet, batch-
Size, true)
    val emnistTest = new EmnistDataSetIterator(emnistSet, batch-
Size, false)
```

Для любой нейросети, построенной в *Deeplearning*4*j*, основой является класс *NeuralNetConfiguration*. Здесь происходит настройка гиперпараметров, значений, определяющих архитектуру и то, как алгоритм учится.

Метод *list*( ) определяет количество слоев в сети; эта функция реплицирует конфигурацию *n* раз и строит конфигурацию по слоям.

```
val outputNum = EmnistDataSetIterator.numLabels(emnistSet) // total
output classes
val rngSeed = 123 // integer for reproducability of a random number
generator
val numRows = 28 // number of "pixel rows" in an mnist digit
val numColumns = 28

val conf = new NeuralNetConfiguration.Builder()
            .seed(rngSeed)
            .updater(new Adam())
            .l2(1e-4)
            .list()
            .layer(new DenseLayer.Builder()
                .nIn(numRows * numColumns) // Number of input data-
points.
```

```
                .nOut(1000) // Number of output datapoints.
                .activation(Activation.RELU) // Activation function.
                .weightInit(WeightInit.XAVIER) // Weight initializa-
tion.
                .build())
            .layer(new Output-
Layer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
                .nIn(1000)
                .nOut(outputNum)
                .activation(Activation.SOFTMAX)
                .weightInit(WeightInit.XAVIER)
                .build())
            .build()
```

Теперь, когда создан *NeuralNetConfiguration*, можно использовать конфигурацию для создания многоуровневой сети (*MultiLayerNetwork*). Когда вызывается метод *init*( ) в сети, он применяет выбранную инициализацию весов по всей сети и позволяет передавать данные для тренировки.

В реализованной модели имеется метод *fit*( ), который принимает итератор набора данных (итератор, расширяющий *BaseDatasetIterator*), один *DataSet* или *ND-Array* (реализация *INDArray*). Так как итератор *EMNIST* уже расширяет базовый класс итератора, то возможно передать его непосредственно в нужное место. Если необходимо обучать сеть в течение нескольких эпох, следует поместить число полных эпох во второй аргумент метода *fit*( ).

```
    // create the MLN
    val network = new MultiLayerNetwork(conf)
    network.init()

    // pass a training listener that reports score every 10 iterations
    val eachIterations = 10
    network.addListeners(new
ScoreIterationListener(eachIterations))

    // fit a dataset for a single epoch
    // network.fit(emnistTrain)

    // fit for multiple epochs
    // val numEpochs = 2
    // network.fit(emnistTrain, numEpochs)

    // or simply use for loop
    // for(i <- 1 to numEpochs) {
    //   println("Epoch " + i + " / " + numEpochs)
    //   network.fit(emnistTrain)
    // }
```

Оценка модели. *Deeplearning4j* предоставляет несколько инструментов для оценки работы модели. Можно выполнить базовую оценку и получить такие метрики, как *precision* и *accuracy*, или же воспользоваться функцией *ROC* (*Receiver Operating Characteristic*). Следует отметить, что общий класс *ROC* работает для бинарных классификаторов, в то время как *ROCMultiClass* предназначен для классификаторов, подобных создаваемой здесь модели.

В *MultiLayerNetwork* имеется несколько встроенных методов, которые позволяют произвести оценку. Итератор датасета с данными тестирования/проверки можно передать в метод *evaluation*( ).

```
// evaluate basic performance
val eval = network.evaluate[Evaluation](emnistTest)
println(eval.accuracy())
println(eval.precision())
println(eval.recall())

// evaluate ROC and calculate the Area Under Curve
val roc = network.evaluateROCMultiClass[ROCMultiClass](emnistTest, 0)
roc.calculateAUC(classIndex)

// optionally, you can print all stats from the evaluations
print(eval.stats())
print(roc.stats())
```

Таким образом, одни и те же задачи машинного обучения могут быть решены на различных платформах с применением различных фреймворков глубокого обучения и различных моделей. Однако наиболее удобным является использование библиотекb *Keras*, являющейся высокоуровневой прослойкой и позволяющей использовать большинство актуальных фреймворков глубокого обучения. Данная библиотека дает возможность описать модель нейронной сети в едином универсальном стиле, а затем выполнить ее с задействованием любого необходимого фреймворка.

6.1.2. Распознавание кошек и собак (*Dog vs Cats*) *и видов одежды* (*FASHION_MNIST*)

Fashion MNIST предназначен для замены классического набора данных *MNIST*, который содержит изображения рукописных цифр (0, 1, 2 и т. д.) в формате, идентичном формату изображений одежды в наборе данных *Fashion MNIST* [23]. Пример данных *Fashion MNIST* приведен на рис. 6.4.

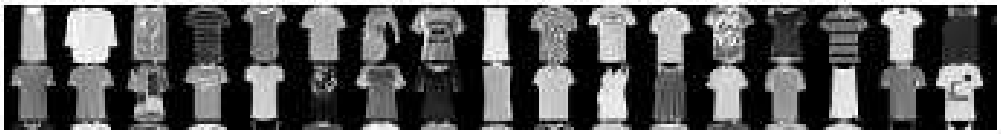


Рис. 6.4. Пример данных *Fashion MNIST*

Набор данных содержит изображения различных типов одежды, таких как футболки, топы, сандалии и т. д. Полный перечень классов дан на рис. 6.5.

Метка	Класс
0	Футболка / топ
1	Шорты
2	Свитер
3	Платье
4	Плащ
5	Сандали
6	Рубашка
7	Кроссовки
8	Сумка
9	Ботинки

Рис. 6.5. Перечень классов данных

Каждому входному изображению соответствует одна из перечисленных выше меток. Набор данных *Fashion MNIST* содержит 70000 изображений. Для обучения нейронной сети используется 60000 изображений. Оставшиеся 10000 изображений используются для проверки, насколько правильно сеть обучилась классификации.

Доступ к набору данных *Fashion MNIST* может быть получен напрямую из *TensorFlow* за счет импорта с применением библиотеки *Keras*.

Первым шагом является подключение необходимых для работы библиотек.

```
# TensorFlow и tf.keras
import tensorflow as tf
from tensorflow import keras

# Вспомогательные библиотеки
import numpy as np
import matplotlib.pyplot as plt
```

Далее необходимо осуществить импорт и загрузку данных из датасета. Это делается при помощи библиотеки *Keras*, имеющей в своем составе функцию загрузки наборов данных.

```
# Загрузка и импорт набора данных
fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fash-
ion_mnist.load_data()
```

Загрузка датасета возвращает четыре массива *NumPy*:

– массивы *train_images* и *train_labels* являются тренировочным набором данных – данными, на которых модель будет обучаться;

– модель тестируется на проверочном наборе данных, а именно массивах *test_images u test_labels*.

Изображения являются 28 × 28 массивами *NumPy*, где значение пикселей варьируется от 0 до 255. Метки (*labels*) – это массив целых чисел от 0 до 9. Они соответствуют классам одежды, изображенной на картинках.

Для дальнейшего удобного вывода результатов классификации целесообразно сохранить массив наименований классов.

```
# Массив наименований классов
class_names = ['футболка', 'брюки', 'свитер', 'платье',
'пальто', 'туфли', 'рубашка', 'кроссовки', 'сумка', 'ботинки']
```

Прежде чем обучать нейронную сеть, полученные данные должны быть предобработаны соответствующим образом. Данные в тренировочном датасете содержат изображения со значениями пикселей в диапазоне от 0 до 255. Необходимо привести эти значения к диапазону от 0 до 1. Для этого следует поделить их на 255. Важно, чтобы тренировочный сет и проверочный сет были предобработаны одинаково.

Построение модели. Построение модели нейронной сети требует правильной конфигурации каждого слоя и последующей компиляции модели. Базовым строительным блоком нейронной сети является слой. Слои извлекают образы из данных, которые в них подаются.

Большая часть глубокого обучения состоит из соединения в последовательность простых слоев. Большинство слоев, таких как *Dense* и *Conv*2*D*, имеют параметры, которые настраиваются во время обучения.

```
# Построение модели
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])
```

Первый слой данной сети – *Flatten*, преобразует формат изображения из двумерного массива (28 на 28 пикселей) в одномерный (размерностью 28 × 28 = 784 пикселя). Слой извлекает строки пикселей из изображения и выстраивает их в один вектор. Этот слой не имеет параметров для обучения – он только переформатирует данные.

После разложения пикселей модель содержит два слоя *Dense*. Это полносвязные слои нейронной сети. Первый *Dense* слой состоит из 128 узлов (или нейронов). Второй – 10-узловой *softmax* слой – возвращает массив из 10 вероятностных оценок, дающих в сумме 1. Каждый узел содержит оценку, указывающую вероятность принадлежности изображения к одному из 10 классов.

Для начала обучения модели ее необходимо предварительно скомпилировать с нужными параметрами.

```
# Компиляция и обучение модели
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=10)
```

При компиляции для модели задаются следующие параметры:

– функция потерь (*Loss function*) – измеряет точность модели во время обучения. Необходимо минимизировать эту функцию, чтобы направить модель в верном направлении;

– оптимизатор (*Optimizer*) – указывает, каким образом обновляется модель на основе входных данных и функции потерь;

– метрики (*Metrics*) – используются для мониторинга тренировки и тестирования модели. В данном случае используется метрика *accuracy* (точность), равная доле правильно классифицированных изображений.

Обучение модели нейронной сети состоит из следующих шагов:

– подача тренировочных данных в модель. В этом примере тренировочные данные – это массивы *train_images* и *train_labels*;

– обучение модели ассоциировать изображения с нужными классами;

– проверка модели путем осуществления прогнозов для проверочного набора данных (в данном примере – массив *test_images*). Проверяется, соответствуют ли предсказанные классы меткам из массива *test_labels*.

Для начала обучения необходимо вызвать метод *model.fit*, осуществляющий тренировку модели на указанном наборе данных.

После обучения модель может быть использована для формирования предсказаний. Запросить предсказание для изображения можно следующим образом:

```
# Предсказание класса
predictions = model.predict(test_images)
predictions[0]
```

Прогноз представляет из себя массив из 10 чисел. Они описывают «уверенность» (*confidence*) модели в том, насколько изображение соответствует каждому из 10 разных видов одежды.

```
array([3.9092092e-06, 5.0955748e-08, 1.0088089e-08, 1.7653681e-09,
1.0856661e-08,    2.4640898e-04,    3.9162201e-06,    1.8451020e-02,
1.2714973e-05, 9.8128188e-01], dtype=float32)
```

С помощью команды *np.argmax*(*predictions*[0]) можно просмотреть отдельно, какой метке класса соответствует максимальное значение.

6.1.3. Распознавание объектов на изображениях

Неотъемлемой частью компьютерного зрения является обнаружение объектов. Сверточные нейронные сети (*CNN*) – наиболее популярная архитектура для работы с изображениями. Однако ее ключевой задачей является классификация изображений по их содержимому. Разница между алгоритмами распознавания объектов и алгоритмами классификации заключается в том, что в алгоритмах распознавания требуется нарисовать ограничивающий прямоугольник вокруг интересующего объекта либо множества различных объектов, чтобы найти их на изображении, в то время как задачей классификации является идентификация находящегося на изображении объекта.

Основной причиной, почему данная задача не может быть решена путем построения стандартной сверточной сети, за которой следует полностью связанный слой, заключается в том, что размер выходного слоя является переменной, а не постоянной величиной. Это обусловле-

но тем фактом, что число вхождений объектов, представляющих интерес, не известно заранее. Выделение областей интереса с последующей передачей в *CNN* для классификации присутствующих в них объектов также не даст результата, так как представляющие интерес объекты могут иметь разное пространственное местоположение на изображении и разные пропорции, что потребует создания огромного множества различных регионов.

Для решения данной задачи были разработаны более совершенные модели сверточных нейронных сетей, такие как *R–CNN*, *YOLO* и т. д.

Алгоритм *R–CNN* (*Regions With CNNs*) стал первой попыткой решения проблемы множества регионов (рис. 6.6). Авторы данного алгоритма применяют метод *Selective Search* для получения ~2000 регионов изображения, предположительно содержащих объекты. Принцип работы данного метода:

– начальная сегментация на множество областей-кандидатов;

– применение жадного алгоритма для рекурсивного объединения похожих областей в более крупные;

– использование сгенерированных областей в качестве окончательных областей-кандидатов.

Полученные области преобразуются в квадратные фрагменты и передаются на сверточную сеть, выдающую 4096-мерный вектор признаков. Данный вектор передается слою с *SVM* на окончательную классификацию объекта в предложенных областях-кандидатах [31].

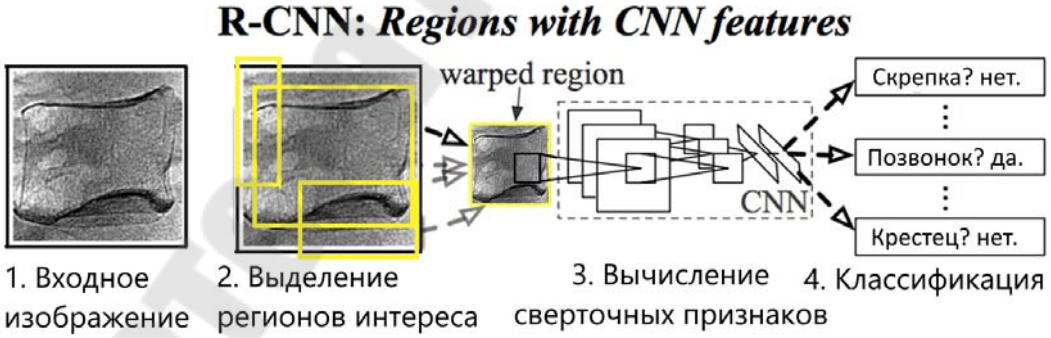


Рис. 6.6. Общая схема работы *R–CNN*

Недостатки *R–CNN*:

– обучение сети занимает длительное количество времени, так как требуется классифицировать 2000 предложений по регионам для каждого изображения;

– не может быть реализован в режиме реального времени, поскольку для каждого изображения требуется около 47 с на локализацию;

– алгоритм *Selective Search* является фиксированным алгоритмом. Для него не проводится обучения и он может генерировать плохие регионы-кандидаты.

Fast R–CNN – схожий с классическим *R–CNN* алгоритм, однако в данном случае на исходном изображении не осуществляется поиск регионов-кандидатов (рис. 6.6). Оно целиком передается в *CNN* для формирования карты признаков, на которой затем уже находятся регионы-кандидаты, классифицируемые полносвязным слоем. Ускорение достигается за счет единоразового использования сверточной модели.

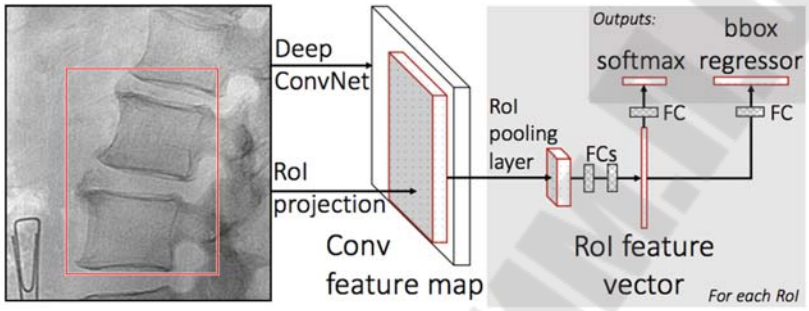


Рис. 6.7. Общий принцип работы *Fast R–CNN*

Более совершенной архитектурой является *Faster R–CNN*. В данной архитектуре отсутствует алгоритм *selective search*. Аналогично *Fast R–CNN* на первом этапе формируется карта признаков. Однако для поиска регионов-кандидатов используется отдельная обучаемая нейронная сеть, генерирующая значительно меньшее число регионов.

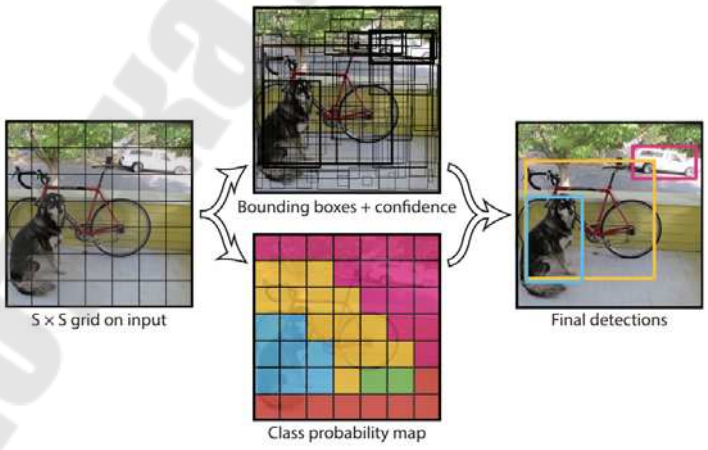


Рис. 6.8. Принцип анализа изображений в *YOLO*

Наиболее быстрой и точной архитектурой для локализации объектов на изображении является *YOLO* (рис. 6.8). Данная модель имеет одну сверточную сеть и для поиска регионов, и для их классификации.

Принцип работы такой сети состоит в разбиении изображения на фиксированные области, в которых затем происходит поиск наличия объекта. Далее найденные ячейки идентифицируются как принадлежащие к разным объектам. Соседние ячейки, содержащие части одного объекта, объединяются. После объединения обнаруженные объекты передаются классификатору. Таким образом, классификатору передаются лишь фрагменты изображения, гарантированно содержащие объекты, а не все изображение.

6.1.4. Анализ текстов с помощью рекуррентных нейронных сетей

Пример кода для построения и обучения простейшей рекурентной сети для анализа текстов отзывов показан ниже:

Шаг 1. Импортирование необходимых библиотек в программу.

```
# Импорты
 import numpy as np
 from keras.preprocessing import sequence
 from keras.models import Sequential
 from keras.layers import Dense, Activation, Embedding
 from keras.layers import Dense, Activation, Embedding
 from keras.layers import LSTM
 from keras.datasets import imdb
```

Шаг 2. Необходимо обеспечить повторяемость результатов, а также указать какое количество слов из словаря будет использовано при обучении. Это осуществляется с помощью следующего фрагмента кода:

```
np.random.seed(42)
max_features = 5000
```

Шаг 3. Далее следует загрузить в программу обучающий набор данных и осуществить его предварительную подготовку. Для этого необходимо задать размер одного отзыва в словах, а затем обрезать слишком длинные отзывы датасета либо расширить короткие путем заполнения пробелами.

```
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words =
max_features)
maxlen = 80
X_train = sequence.pad_sequences(X_train, maxlen = maxlen)
X_test = sequence.pad_sequences(X_test, maxlen = maxlen)
```

Шаг 4. На данном этапе задается модель последовательной сети и формируются составляющие ее слои:

1) два слоя векторного представления слов с показателем *dropout* 0.2;

2) слой долго-краткосрочной памяти размером 100 элементов и показателем *dropout* 0.2 является основой модели;

3) полносвязный слой из одного элемента с сигмоидальной функцией активации.

```
model = Sequential()
model.add(Embedding(max_features, 32, dropout = 0.2))
model.add(Embedding(max_features, 32, dropout = 0.2))
model.add(LSTM(100, dropout_W = 0.2, dropout_U = 0.2))
model.add(Dense(1, activation = 'sigmoid'))
```

Шаг 5. Завершающий шаг, на котором сформированная модель компилируется, обучается и проверяется.

```
model.compile(loss = 'binary_crossentropy',
              optimizer = 'adam',
              metrics = ['accuracy'])
# Обучение
model.fit(X_train, y_train,
           batch_size = 64,
           nb_epoch = 7,
          validation_data = (X_test, y_test),
           verbose = 1)
scores = model.evaluate(X_test, y_test, batch_size = 64)
print('Точность на тестовых данных: %.2f%%' % (scores[1] * 100))
```

6.1.5. Распознавание голоса и речи

Подобно распознаванию изображений, наиболее важной частью распознавания речи является преобразование аудиофайлов в массивы 2 × 2. Наиболее очевидный способ – построение спектрограммы аудиотрека.

Спектрограмма представляет собой спектрально-временное представление звука. Горизонтальное направление спектрограммы представляет время, вертикальное направление – частоту. Спектрограммы могут использоваться для визуализации изменения частотного содержания нестационарного сигнала с течением времени. Формула спектрограммы приведена ниже.

$$spectrogramm(t,\omega) = \left|STFT(t,\omega)\right|^2. \qquad (6.1)$$

Получить спектрограмму *Wav*-файла в языке *Python* [25] можно при помощи следующего фрагмента кода:

```
  def log_spectrogram(file, label):
      sample_rate, samples = wavfile.read(str(train_audio_path) +
'\\'+label+'\\' + file)
      signals = tf.cast(tf.reshape(samples, [1,-1 ]),tf.float32)
      spectrogram = signal.stft(signals, frame_length=1024,
frame_step= 512)
      magnitude_spectrograms = tf.abs(spectrogram)
      log_offset = 1e-6
      #When compressing with a logarithm, it's a good idea to use a
stabilizing offset
      #to avoid high dynamic ranges caused by the singularity at
zero.
      log_magnitude_spectrograms = tf.log(magnitude_spectrograms +
log_offset)
      return log_magnitude_spectrograms

  log_spe_bed =
log_spectrogram(train.file[0],train.label[0]).numpy()
  array_bed = log_spe_bed.astype(np.float)[0]
  fig = plt.figure(figsize=(14,8))
  #plt.ylabel("Freqs in Hz")
  plt.xlabel("Log_Spectrogram")
  plt.imshow(np.swapaxes(array_bed,0,1).T)
```

Результат представлен на рис. 6.9.

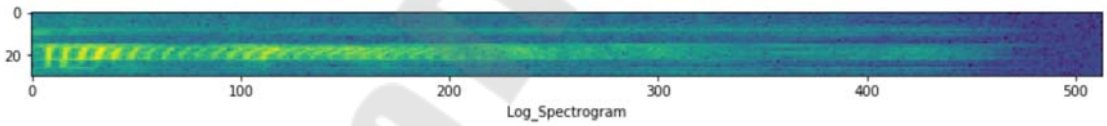


Рис. 6.9. Пример спектрограммы звукового файла

Альтернативой спектрограмме являются мел-кепстральные коэффициенты (*MFCC*). Это функция, которая широко используется для автоматического распознавания речи и речи. Шкала *Mel* соотносит воспринимаемую частоту или высоту чистого тона с фактической измеренной частотой. Люди гораздо лучше различают небольшие изменения высоты звука на низких частотах, чем на высоких. Включение этой шкалы делает функции более близкими к тому, что слышат люди. В данном случае в результирующем изображении содержится куда меньше побочной информации, что упрощает задачу распознавания речи.

Получить их можно следующим образом:

```
def mfcc(file=train['file'].tolist(), label=train['label'].tolist()):
    sample_rate, samples = wavfile.read(str(train_audio_path) +
'\\'+label+'\\' + file)
    if len(samples) < 16000:
        samples = np.pad(samples, (0,16000-len(samples)), 'linear_ramp')
    else:
        samples = samples[:16000]
    signals = tf.cast(tf.reshape(samples, [1,-1 ]),tf.float32)
    spectrogram = signal.stft(signals, frame_length=1024,
frame_step= 512)
    magnitude_spectrograms = tf.abs(spectrogram)
    num_spectrogram_bins = magnitude_spectrograms.shape[-1].value
    lower_edge_hertz, upper_edge_hertz, num_mel_bins = 80.0, 7600.0,
64
    linear_to_mel_weight_matrix =
tf.contrib.signal.linear_to_mel_weight_matrix(num_mel_bins,
num_spectrogram_bins, sample_rate,
lower_edge_hertz,upper_edge_hertz)
    mel_spectrograms = tf.tensordot(magnitude_spectrograms, lin-
ear_to_mel_weight_matrix, 1)
case.mel_spectrograms.set_shape(magnitude_spectrograms.shape[:-
1].concatenate(linear_to_mel_weight_matrix.shape[-1:]))
    log_offset = 1e-6
    log_mel_spectrograms = tf.log(mel_spectrograms + log_offset)
    num_mfccs = 13
    mfccs =
tf.contrib.signal.mfccs_from_log_mel_spectrograms(log_mel_spectrogra
ms)[..., :num_mfccs]
    return mfccs.numpy()[0]
mfcc_bed = mfcc(train.file[0],train.label[0])
fig = plt.figure(figsize=(14,8))
plt.ylabel("MFCC (log) coefficient")
     plt.imshow(np.swapaxes(mfcc_bed,0,1))
```

На рис. 6.10 представлено графическое отображение *MFCC*.

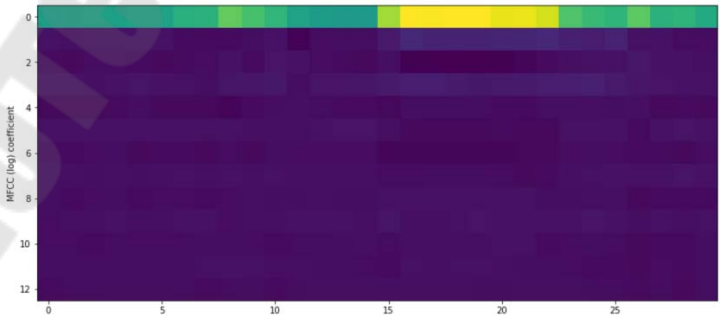


Рис. 6.10. Пример *MFCC*-коэффициентов звукового файла

Далее с данными изображениями может работать классическая модель на основе *Keras*, собранная из *Conv*-слоев или *Dense*-слоев.

Пример кода модели:

```
    # Dense(64) is a fully-connected layer with 64 hidden units.
    # in the first layer, you must specify the expected input data
shape:
    # here, 20-dimensional vectors.
    with tf.Session() as sess0:
        assert not tf.executing_eagerly()
        model = Sequential()

        model.add(layers.Dense(32,
input_shape=X_train_array.shape[1:], activation='tanh'))
        model.add(Dense(64, activation='tanh'))
        model.add(Dense(128, activation='tanh'))
        model.add(Flatten())
        model.add(Dense(256, activation='relu'))
        model.add(Dense(30))
        model.add(Activation('sigmoid'))
        model.compile(loss='categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])
        model.summary(
       #  history = model.fit(x=X_train_array, y=y_train_array,
epochs=5, verbose=1, validation_split = 0.33, shuffle=True,
class_weight=get_class_weights(pd.Series((list(set(labels))),dtype='
category').cat.codes.values),batch_size=batch_size)
        history = model.fit(x=X_train_array, y=y_train_array,
epochs=25, verbose=1, validation_split = 0.1, shuffle=True,
class_weight=get_class_weights(pd.Series(Y_train,dtype='category').c
at.codes.values),batch_size=128)
        model_evaluation = model.evaluate(x=X_test_array,
y=y_test_array, batch_size=None, verbose=1)
        prediction = model.predict(X_test_array, batch_size = 128,
verbose = 1)
        april_tst = model.predict(mfcc_april_test, batch_size =
128, verbose = 1)
        sess0.close()
```

6.2. Использование предобученных архитектур и переобучение

6.2.1. Перенос обучения (transfer learning). Использование предобученной нейросети

Перенос обучения (трансферное обучение) – это техника машинного обучения, при которой модель, обученная по одному заданию, переопределяется по второму связанному заданию.

Современные глубокие нейронные сети в большинстве случаев имеют множество слоев со сложной структурой. На практике обучение целых глубоких нейронный сетей обычно не производится с нуля с произвольной инициализацией. Причина состоит в том, что для многих задач не удается найти набор данных достаточного размера, требуемого для тренировки сети нужной глубины.

Вместо этого чаще всего происходит предварительное обучение сети на очень крупном универсальном наборе данных, а затем использование весов обученной модели либо в качестве инициализации, либо в качестве выделения отличительных признаков для определенной задачи. В данных условиях *transfer learning* позволяет переобучить новым признакам лишь конечный формирующий предсказания слой либо осуществить перенастройку уже обученных весов сети (тонкая настройка/*fine tuning*), что значительно снижает как время обучения, так и требования к размеру обучающего набора данных.

Тонкая настройка. Стратегии переноса обучения зависят от разных факторов, но наиболее важными являются два: размер нового набора данных и его схожесть с исходным набором данных. Если учесть, что характер работы глубоких нейронных сетей более универсален на ранних слоях и становится более тесно связанным с конкретным набором данных на последующих слоях, можно выделить четыре основных сценария:

– новый набор данных меньше по размеру и аналогичен по содержанию исходному набору данных. Если объем данных невелик, то нет смысла проводить тонкую настройку нейросети из-за переобучения (*overfittin*). Поскольку данные схожи с изначальными, можно предполагать, что отличительные черты в нейронной сети будут релевантны и для этого набора данных. Поэтому оптимальным решением является обучение лишь линейного классификатора отличительным признаком нейронной сети;

– новый набор данных относительно крупный и аналогичен по содержанию исходному набору данных. Поскольку имеется больше данных, можно не беспокоиться о переобучении при попытке провести тонкую настройку всей сети;

– новый набор данных меньше по размеру и существенно отличается по содержанию от исходного набора данных. В силу малого объема данных будет достаточно переобучения только линейного классификатора. Так как данные существенно отличаются, лучше обучать классификатор не с вершины сети, где содержатся более конкретные данные, а активировав его на более ранних слоях сети;

– новый набор данных относительно крупный и существенно отличается по содержанию от исходного набора данных. Поскольку набор данных очень крупный, можно обучить всю глубокую нейронную сеть с нуля. Тем не менее на практике оказывается выгоднее использовать для инициализации веса из заранее обученной модели.

В большинстве случаев при тонкой настройке затрагиваются лишь верхние слои нейронной сети, так как на более ранних слоях содержатся глобальные более универсальные признаки (такие как определение краев и цветов), слабо зависящие от набора данных.

Ограничения *transfer learning*. Поскольку используется заранее обученная сеть, выбор архитектуры модели несколько ограничен. Например, невозможно произвольным образом убрать сверточные слои из заранее обученной модели. Тем не менее благодаря совместному использованию параметров становится возможным запустить заранее обученную сеть для изображений разного пространственного размера. Это наиболее очевидно в случае сверточных и выборочных слоев, поскольку их функция перенаправления не зависит от пространственного размера входных данных.

Наиболее популярно использование предобученных нейронных сетей с тонкой настройкой в области распознавания изображений на основе *CNN*.

Среди наиболее популярных и актуальных предобученных моделей *CNN* можно выделить следующие модели:

– оксфордские *VGG*16 и *VGG*19;
– *Google Inception*;
– *Microsoft ResNet*;
– *AlexNet*.

6.2.2. Извлечение дескрипторов со сверточных слоев нейросети

Карта активации, именуемая также картой признаков, отражает результат применения фильтров к входным данным, таким как изображения или иные карты признаков.

Идея визуализации карты признаков для конкретного входного изображения заключается в том, чтобы понять, какие признаки входного изображения обнаруживаются или сохраняются на карте признаков. Предполагается, что карты признаков, близкие к входным данным, обнаруживают мелкие тонкие детали, в то время как карты признаков, близкие к выходным данным модели, захватывают более общие признаки.

Для примера визуализации карты признаков можно воспользоваться предобученной моделью *VGG*16. В качестве входных данных используется изображение птицы '*bird.jpg*' (рис. 6.11).

Рис. 6.11. Пример обрабатываемого изображения

Далее необходимо четкое представление о форме выходных карт признаков по каждому из сверточных слоев и номер индекса слоя для возможности получать соответствующий выходной сигнал слоя.

С помощью примера кода ниже можно получить выходной размер для каждого из сверточных слоев, а также индекс слоя в модели.

```
# Подсчет размера карты признаков для каждого сверточного слоя
from keras.applications.vgg16 import VGG16
from matplotlib import pyplot
# load the model
model = VGG16()
# summarize feature map shapes
for i in range(len(model.layers)):
      layer = model.layers[i]
      # check for convolutional layer
      if 'conv' not in layer.name:continue
      # summarize output shape
      print(i, layer.name, layer.output.shape)
```

Результатом выполнения данного фрагмента кода является размерность карты признаков каждого сверточного слоя.

```
1 block1_conv1 (?, 224, 224, 64)
2 block1_conv2 (?, 224, 224, 64)
4 block2_conv1 (?, 112, 112, 128)
5 block2_conv2 (?, 112, 112, 128)
7 block3_conv1 (?, 56, 56, 256)
8 block3_conv2 (?, 56, 56, 256)
9 block3_conv3 (?, 56, 56, 256)
```

```
11 block4_conv1 (?, 28, 28, 512)
12 block4_conv2 (?, 28, 28, 512)
13 block4_conv3 (?, 28, 28, 512)
15 block5_conv1 (?, 14, 14, 512)
16 block5_conv2 (?, 14, 14, 512)
17 block5_conv3 (?, 14, 14, 512)
```

Для получения карты признаков после загрузки *VGG*-модели следует определить вспомогательную модель, которая выдаст карту признаков из первого сверточного слоя (индекс 1) следующим образом:

```
# Переназначение модели на вывод после первого скрытого слоя
model = Model(inputs=model.inputs, outputs = model.layers[1].output)
```

При формировании прогноза с помощью данной модели будет получена карта признаков для первого сверточного слоя входного изображения.

После определения модели необходимо загрузить изображение птицы с размером, ожидаемым моделью. Далее изображение необходимо преобразовать в массив пикселей *NumPy* и развернуть из 3*D*-массива в 4*D*-массив с размерами [*samples*, *rows*, *cols*, *channels*].

Затем значения пикселей должны быть масштабированы в соответствии с требованиями модели *VGG*.

```
# загрузка изображения в необходимом размере
img = load_img('bird.jpg', target_size=(224, 224))
# преобразование изображения в массив
img = img_to_array(img)
# развертывание массива
img = expand_dims(img, axis=0)
# подготовка изображения
img = preprocess_input(img)
```

После этого карта признаков может быть получена путем вызова функции *model.predict*( ) на подготовленное изображение.

```
# получение карты признаков для первого слоя
feature_maps = model.predict(img)
```

Так как известно, что в результате будет получена карта признаков с размерами 224 × 224 × 64, можно вывести все 64 двумерные изображения в виде квадрата 8 × 8.

```
# plot all 64 maps in an 8x8 squares
square = 8
```

```
ix = 1
for _ in range(square):
      for _ in range(square):
            # specify subplot and turn of axis
            ax = pyplot.subplot(square, square, ix)
            ax.set_xticks([])
            ax.set_yticks([])
            # plot filter channel in grayscale
            pyplot.imshow(feature_maps[0, :, :, ix-1], cmap='gray')
            ix += 1
# show the figure
pyplot.show()
```

В результате создается рисунок, показывающий все 64 карты объектов в виде подгрупп (рис. 6.12).

Рис. 6.12. Изображения карты признаков модели

Подобным образом могут быть выведены карты признаков для других групп сверточных слоев (рис. 6.13).

Легко заметить, что карты признаков, расположенные ближе к входу модели, фиксируют много мелких деталей на изображении, и по мере углубления в модель карты признаков показывают все меньше и меньше деталей.

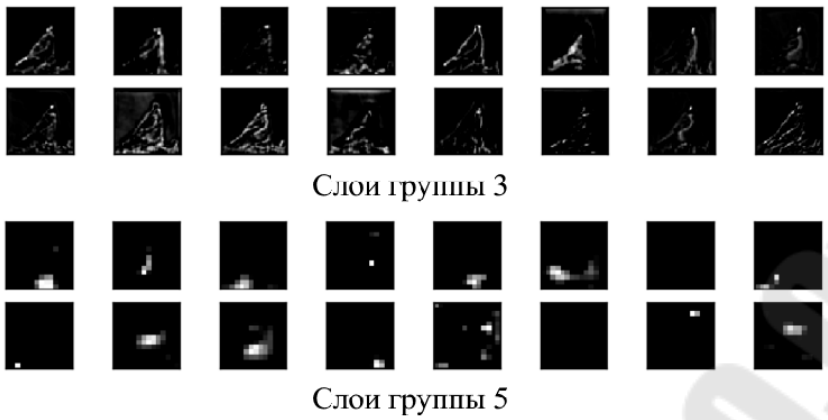


Рис. 6.13. Примеры карт признаков различных слоев модели

Такое поведение ожидаемо, так как модель усредняет признаки из изображения в более общие концепции, которые могут быть использованы для классификации. Из конечного изображения не ясно, что модель видит птицу, и человек в целом теряет возможность интерпретировать эти более глубокие карты признаков.

6.3. Глубокое обучение с подкреплением

Обучение с подкреплением (ОП) – это подход к машинному обучению, предполагающий обучение на практике. В то время как другие методы машинного обучения предполагают пассивную передачу входных данных и обнаружение в них структур, для ОП используются агенты обучения, обеспечивающие активное принятие решений [28] и обучение на собственных результатах.

В обучении с подкреплением существует агент (*agent*), взаимодействующий с окружающей средой (*environment*) и предпринимающий действия (*actions*). Окружающая среда дает награду (*reward*) за эти действия, а агент продолжает их предпринимать [28].

Алгоритмы с частичным обучением пытаются найти стратегию, приписывающую состояниям (*states*) окружающей среды действия, одно из которых может выбрать агент в этих состояниях.

Среда обычно формулируется как марковский процесс принятия решений (МППР) с конечным множеством состояний, и в этом смысле алгоритмы обучения с подкреплением тесно связаны с динамическим программированием. Вероятности выигрышей и перехода состояний в МППР обычно являются величинами случайными, но стационарными в рамках задачи.

При обучении с подкреплением, в отличие от обучения с учителем, не предоставляются верные пары «входные данные – ответ», а принятие субоптимальных решений (дающих локальный экстремум) не ограничивается явно. Обучение с подкреплением пытается найти компромисс между исследованием неизученных областей и применением имеющихся.

Формально простейшая модель обучения с подкреплением состоит из:

– множества состояний окружения (*states*) S;

– множества действий (*actions*) A;

– множества вещественнозначных скалярных «выигрышей» (*rewards*).

В произвольный момент времени t агент характеризуется состоянием $s_t \in S$ и множеством возможных действий $A(s_t)$. Выбирая действие $a \in A(s_t)$, он переходит в состояние $s_t + 1$ и получает выигрыш r_t. Основываясь на таком взаимодействии с окружающей средой, агент, обучающийся с подкреплением, должен выработать стратегию $\pi : S \to A$, которая максимизирует величину $R = r_0 + r_1 + \cdots + r_n$ в случае МППР, имеющего терминальное состояние, или величину

$$R = \sum_t \gamma^t r_t \qquad (6.2)$$

для МППР без терминальных состояний (где $0 \le \gamma \le 1$ – дисконтирующий множитель для «предстоящего выигрыша»).

Таким образом, обучение с подкреплением особенно хорошо подходит для решения задач, связанных с выбором между долгосрочной и краткосрочной выгодой.

Постановка задачи обучения с подкреплением

S – множество состояний среды.

Игра агента со средой:

– инициализация стратегии $\pi_1(a \mid s)$ и состояния среды s_1;

– для всех $t = 1 \ldots T$;

– агент выбирает действие $a_t \sim \pi_t(a \mid s_t)$;

– среда генерирует награду $r_{t+1} \sim p(r \mid a_t, s_t)$ и новое состояние $s_{t+1} \sim p(s \mid a_t, s_t)$;

– агент корректирует стратегию $\pi t + 1(a \mid s)$.

Это МППР, если

$$P(s_t + 1 = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, .., s_1, a_1) =$$
$$= P(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t). \quad (6.3)$$

МППР называется финитным, если $|A| < \infty, |S| < \infty$.

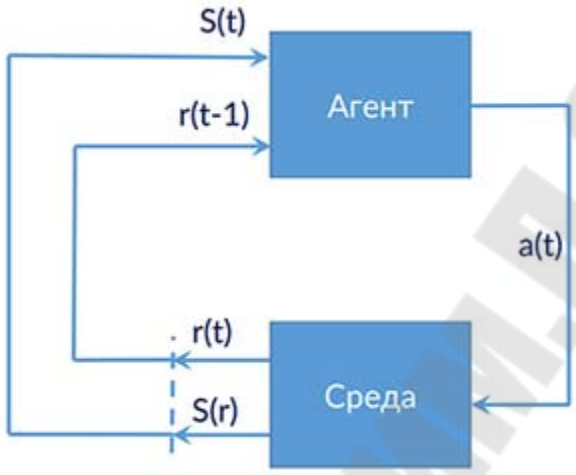


Рис. 6.14. Схема обучения с подкреплением

Схема обучения с подкреплением приведена на рис. 6.14.

6.3.1. Алгоритмы

Теперь, когда определена функция выигрыша, нужно определить алгоритм, который будет использоваться для нахождения стратегии, обеспечивающей наилучший результат.

Наивный подход к решению этой задачи подразумевает следующие шаги:

– опробовать все возможные стратегии;

– выбрать стратегию с наибольшим ожидаемым выигрышем.

Первая проблема такого подхода заключается в том, что количество доступных стратегий может быть очень велико или бесконечно. Вторая проблема возникает, если выигрыши стохастические – чтобы точно оценить выигрыш от каждой стратегии, потребуется многократно применить каждую из них. Этих проблем можно избежать, если допустить некоторую структуризацию и, возможно, позволить результатам, полученным от пробы одной стратегии, влиять на оценку для другой. Двумя основными подходами для реализации этих идей являются оценка функций полезности и прямая оптимизация стратегий.

Подход с использованием функции полезности использует множество оценок ожидаемого выигрыша только для одной стратегии π (либо текущей, либо оптимальной). При этом пытаются оценить либо ожидаемый выигрыш, начиная с состояния *s*, при дальнейшем следовании стратегии π,

$$V(s) = E[R \mid s, \pi], \tag{6.4}$$

либо ожидаемый выигрыш, при принятии решения *a* в состоянии *s* и дальнейшем соблюдении π,

$$Q(s, a) = E[R \mid s, \pi, a]. \tag{6.5}$$

Если для выбора оптимальной стратегии используется функция полезности *Q*, то оптимальные действия всегда можно выбрать как действия, максимизирующие полезность.

Если же мы пользуемся функцией *V*, необходимо либо иметь модель окружения в виде вероятностей $P(s' \mid s, a)$, что позволяет построить функцию полезности вида

$$Q(s, a) = \sum s' V(s') P(s' \mid s, a), \tag{6.6}$$

либо применить так называемый метод «исполнитель–критик», в котором модель делится на две части: критик, оценивающий полезность состояния *V*, и исполнитель, выбирающий подходящее действие в каждом состоянии.

Имея фиксированную стратегию π, оценить $E[R \mid \cdot]$ при $\gamma = 1$ можно, просто усреднив непосредственные выигрыши. Наиболее очевидный способ оценки при $\gamma \in (0,1)$ – усреднить суммарный выигрыш после каждого состояния. Однако для этого требуется, чтобы МППР достиг терминального состояния (завершился).

Поэтому построение искомой оценки при $\gamma \in (0,1)$ неочевидно. Однако можно заметить, что *R* образуют рекурсивное уравнение Беллмана:

$$E[R \mid s_t] = r_t + \gamma E[R \mid s_{t+1}]. \tag{6.7}$$

Подставляя имеющиеся оценки *V* и применяя метод градиентного спуска с квадратичной функцией ошибок, мы приходим к алгоритму обучения с временными воздействиями (*temporal difference* (*TD*) *learning*). В простейшем случае и состояния, и действия дискретны и можно придерживаться табличных оценок для каждого состояния.

6.3.2. Q-обучение (*Q-learning*)

Q-обучение (*Q-learning*) – метод, применяемый в искусственном интеллекте при агентном подходе. Относится к экспериментам вида обучение с подкреплением. На основе получаемого от среды вознаграждения агент формирует функцию полезности *Q*, что впоследствии дает ему возможность уже не случайно выбирать стратегию поведения, а учитывать опыт предыдущего взаимодействия со средой. Одно из преимуществ *Q*-обучения – то, что оно в состоянии сравнить ожидаемую полезность доступных действий, не формируя модели окружающей среды. Применяется для ситуаций, которые можно представить в виде МППР.

Алгоритм *Q-learning*:

1. *Initialization* (Инициализация):

– *for each s and a do Q*[*s*, *a*] = *RND* // инициализируем функцию полезности *Q* от действия *a* в ситуации *s* как случайную для любых входных данных.

2. *Observe* (Наблюдение):

– *s′* = *s* // Запомнить предыдущие состояния;

– *a′* = *a* // Запомнить предыдущие действия;

– *s* = *FROM_SENSOR* // Получить текущие состояния с сенсора;

– *r* = *FROM_SENSOR* // Получить вознаграждение за предыдущее действие.

3. *Update* (Обновление полезности):

– *Q*[*s′*,*a′*] = *Q*[*s′*,*a′*] + *LF* * (*r* + *DF* * *MAX*(*Q*,*s*) – *Q*[*s′*,*a′*]).

4. *Decision* (Выбор действия):

– *a* = *ARGMAX*(*Q*, *s*);

– *TO_ACTIVATOR* = *a.*

5. *Repeat: GO TO* 2.

Обозначения:

LF – это фактор обучения. Чем он выше, тем сильнее агент доверяет новой информации.

DF – это фактор дисконтирования. Чем он меньше, тем меньше агент задумывается о выгоде от будущих своих действий.

Функция *MAX*(*Q*,*s*):

1. *max* = *minValue*

2. *for each a of ACTIONS*(*s*) *do*

i.if Q[*s*, *a*] > *max then max* = *Q*[*s*, *a*]

3. *return max.*

Функция *ARGMAX*(*Q*,*s*):
1. *amax = First of ACTION*(*s*)
2. *for each a of ACTION*(*s*) *do*
   i. if Q[*s*, *a*] > *Q*[*s*, *amax] then amax = a*
3. *return amax.*

6.3.3. Простой policy gradient алгоритм (*REINFORCE*)

В алгоритме *Q-learning* агент обучает функцию полезности действия $Q\theta(s,a)$. Стратегия агента $\pi\theta(a \mid s)$ определяется согласно текущим значениям $Q(s,a)$ с использованием жадного, ε-жадного или *softmax* подхода. Однако существуют методы, которые позволяют оптимизировать стратегию $\pi\theta(s \mid a)$ напрямую. Такие алгоритмы относятся к классу алгоритмов *Policy Gradient*.

Схема алгоритма *Policy Gradient* представлена на рис. 6.15.

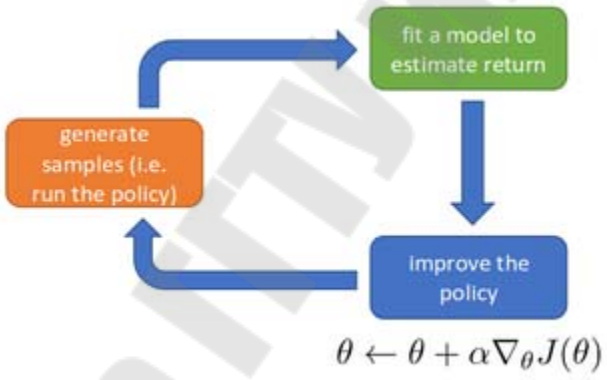


Рис. 6.15. Схема алгоритма *Policy Gradient*

Рассмотрим МППР, имеющий терминальное состояние. Задача – максимизировать сумму всех выигрышей $R = r_0 + r_1 + \cdots + r_T$, где T – шаг, на котором произошел переход в терминальное состояние.

Будем использовать букву τ для обозначения некоторого «сценария» – последовательности состояний и произведенных в них действий: $\tau = (s_1, a_1, s_2, a_2, \ldots s_T, a_T)$. Будем обозначать сумму всех выигрышей, полученных в ходе сценария, как $R_\tau = \sum_\tau r(s_t, a_t)$.

Не все сценарии равновероятны. Вероятность реализации сценария зависит от поведения среды, которое задается вероятностями перехода между состояниями $p(s_{t+1} \mid s_t, a_t)$, распределением начальных состояний $p(s_1)$ и поведения агента, которое определяется его стохас-

тической стратегией $\pi_\theta(a_t \mid s_t)$. Вероятностное распределение над сценариями, таким образом, задается как

$$p_\theta(\tau) = p_\theta(s_1, a_1, \ldots s_T, a_T) = p(s_1)\prod_{t=1}^{T} \pi_\theta(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t). \qquad (6.8)$$

Мы предполагаем, что вероятности переходов между состояниями агенту неизвестны, т. е. у агента нет модели поведения окружающей среды («*model-free learning*»).

Нам нужно выбрать такой набор параметров агента θ, задающий $\pi_\theta(a \mid s)$, чтобы максимизировать матожидание суммы полученных выигрышей:

$$J(\theta) = E_{\tau \sim p_\theta(\tau)}[R_\tau] = \int p_\theta(\tau) R_\tau d\tau. \qquad (6.9)$$

Пусть мы хотим максимизировать функцию $J(\theta)$ методом градиентного подъема. Для этого нам необходимо уметь рассчитывать ее градиент:

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) R_\tau d\tau. \qquad (6.10)$$

Невозможно подсчитать $\nabla_\theta p_\theta(\tau)$ напрямую, потому что в выражение для $p_\theta(\tau)$ входят вероятности переходов между состояниями, которые агенту неизвестны. Однако, так как

$$p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \nabla_\theta p_\theta(\tau), \qquad (6.11)$$

то становится возможным заменить $\nabla_\theta p_\theta(\tau)$ на $p_\theta(\tau) \nabla_\theta \log p_\theta(\tau)$:

$$\nabla_\theta J(\theta) = \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) R_\tau d\tau = E_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) R_\tau]. \qquad (6.12)$$

Рассмотрим $\log p_\theta(\tau)$:

$$\log p_\theta(\tau) = \log\left(p(s_1)\prod_{t=1}^{T} \pi_\theta(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t) \right) =$$

$$= \log p(s_1) + \sum_{t=1}^{T} \left(\log \pi_\theta(a_t \mid s_t) + \log p(s_{t+1} \mid s_t, a_t) \right). \qquad (6.13)$$

Тогда

$$\nabla_\theta \log p_\theta(\tau) = \underbrace{\nabla_\theta \log p(s_1)}_{=0} + \sum_{t=1}^{T}\left(\nabla_\theta \log \pi_\theta(a_t \mid s_t) + \underbrace{\nabla_\theta \log p(s_{t+1} \mid s_t, a_t)}_{=0} \right) =$$

$$= \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t). \qquad (6.14)$$

Подставляя в определение $\nabla_\theta J(\theta)$:

$$\nabla_\theta J(\theta) = E_{\tau \sim p_\theta(\tau)}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right) R_\tau \right]. \qquad (6.15)$$

Заметим, что в получившееся выражение для $\nabla_\theta J(\theta)$ уже не входят напрямую значения $p(s_{t+1} \mid s_t, a_t)$ и $p(s_1)$, которые нам неизвестны. Таким образом, если у нас есть в наличии сценарий τ и соответствующее ему значение R_τ, мы можем вычислить величину $\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right) R_\tau$. Значит, если у нас есть выборка из N уже известных сценариев $\tau^i = \left(s_1^i, a_1^i, \ldots s_{T^i}^i, a_{T^i}^i \right)$, полученная из распределения $\tau \sim p_\theta(\tau)$, то мы можем посчитать приблизительное значение $\nabla_\theta J(\theta)$ по методу Монте-Карло – вычислив выборочное среднее случайной величины:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta\left(a_t^i \mid s_t^i\right) \right) R_{\tau^i} =$$

$$= \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T^i} \nabla_\theta \log \pi_\theta\left(a_t^i \mid s_t^i\right) \right) \left(\sum_{t=1}^{T^i} r\left(s_t^i, a_t^i\right) \right). \qquad (6.16)$$

Осталось понять, как получить несмещенную выборку сценариев τ из вероятностного распределения $p_\theta(\tau)$. Однако это очень просто – нам всего лишь нужно зафиксировать параметр θ и провзаимодействовать со средой, так как распределение $p_\theta(\tau)$ задает именно вероятность реализации сценария τ при взаимодействии агента с фиксированной стратегией со средой.

Таким образом, оптимизировать $J(\theta)$ можно с помощью следующего простого алгоритма (*REINFORCE*):

1. Прогнать N сценариев τ_i со стратегией $\pi_\theta(a \mid s)$.

2. Посчитать среднее арифметическое:

$$\nabla_\theta J(\theta) \leftarrow \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T^i}\nabla_\theta \log \pi_\theta\left(a_t^i \mid s_t^i\right)\right)\left(\sum_{t=1}^{T^i} r\left(s_t^i, a_t^i\right)\right); \qquad (6.17)$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta). \qquad (6.18)$$

Если не сошлись к экстремуму, повторить с п. 1.

6.4. Нечеткие нейронные сети в интеллектуальном анализе данных

Задачей интеллектуального анализа данных является выявление латентных правил и закономерностей в наборах данных.

Метод нейронных сетей используется для классификации, кластеризации, прогнозирования и распознавания образов. Существует множество типов анализа данных, основанных на нейронных сетях, но можно выделить два из них, наиболее популярных. Они основаны на самоорганизующихся нейронных сетях и на нечетких сетях [29].

В основе нечетких нейронных сетей лежит идея использования существующей выборки данных для определения параметров функций принадлежности, выводы делаются на основе аппарата нечеткой логики [31], а для нахождения параметров функций принадлежности используются алгоритмы обучения нейронных сетей. Такие системы могут использовать заранее известную информацию, обучаться, приобретать новые знания, прогнозировать временные ряды, выполнять классификацию образов. Но одним из главных достоинств является наглядность работы такой сети для пользователя.

Важнейшим преимуществом нейронечеткой сети является возможность построения одной сети для вычисления нескольких выходных значений по нескольким входным, а также способность к логическому описанию процессов и ручной корректировке функций принадлежности.

Сети данного типа получили свое название в силу того, что для аппроксимации зависимости выходного сигнала от входного вектора $X = [x_1, x_2, ..., x_N,]^T$ в них используются выражения, заимствованные

из нечетких систем (в частности, из систем Мамдани–Заде и Такаги–Сугено–Канга). Теоретически доказано, что эти выражения позволяют с произвольной точностью аппроксимировать любую непрерывную нелинейную функцию многих переменных суммой функций (называемых нечеткими) одной переменной.

6.4.1. Сеть Такаги–Сугено–Канга

В сети Такаги–Сугено–Канга (сокращенно *TSK*) выходной сигнал рассчитывается с помощью выражения

$$y(X) = \mathrm{sum}[i = 1 : M](w_i * y_i(X)) / \mathrm{sum}[i = 1 : M](w_i), \qquad (6.19)$$

где $y_i(X) = p_{i0} + \mathrm{sum}[j = 1 : N](p_{ij} * x_j)$ – i-й полиномиальный компонент аппроксимации.

Веса w_i компонентов рассчитываются по следующей формуле (с использованием рациональной формы функции Гаусса):

$$w_i = \mathrm{prod}[j = 1 : N](w_{ij}(x_j)) =$$

$$= \mathrm{prod}[j = 1 : N]\left(1 / \left(1 + \left((x_j - c_{ij}) / s_{ij}\right)^{2*b_{ij}}\right)\right). \qquad (6.20)$$

Приведенным выражениям соответствует пятислойная нейронная сеть, структурная схема которой представлена на рис. 6.16.

Первый слой содержит $N * M$ узлов, каждый из которых реализует расчет функции Гаусса с параметрами c_{ij}, s_{ij} и b_{ij}. С точки зрения нечетких систем это слой фуззификации входных переменных. Слой называется параметрическим, поскольку в процессе обучения сети подбору подлежат параметры этого слоя.

Второй слой параметров не содержит. С точки зрения нечетких систем это слой агрегирования левых частей продукций.

Третий слой – генератор (полиномиальных) функций *TSK* $y_i(X)$ и их умножитель на весовой коэффициент w_i. Это параметрический слой, в котором в процессе обучения сети адаптации подвергаются коэффициенты p_{ij}, $i = 1, 2, ..., M$, $j = 0, 1, ..., N$. Общее количество коэффициентов p_{ij} в сети равно $M * (N + 1)$.

Четвертый слой составляют два нейрона-сумматора. Первый раяссчитывает взвешенную сумму сигналов $y_i(X)$, а второй – сумму весов w_i, $i = 1, 2, ..., M$. Это непараметрический слой.

Последний, пятый, слой осуществляет нормализацию весов. Это также непараметрический слой.

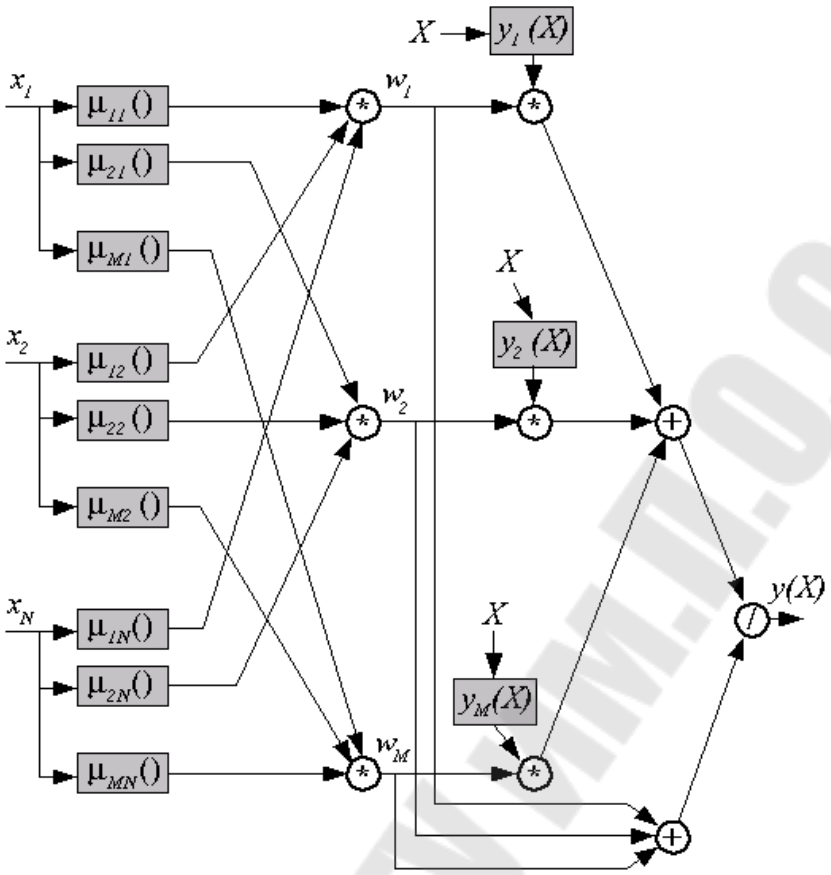


Рис. 6.16. Схема нейронной сети Такаги–Сугено–Канга

Из описания сети *TSK* следует, что она содержит два параметрических слоя (первый и третий), параметры которых подлежат подбору в процессе обучения. Параметры первого слоя будем называть нелинейными, так как они относятся к нелинейной функции, а параметры третьего слоя – линейными.

Общее количество параметров (линейных и нелинейных) сети 1 равно

$$M * 3 * N + M * (N + 1) = M * (4 * N + 1). \qquad (6.21)$$

Во многих практических приложениях это чрезмерная величина, поэтому часто для входных переменных x_j используют ограниченный набор функций $mu(x_j)$, что уменьшает количество нелинейных параметров.

6.4.2. Сеть Ванга–Менделя

В сети данного типа выходной сигнал рассчитывается с помощью выражения

$$y(X)=\frac{\text{sum}[i=1:M](c_i * w_i)}{\text{sum}[i=1:M](w_i)}=$$

$$=\frac{\text{sum}[i=1:M](c_i * \text{prod}[j=1:N](mu_{ij}(x_j)))}{\text{sum}[i=1:M](\text{prod}[j=1:N](mu_{ij}(x_j)))}, \qquad (6.22)$$

где c_i – весовой коэффициент (с точки зрения нечетких систем это центр функции принадлежности правой части продукции); $mu_{ij}()$ – функция Гаусса (в экспоненциальном или рациональном виде) с параметрами центра c_{ij}, ширины s_{ij} и формы b_{ij} (с точки зрения нечетких систем $mu_{ij}()$ – функция принадлежности к нечеткому множеству).

Легко заметить, что выражение для $y(X)$ в сети Ванга–Менделя является частным случаем аналогичного выражения в сети *TSK*, если в последней принять $y(X)=c_i$. Поэтому сеть Ванга–Менделя проще и имеет следующую трехслойную структуру (рис. 6.17).

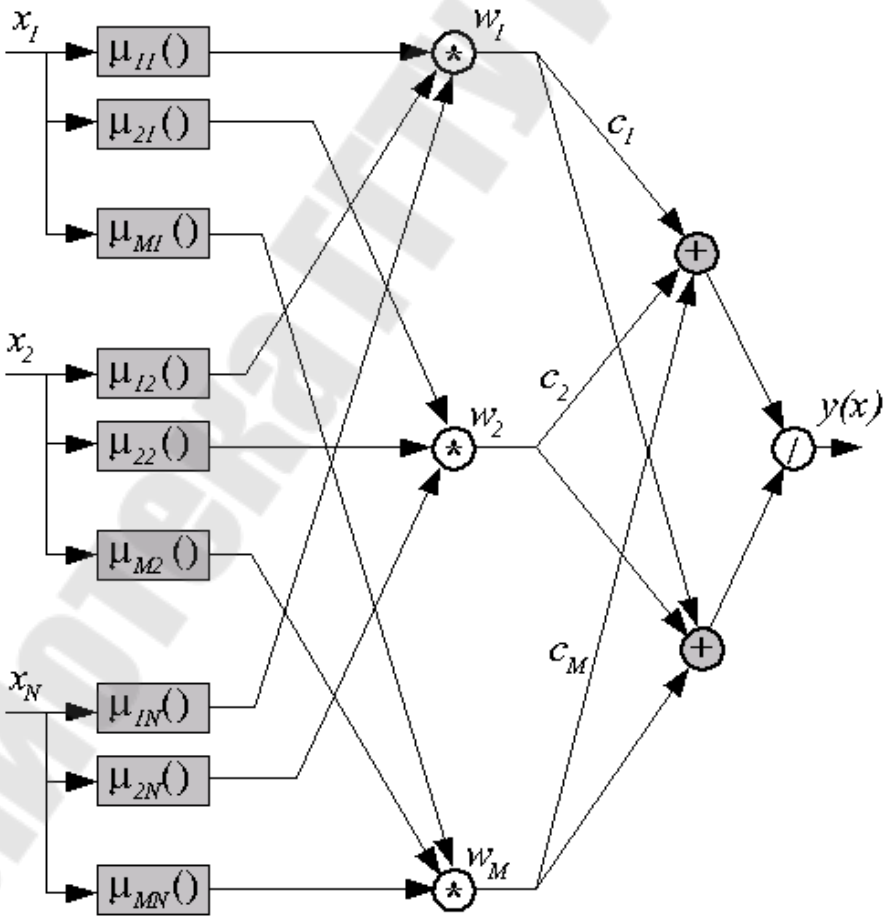


Рис. 6.17. Схема нейронной сети Ванга–Менделя

В данной сети параметрическими являются первый и третий слои. Первый содержит $M * N * 3$ нелинейных параметров функции Гаусса, а третий – M линейных параметров c_i.

Нечеткие нейронные сети (как Ванга–Менделя, так и *TSK*) могут быть обобщены на случай многих выходных переменных. Их обучение, также как и классических сетей, может проводиться как с учителем, так и без оного. Обучение с учителем основано на минимизации целевой функции, определяемой с использованием эвклидовой нормы:

$$E = (1/2) * \mathrm{sum}[k = 1 : p]\left(\left(y\left(X^k\right) - d^k\right)^2\right). \qquad (6.23)$$

Обучение без учителя основано на самоорганизации сети, обеспечивающей кластеризацию входных данных.

6.4.3. Гибридный алгоритм обучения

Данный алгоритм применим к обеим описанным выше структурам, но рассмотрим его касательно сетей *TSK* как более общих. Гибридный алгоритм обучения нечетких сетей можно считать вариантом гибридного алгоритма обучения радиальных сетей.

Алгоритм реализуется чередованием двух этапов:

1. При зафиксированных значениях нелинейных параметров c_{ij}, s_{ij} и b_{ij} первого слоя нейронов отыскиваются значения линейных параметров p_{ij} третьего слоя сети.

2. При зафиксированных значениях линейных параметров p_{ij} третьего слоя уточняются нелинейные параметры c_{ij}, s_{ij} и b_{ij} первого слоя сети.

Этап 1. На данном этапе обучения нелинейные параметры фиксированы. Выходной сигнал определяется как

$$y(X) = \mathrm{sum}[i = 1 : M]\left(w' * \left(p_{i0} + \mathrm{sum}[j = 1 : N]\left(p_{ij} * x_j\right)\right)\right), \qquad (6.24)$$

где $w' = v_i = \dfrac{\mathrm{prod}[j = 1 : N]\left(mu_{ij}\left(x_j\right)\right)}{\mathrm{sum}[l = 1 : M]\left(\mathrm{prod}[j = 1 : N]\left(mu_{lj}\left(x_j\right)\right)\right)} = \mathrm{const}.$

Для K обучающих выборок $< X^k,\ d^k >,\ k = 1, 2, \ldots, K$ получаем систему K линейных уравнений

$$AP = D, \tag{6.25}$$

где $P = [p_{10}, p_{11}, ..., p_{1N}, ..., p_{M0}, p_{M1}, ..., p_{MN}]^T$ – вектор весов третьего слоя сети, а $D = [d^1, d^2, ..., d^k]^T$ – вектор ожидаемых значений, составленный из всех K обучающих выборок. Матрица A представлена ниже:

$$\begin{aligned}
&v_1^1 v_1^1 * x_1^1 \dots v_1^1 * x_N^1 \dots v^1{}_M v^1{}_M * x_1^1 \dots v_M^1 * x_N^1;\\
&v_1^2 v_1^2 * x_1^2 \dots v_1^2 * x_N^2 \dots v_M^2 v_M^2 * x_1^2 \dots v_M^2 * x_N^2;\\
&\dots\dots\dots\\
&v_1^k v_1^k * x_1^k \dots v_1^k * x_N^k \dots v_M^k v_M^k * x_1^k \dots v_M^k * x_N^k.
\end{aligned} \tag{6.26}$$

Количество строк K матрицы A значительно больше количества ее столбцов $M(N+1)$. Решение этой системы линейных алгебраических уравнений может быть получено за один шаг следующим образом:

$$P = A^+ * D, \tag{6.27}$$

где A^+ – псевдоинверсия матрицы A.

Этап 2. Здесь фиксируются значения коэффициентов полиномов третьего слоя и осуществляется уточнение (обычно многократное) коэффициентов функции Гаусса для первого слоя сети стандартным методом градиента:

$$\begin{aligned}
c_{ij}^{k+1} &= c_{ij}^k - nu_c * \partial E^k / \partial c_{ij}^k;\\
s_{ij}^{k+1} &= s_{ij}^k - nu_s * \partial E^k / \partial s_{ij}^k;\\
b_{ij}^{k+1} &= b_{ij}^k - nu_b * \partial E^k / \partial b_{ij}^k,
\end{aligned} \tag{6.28}$$

где k – номер очередного цикла обучения (в режиме «онлайн» он совпадает с номером обучающей выборки). С технической точки зрения получение аналитических выражений для производных целевой функции по нелинейным параметрам проблем не представляет. Однако здесь в силу громоздкости эти выражения не приводятся.

Поскольку в череде этапов этап уточнения нелинейных параметров функции Гаусса имеет много меньшую скорость сходимости, то в ходе обучения реализацию этапа 1, как правило, сопровождает реализация нескольких этапов 2.

6.4.4. Нечеткие сети с самоорганизацией

Сети данного типа на этапе обучения осуществляют группирование входных векторов X^k, $k = 1, 2, ..., p$, в M кластеров, каждый из которых определяется своим центром C_i, $i = 1, 2, ..., M$. На этапе классификации сеть отождествляет очередной входной вектор данных X с одним из ранее определенных кластеров.

Нечеткая сеть с самоорганизацией имеет простую двухслойную структуру (рис. 6.18).

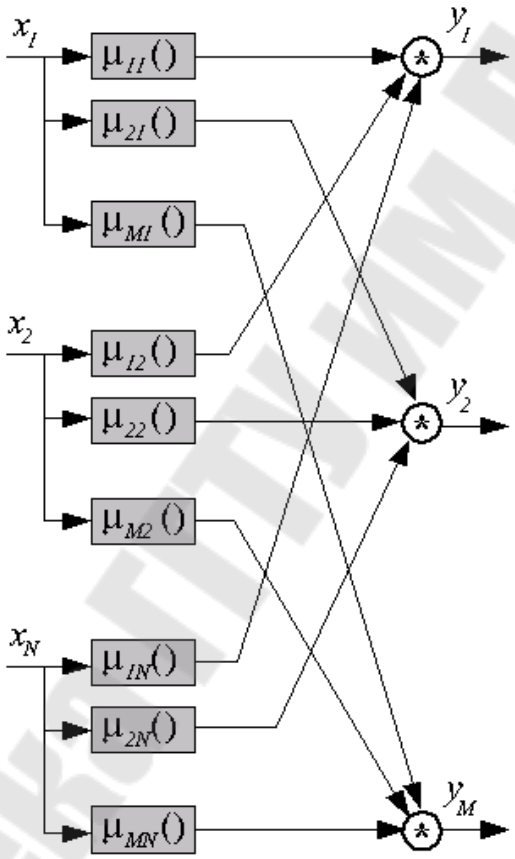


Рис. 6.18. Схема нечеткой нейронной сети с самоорганизацией

Нейроны первого слоя реализуют обобщенную функцию Гаусса в рациональной форме:

$$mu_{ij}(x_j) = 1/\left(1 + \left((x_j - c_{ij})/ s_{ij}\right)^{2b_{ij}}\right). \qquad (6.29)$$

Каждый нейрон второго слоя характеризуется центром $C_i = [ц_{1i}, ц_{2i}, ..., ц_{Ni}]^T$.

6.4.5. Алгоритм нечеткой самоорганизации C-means

В данном алгоритме подаваемый на вход очередной обучающий вектор X^k принадлежит различным кластерам, представленным своими центрами C_i, $i = 1, 2, ..., M$ в степени u_i^k, $0 < u_i^k < 1$, при соблюдении условия

$$\text{sum}[i = 1 : M]\left(u_i^k\right) = 1. \tag{6.30}$$

При этом значение u_i^k тем больше, чем ближе X^k к C_i.

Погрешность соотнесения обучающих векторов X^k и центров C_i для всех p обучающих векторов может быть выражена следующим образом:

$$E = \text{sum}[i = 1 : M]\left(\text{sum}[k = 1 : p]\left(\left(u^k{}_i\right)^m \left|X^k - C_i\right|_2\right)\right), \tag{6.31}$$

где m – показатель, выбираемый из ряда 1, 2, 3, ...

Цель обучения – подбор таких значений центров C_i, которые обеспечивают минимальное значение погрешности E при одновременном соблюдении условия

$$[i = 1 : M]\left(u^k{}_i\right) = 1. \tag{6.32}$$

Решение этой задачи можно свести к минимизации функции Лагранжа в виде

$$LE = \text{sum}[i = 1 : M]\left(\text{sum}[k = 1 : p]\left(\left(u_i^k\right)^m \left|X^k - C_i\right|_2\right)\right) +$$

$$+ \text{sum}[k = 1 : p]\left(L_k\left(\text{sum}[i = 1 : M]\left(u_i^k\right) - 1\right)\right), \tag{6.33}$$

где L_k, $k = 1, 2, ..., p$ – множители Лагранжа.

Доказано, что решение этой задачи можно представить в виде:

$$C_i = \text{sum}[k = 1 : p]\left(\left(u_i^k\right)^m X^k\right) / \text{sum}[k = 1 : p]\left(\left(u_i^k\right)^m\right); \tag{6.34}$$

$$u_i^k = 1 / \text{sum}[l = 1 : M]\left(\left(\left(d_i^k\right)^2 / \left(d_l^k\right)^2\right)\right)^{1/(m-1)}, \tag{6.35}$$

где $d_i^k = \left|X^k - C_i\right|_2$ – эвклидово расстояние между X^k и C_i.

Алгоритм обучения, реализующий описанную выше идею, получил название *C-means*. Он носит итерационный характер и может быть описан следующим образом:

1. Выполнить случайный выбор коэффициентов u_i^k из диапазона [0,1] при соблюдении условия $\mathrm{sum}[i=1:M]\left(u^k{}_i\right)=1$.

2. Вычислить все *M* центров C_i по приведенной выше формуле.

3. Рассчитать значение погрешности *E*. Если это значение меньше установленного порога или незначительно изменилось относительно предыдущей итерации, то закончить вычисления. Иначе перейти к п. 4.

4. Рассчитать новые значения u_i^k по приведенной выше формуле и перейти к п. 2.

Описанный выше итерационный алгоритм ведет к достижению минимума погрешности *E*, который, однако, необязательно будет глобальным минимумом. На вероятность отыскания глобального минимума влияет выбор начальных значений u_i^k и C_i. Специально для подбора «хороших» начальных значений центров C_i разработаны процедуры инициализации, две из которых представлены ниже.

6.4.6. Алгоритм пикового группирования

Для отыскания «первого приближения» к наилучшему расположению центров C_i в данном алгоритме используются так называемые пиковые функции. При подаче на вход сети *p* обучающих векторов X^k создается равномерная сетка, покрывающая все пространство, занимаемое данными векторами.

Узлы сетки обозначим как V_l, для каждого из них рассчитывается значение пиковой функции

$$m(V_l)=\mathrm{sum}[k=1:p]\left(\exp\left(-\left(\left|X^k-V_l\right|_2{}^{2b}/\left(2s^2\right)\right)\right)\right), \qquad (6.36)$$

где *s* – константа, индивидуально подбираемая для каждой задачи.

Значение $m(V_l)$ пропорционально количеству обучающих векторов X^k, находящихся в окрестности потенциального центра V_l. Малое значение $m(V_l)$ говорит о том, что V_l находится в области, где количество векторов X^k мало. Следует отметить, что коэффициент *s* оказывает незначительное влияние на соотношение значений V_l для

разных узлов сетки, поэтому подбор его величины не является критичным.

После расчета $m(V_l)$ для всех потенциальных центров (узлов сетки) отбирается узел, имеющий наибольшее значение пиковой функции. С этим узлом отождествляется первый центр C_1. Для выбора аналогичным образом следующего центра из рассмотрения исключается центр C_1 и соседние с ним узлы сетки. Это удобно сделать переопределением пиковой функции

$$m_{new}(V_l) = m(V_l) - m(C_1)\exp\left(-\left(|V_l - C_1|^{2b} / \left(2s^2\right)\right)\right), \qquad (6.37)$$

где $m(C_1)$ – значение пиковой функции в центре C_1.

Процесс последовательного отыскания центров C_1, C_2, C_3, ... завершается после обнаружения центра *CM*.

Основной недостаток алгоритма пикового группирования – экспоненциальный рост сложности с увеличением размерности векторов входных данных X^k. Следовательно, он применим лишь при небольшом количестве входных сигналов *N*. Представленный далее алгоритм также имеет экспоненциальный рост сложности, но это рост в зависимости от количества обучающих выборок *p*.

6.4.7. Алгоритм разностного группирования

В этом алгоритме в качестве потенциальных центров рассматриваются обучающие векторы X^k, $k = 1, 2, ..., p$. Пиковая функция $m\left(X^i\right)$ определяется в следующем виде:

$$m\left(X^i\right) = \text{sum}[k = 1 : p]\left(\exp\left(-\left(\left|X^k - X^i\right|^{2b} / \left(r_1 / 2\right)^2\right)\right)\right), \qquad (6.38)$$

где значение коэффициента r_1 определяет размер сферы соседства. При большой плотности входных векторов вокруг X^i значение функции велико, и, напротив, малое значение $m\left(X^i\right)$ свидетельствует о незначительном количестве соседей.

После расчета значений $m\left(X^i\right)$ для всех входных векторов в качестве первого центра C_1 принимается X^i с наибольшим значением пиковой функции.

Для отыскания второго центра используется модифицированная пиковая функция в виде

$$m_{new}\left(X^i\right) = m\left(X^i\right) - m(C_1)\exp\left(-\left(\left|X^i - C_1\right|_2^{2b} / (r_2/2)^2\right)\right), \qquad (6.39)$$

где r_2 задает новый размер сферы соседства, обычно $r_2 >= r_1$.

Пиковая функция $m_{new}\left(X^i\right)$ принимает нулевое значение для $X^i = C_1$.

Литература

1. Бишоп, К. М. Распознавание образов и машинное обучение / К. М. Бишоп. – М. : Вильямс, 2020. – 960 с.

2. Барский, А. Б. Нейронные сети: распознавание, управление, принятие решений / А. Б. Барский. – М. : Финансы и статистика, 2004. – 176 с.

3. Вентцель, Е. С. Теория случайных процессов и ее инженерные приложения / Е. С. Вентцель, Л. А. Овчаров. – М. : КНОРУС, 2013. – 448 с.

4. Розенблатт, Ф. Принципы нейродинамики. Перцептрон и теория механизмов мозга / Ф. Розенблатт. – М. : Мир, 1965. – 480 с.

5. Хайкин, С. Нейронные сети: полный курс / С. Хайкин. – 2-е изд. – М. : Вильямс, 2006. – 1104 с.

6. Аксенов, С. В. Организация и использование нейронных сетей (методы и технологии) / С. В. Аксенов, В. Б. Новосельцев. – Томск : НТЛ, 2006. – 128 с.

7. Галушкин, А. И. Теория нейронных сетей : учеб. пособие для вузов / А. И. Галушкин. – М. : Радиотехника, 2000. – 415 с.

8. Круглов, В. В. Искусственные нейронные сети. Теория и практика / В. В. Круглов, В. В. Борисов. – М. : Горячая линия-Телеком, 2002. – 382 с.

9. Гудфеллоу, Я. Глубокое обучение / Я. Гудфеллоу, И. Бенджио, А. Курвилль. – М. : ДМК-Пресс, 2018. – 652 с.

10. Рашид, Т. Создаем нейронную сеть / Т. Рашид. – М. : Вильямс, 2017. – 272 с.

11. Максимей, И. В. Математическое моделирование больших систем : учеб. пособие для специальности «Прикладная математика» / И. В. Максимей. – Минск : Выш. шк., 1985. – 118 с.

12. Заенцев, И. В. Нейронные сети. Основные модели / И. В. Заенцев. – Воронеж : ВГУ, 1999. – 76 с.

13. Мюллер, А. Введение в машинное обучение с помощью Python. Руководство для специалистов по работе с данными / А. Мюллер, С. Гвидо. – М. : Вильямс, 2017. – 480 с.

14. Макаренко, С. И. Интеллектуальные информационные системы : учеб. пособие / С. И. Макаренко. – Ставрополь : СФ МГГУ им. М. А. Шолохова, 2009. – 206 с.

15. Ширяев, В. И. Финансовые рынки, нейронные сети, хаос и нелинейная динамика / В. И. Ширяев. – М. : Ленанд, 2019. – 232 с.

16. Черняк, Е. Введение в глубокое обучение / Е. Черняк. – СПб. : Диалектика, 2020. – 192 с.

17. Шакла, Н. Машинное обучение и TensorFlow / Н. Шакла. – СПб. : Питер, 2019. – 336 с.

18. Орельен, Ж. Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow / Ж. Орельен. – СПб. : Диалектика, 2019. – 683 с.

19. Аггарвал, Ч. Нейронные сети и глубокое обучение : учеб. курс / Ч. Аггарвал. – СПб. : Диалектика, 2020. – 752 с.

20. Траск, Э. Грокаем глубокое обучение / Э. Траск. – СПб. : Питер, 2019. – 352 с.

21. Микелуччи, У. Прикладное глубокое обучение: Подход к пониманию глубоких нейронных сетей на основе метода кейсов / У. Микелуччи. – СПб. : БХВ-Петербург, 2020. – 368 с.

22. Джулли, А. Библиотека Keras – инструмент глубокого обучения / А. Джулли, С. Пал. – М. : ДМК-Пресс, 2017. – 294 с.

23. Дейтел, П. Python: Искусственный интеллект, большие данные и облачные вычисления / П. Дейтел, Х. Дейтел. – СПб. : Питер, 2020. – 864 с.

24. Шолле Ф. Глубокое обучение на Python / Ф. Шолле. – СПб. : Питер, 2018. – 400 с.

25. Плас, Дж. Python для сложных задач. Наука о данных и машинное обучение / Дж. Плас. – СПб. : Питер, 2019. – 576 с.

26. Николенко, С. Глубокое обучение. Погружение в мир нейронных сетей / С. Николенко, Е. Архангельская, А. Кадурин. – СПб. : Питер, 2018. – 480 с.

27. Коломоец, Ф. Г. Основы системного анализа и теория принятия решений : пособие для исследователей, управленцев и студентов вузов / Ф. Г. Коломоец. – Минск : Тесей, 2006. – 315 с.

28. Саттон Р. С. Обучение с подкреплением / Р. С. Саттон, Э. Г. Барто. – М. : БИНОМ, 2011. – 399 с.

29. Яхъяева, Г. Э. Нечеткие множества и нейронные сети : учеб. пособие / Г. Э. Яхъяева. – М. : Интернет-Ун-т информ. технологий, 2008. – 316 с.

30. Змитрович, А. И. Интеллектуальные информационные системы / А. И. Змитрович. – Минск : ТетраСистемс, 1997. – 368 с.

31. Курочка, К. С. Локализация позвонков человека на рентгеновских изображениях с использованием Darknet yolo / К. С. Курочка, Т. В. Лучшева, К. А. Панарин // Докл. БГУИР. – Минск, 2018. – № 3 (113). – С. 32–38.

ПРИЛОЖЕНИЯ

Приложение 1

Исходный код *LeNet5*

```
     #!/usr/bin/python3
     import numpy as np
     import tensorflow as tf
     tf.logging.set_verbosity(tf.logging.INFO)
     # Load trainig and eval data
     mnist = tf.contrib.learn.datasets.load_dataset("mnist")
     train_data = mnist.train.images
     train_labels = np.asarray(mnist.train.labels,
dtype=np.int32)
     eval_data = mnist.test.images
     eval_labels = np.asarray(mnist.test.labels,
dtype=np.int32)
     def cnn_model_fn(features, labels, mode):

     """Model function for CNN"""
     # Input layer
     input_layer = tf.reshape(features["x"], [-1,28,28,1])
     # Convolutional Layer #1
     # Input 28x28x1
     # Output 28x28x6
     conv1 = tf.layers.conv2d(
     inputs=input_layer,
     filters=6,
     kernel_size=[5,5],
     padding='same',
     activation=tf.nn.tanh)
     # Pooling Layer #1
     # Output 14x14x6
     pool1 = tf.layers.max_pooling2d(inputs=conv1,
pool_size=[2,2], strides=2)
     # Convolutional Layer #2
     # Output 10x10x16
     conv2 = tf.layers.conv2d(
     inputs=pool1,
     filters=16,
     kernel_size=[5,5],
     padding='valid',
     activation=tf.nn.tanh)
     # Pooling Layer #2
     # Output 5x5x16
     pool2 = tf.layers.max_pooling2d(inputs=conv2,
pool_size=[2,2], strides=2)
     # Dense layers
```

```
    pool2_flat = tf.reshape(pool2, [-1, 16 * 5 * 5])
    dense1 = tf.layers.dense(
    inputs=pool2_flat,
    units=120,
    activation=tf.nn.tanh)
    dense2 = tf.layers.dense(inputs=dense1, units=84, activa-
tion=tf.nn.tanh)
    # Logits layer
    logits = tf.layers.dense(inputs=dense2, units=10)
    predictions = {
    # Generate predictions (for both TRAIN and EVAL mode)
    "classes": tf.argmax(input=logits, axis=1),
    # Add 'softmax-tensor to the graph. It is used for PREDICT
and by
    # 'logging_hook'
    "probabilities": tf.nn.softmax(logits,
name="softmax_tensor")
    }
    if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predic-
tions=predictions)
    # Calculate loss (for both TRAIN and EVAL modes)
    loss =
tf.losses.sparse_softmax_cross_entropy(labels=labels,
logits=logits)
    # Configure the Training Op (for TRAIN mode)
    if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer =
tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
    loss=loss,
    global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(
    mode=mode,
    loss=loss,
    train_op=train_op)
    # Add evaluation metrics (for EVAL mode)
    eval_metric_ops = {
    "accuracy": tf.metrics.accuracy(
    labels=labels,
    predictions=predictions["classes"])}
    return tf.estimator.EstimatorSpec(
    mode=mode,
    loss=loss,
    eval_metric_ops=eval_metric_ops)
    def run_model():
    mnist_classifier = tf.estimator.Estimator(
    model_fn=cnn_model_fn,
    model_dir='/tmp/lenet5')
    # Setup a logging hook for predictions
```

```
    tensor_to_log = {"probabilities": "softmax_tensor"}
    logging_hook = tf.train.LoggingTensorHook(
    tensors=tensor_to_log,
    every_n_iter=50)
    # Train the model
    train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x = {"x": train_data},
    y = train_labels,
    batch_size = 100,
    num_epochs = None,
    shuffle = True)
    mnist_classifier.train(
    input_fn = train_input_fn,
    steps = 50000,
    hooks = [logging_hook])
    # Evaluate the model and print results
    eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    x = {"x": eval_data},
    y = eval_labels,
    num_epochs = 1,
    shuffle = False)
    eval_results =
mnist_classifier.evaluate(input_fn=eval_input_fn)
    print(eval_results)
    if __name__ == "__main__":
    run_model()
```

Приложение 2

Исходный код *AlexNet*

```
"""
AlexNet Keras Implementation

BibTeX Citation:

@inproceedings{krizhevsky2012imagenet,
  title={Imagenet classification with deep convolutional neu-
ral networks},
  author={Krizhevsky, Alex and Sutskever, Ilya and Hinton,
Geoffrey E},
  booktitle={Advances in neural information processing sys-
tems},
  pages={1097--1105},
  year={2012}
}
"""

# Import necessary packages
import argparse

# Import necessary components to build LeNet
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation,
Flatten
from keras.layers.convolutional import Conv2D, MaxPooling2D,
ZeroPadding2D
from keras.layers.normalization import BatchNormalization
from keras.regularizers import l2

def alexnet_model(img_shape=(224, 224, 3), n_classes=10,
l2_reg=0.,
      weights=None):

      # Initialize model
      alexnet = Sequential()

      # Layer 1
      alexnet.add(Conv2D(96, (11, 11), input_shape=img_shape,
            padding='same', kernel_regularizer=l2(l2_reg)))
      alexnet.add(BatchNormalization())
      alexnet.add(Activation('relu'))
      alexnet.add(MaxPooling2D(pool_size=(2, 2)))

      # Layer 2
      alexnet.add(Conv2D(256, (5, 5), padding='same'))
      alexnet.add(BatchNormalization())
```

```
    alexnet.add(Activation('relu'))
    alexnet.add(MaxPooling2D(pool_size=(2, 2)))

    # Layer 3
    alexnet.add(ZeroPadding2D((1, 1)))
    alexnet.add(Conv2D(512, (3, 3), padding='same'))
    alexnet.add(BatchNormalization())
    alexnet.add(Activation('relu'))
    alexnet.add(MaxPooling2D(pool_size=(2, 2)))

    # Layer 4
    alexnet.add(ZeroPadding2D((1, 1)))
    alexnet.add(Conv2D(1024, (3, 3), padding='same'))
    alexnet.add(BatchNormalization())
    alexnet.add(Activation('relu'))

    # Layer 5
    alexnet.add(ZeroPadding2D((1, 1)))
    alexnet.add(Conv2D(1024, (3, 3), padding='same'))
    alexnet.add(BatchNormalization())
    alexnet.add(Activation('relu'))
    alexnet.add(MaxPooling2D(pool_size=(2, 2)))

    # Layer 6
    alexnet.add(Flatten())
    alexnet.add(Dense(3072))
    alexnet.add(BatchNormalization())
    alexnet.add(Activation('relu'))
    alexnet.add(Dropout(0.5))

    # Layer 7
    alexnet.add(Dense(4096))
    alexnet.add(BatchNormalization())
    alexnet.add(Activation('relu'))
    alexnet.add(Dropout(0.5))

    # Layer 8
    alexnet.add(Dense(n_classes))
    alexnet.add(BatchNormalization())
    alexnet.add(Activation('softmax'))

    if weights is not None:
        alexnet.load_weights(weights)

    return alexnet

def parse_args():
    """
    Parse command line arguments.
```

```
        Parameters:
                None
        Returns:
                parser arguments
        """
        parser = argparse.ArgumentParser(description='AlexNet
model')
        optional = parser._action_groups.pop()
        required = parser.add_argument_group('required argu-
ments')
        optional.add_argument('--print_model',
                dest='print_model',
                help='Print AlexNet model',
                action='store_true')
        parser._action_groups.append(optional)
        return parser.parse_args()

if __name__ == "__main__":
        # Command line parameters
        args = parse_args()

        # Create AlexNet model
        model = alexnet_model()

        # Print
        if args.print_model:
                model.summary()
```

Приложение 3

Исходный код *VGG*16

```
from keras.models import Sequential
from keras.layers.core import Flatten, Dense, Dropout
from keras.layers.convolutional import Convolution2D, MaxPool-
ing2D, ZeroPadding2D
from keras.optimizers import SGD
import cv2, numpy as np

def VGG_16(weights_path=None):
    model = Sequential()
    model.add(ZeroPadding2D((1,1),input_shape=(3,224,224)))
    model.add(Convolution2D(64, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(64, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(128, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(128, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))
```

```
    model.add(Flatten())
    model.add(Dense(4096, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(4096, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1000, activation='softmax'))

    if weights_path:
        model.load_weights(weights_path)

    return model

if __name__ == "__main__":
    im = cv2.resize(cv2.imread('cat.jpg'), (224,
224)).astype(np.float32)
    im[:,:,0] -= 103.939
    im[:,:,1] -= 116.779
    im[:,:,2] -= 123.68
    im = im.transpose((2,0,1))
    im = np.expand_dims(im, axis=0)

    # Test pretrained model
    model = VGG_16('vgg16_weights.h5')
    sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(optimizer=sgd,
loss='categorical_crossentropy')
    out = model.predict(im)
    print np.argmax(out)
```

Приложение 4

Исходный код *Inception V3*

```
# -*- coding: utf-8 -*-
"""Inception V3 model for Keras.
Note that the input image format for this model is different
than for
the VGG16 and ResNet models (299x299 instead of 224x224),
and that the input preprocessing function is also different
(same as Xception).
# Reference
- [Rethinking the Inception Architecture for Computer Vi-
sion](http://arxiv.org/abs/1512.00567)
"""
from __future__ import print_function
from __future__ import absolute_import

import warnings
import numpy as np

from keras.models import Model
from keras import layers
from keras.layers import Activation
from keras.layers import Dense
from keras.layers import Input
from keras.layers import BatchNormalization
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import AveragePooling2D
from keras.layers import GlobalAveragePooling2D
from keras.layers import GlobalMaxPooling2D
from keras.engine.topology import get_source_inputs
from keras.utils.layer_utils import con-
vert_all_kernels_in_model
from keras.utils.data_utils import get_file
from keras import backend as K
from keras.applications.imagenet_utils import de-
code_predictions
from keras.applications.imagenet_utils import
_obtain_input_shape
from keras.preprocessing import image


WEIGHTS_PATH = 'https://github.com/fchollet/deep-learning-
mod-
els/releases/download/v0.5/inception_v3_weights_tf_dim_orderin
g_tf_kernels.h5'
```

```
WEIGHTS_PATH_NO_TOP = 'https://github.com/fchollet/deep-
learning-
mod-
els/releases/download/v0.5/inception_v3_weights_tf_dim_orderin
g_tf_kernels_notop.h5'


def conv2d_bn(x,
              filters,
              num_row,
              num_col,
              padding='same',
              strides=(1, 1),
              name=None):
    """Utility function to apply conv + BN.
    Arguments:
        x: input tensor.
        filters: filters in `Conv2D`.
        num_row: height of the convolution kernel.
        num_col: width of the convolution kernel.
        padding: padding mode in `Conv2D`.
        strides: strides in `Conv2D`.
        name: name of the ops; will become `name + '_conv'`
            for the convolution and `name + '_bn'` for the
            batch norm layer.
    Returns:
        Output tensor after applying `Conv2D` and `BatchNor-
malization`.
    """
    if name is not None:
        bn_name = name + '_bn'
        conv_name = name + '_conv'
    else:
        bn_name = None
        conv_name = None
    if K.image_data_format() == 'channels_first':
        bn_axis = 1
    else:
        bn_axis = 3
    x = Conv2D(
        filters, (num_row, num_col),
        strides=strides,
        padding=padding,
        use_bias=False,
        name=conv_name)(x)
    x = BatchNormalization(axis=bn_axis, scale=False,
name=bn_name)(x)
    x = Activation('relu', name=name)(x)
    return x
```

```
def InceptionV3(include_top=True,
                weights='imagenet',
                input_tensor=None,
                input_shape=None,
                pooling=None,
                classes=1000):
    """Instantiates the Inception v3 architecture.
    Optionally loads weights pre-trained
    on ImageNet. Note that when using TensorFlow,
    for best performance you should set
    `image_data_format="channels_last"` in your Keras config
    at ~/.keras/keras.json.
    The model and the weights are compatible with both
    TensorFlow and Theano. The data format
    convention used by the model is the one
    specified in your Keras config file.
    Note that the default input image size for this model is
299x299.
    Arguments:
        include_top: whether to include the fully-connected
            layer at the top of the network.
        weights: one of `None` (random initialization)
            or "imagenet" (pre-training on ImageNet).
        input_tensor: optional Keras tensor (i.e. output of
`layers.Input()`)
            to use as image input for the model.
        input_shape: optional shape tuple, only to be speci-
fied
            if `include_top` is False (otherwise the input
shape
            has to be `(299, 299, 3)` (with `channels_last`
data format)
            or `(3, 299, 299)` (with `channels_first` data
format).
            It should have exactly 3 inputs channels,
            and width and height should be no smaller than
139.
            E.g. `(150, 150, 3)` would be one valid value.
        pooling: Optional pooling mode for feature extraction
            when `include_top` is `False`.
            - `None` means that the output of the model will
be
                the 4D tensor output of the
                last convolutional layer.
            - `avg` means that global average pooling
                will be applied to the output of the
                last convolutional layer, and thus
                the output of the model will be a 2D tensor.
            - `max` means that global max pooling will
                be applied.
```

```
        classes: optional number of classes to classify images
            into, only to be specified if `include_top` is
True, and
            if no `weights` argument is specified.
    Returns:
        A Keras model instance.
    Raises:
        ValueError: in case of invalid argument for `weights`,
            or invalid input shape.
    """
    if weights not in {'imagenet', None}:
        raise ValueError('The `weights` argument should be ei-
ther '
                         '`None` (random initialization) or
`imagenet` '
                         '(pre-training on ImageNet).')

    if weights == 'imagenet' and include_top and classes !=
1000:
        raise ValueError('If using `weights` as imagenet with
`include_top`'
                         ' as true, `classes` should be 1000')

    # Determine proper input shape
    input_shape = _obtain_input_shape(
        input_shape,
        default_size=299,
        min_size=139,
        data_format=K.image_data_format(),
        include_top=include_top)

    if input_tensor is None:
        img_input = Input(shape=input_shape)
    else:
        img_input = Input(tensor=input_tensor,
shape=input_shape)

    if K.image_data_format() == 'channels_first':
        channel_axis = 1
    else:
        channel_axis = 3

    x = conv2d_bn(img_input, 32, 3, 3, strides=(2, 2), pad-
ding='valid')
    x = conv2d_bn(x, 32, 3, 3, padding='valid')
    x = conv2d_bn(x, 64, 3, 3)
    x = MaxPooling2D((3, 3), strides=(2, 2))(x)

    x = conv2d_bn(x, 80, 1, 1, padding='valid')
    x = conv2d_bn(x, 192, 3, 3, padding='valid')
```

```
    x = MaxPooling2D((3, 3), strides=(2, 2))(x)

    # mixed 0, 1, 2: 35 x 35 x 256
    branch1x1 = conv2d_bn(x, 64, 1, 1)

    branch5x5 = conv2d_bn(x, 48, 1, 1)
    branch5x5 = conv2d_bn(branch5x5, 64, 5, 5)

    branch3x3dbl = conv2d_bn(x, 64, 1, 1)
    branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
    branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)

    branch_pool = AveragePooling2D((3, 3), strides=(1, 1),
padding='same')(x)
    branch_pool = conv2d_bn(branch_pool, 32, 1, 1)
    x = layers.concatenate(
        [branch1x1, branch5x5, branch3x3dbl, branch_pool],
        axis=channel_axis,
        name='mixed0')

    # mixed 1: 35 x 35 x 256
    branch1x1 = conv2d_bn(x, 64, 1, 1)

    branch5x5 = conv2d_bn(x, 48, 1, 1)
    branch5x5 = conv2d_bn(branch5x5, 64, 5, 5)

    branch3x3dbl = conv2d_bn(x, 64, 1, 1)
    branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
    branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)

    branch_pool = AveragePooling2D((3, 3), strides=(1, 1),
padding='same')(x)
    branch_pool = conv2d_bn(branch_pool, 64, 1, 1)
    x = layers.concatenate(
        [branch1x1, branch5x5, branch3x3dbl, branch_pool],
        axis=channel_axis,
        name='mixed1')

    # mixed 2: 35 x 35 x 256
    branch1x1 = conv2d_bn(x, 64, 1, 1)

    branch5x5 = conv2d_bn(x, 48, 1, 1)
    branch5x5 = conv2d_bn(branch5x5, 64, 5, 5)

    branch3x3dbl = conv2d_bn(x, 64, 1, 1)
    branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
    branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)

    branch_pool = AveragePooling2D((3, 3), strides=(1, 1),
padding='same')(x)
```

```
    branch_pool = conv2d_bn(branch_pool, 64, 1, 1)
    x = layers.concatenate(
        [branch1x1, branch5x5, branch3x3dbl, branch_pool],
        axis=channel_axis,
        name='mixed2')

    # mixed 3: 17 x 17 x 768
    branch3x3 = conv2d_bn(x, 384, 3, 3, strides=(2, 2), pad-
ding='valid')

    branch3x3dbl = conv2d_bn(x, 64, 1, 1)
    branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
    branch3x3dbl = conv2d_bn(
        branch3x3dbl, 96, 3, 3, strides=(2, 2), pad-
ding='valid')

    branch_pool = MaxPooling2D((3, 3), strides=(2, 2))(x)
    x = layers.concatenate(
        [branch3x3, branch3x3dbl, branch_pool],
axis=channel_axis, name='mixed3')

    # mixed 4: 17 x 17 x 768
    branch1x1 = conv2d_bn(x, 192, 1, 1)

    branch7x7 = conv2d_bn(x, 128, 1, 1)
    branch7x7 = conv2d_bn(branch7x7, 128, 1, 7)
    branch7x7 = conv2d_bn(branch7x7, 192, 7, 1)

    branch7x7dbl = conv2d_bn(x, 128, 1, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 128, 7, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 128, 1, 7)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 128, 7, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)

    branch_pool = AveragePooling2D((3, 3), strides=(1, 1),
padding='same')(x)
    branch_pool = conv2d_bn(branch_pool, 192, 1, 1)
    x = layers.concatenate(
        [branch1x1, branch7x7, branch7x7dbl, branch_pool],
        axis=channel_axis,
        name='mixed4')

    # mixed 5, 6: 17 x 17 x 768
    for i in range(2):
        branch1x1 = conv2d_bn(x, 192, 1, 1)

        branch7x7 = conv2d_bn(x, 160, 1, 1)
        branch7x7 = conv2d_bn(branch7x7, 160, 1, 7)
        branch7x7 = conv2d_bn(branch7x7, 192, 7, 1)
```

```
        branch7x7dbl = conv2d_bn(x, 160, 1, 1)
        branch7x7dbl = conv2d_bn(branch7x7dbl, 160, 7, 1)
        branch7x7dbl = conv2d_bn(branch7x7dbl, 160, 1, 7)
        branch7x7dbl = conv2d_bn(branch7x7dbl, 160, 7, 1)
        branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)

        branch_pool = AveragePooling2D(
            (3, 3), strides=(1, 1), padding='same')(x)
        branch_pool = conv2d_bn(branch_pool, 192, 1, 1)
        x = layers.concatenate(
            [branch1x1, branch7x7, branch7x7dbl, branch_pool],
            axis=channel_axis,
            name='mixed' + str(5 + i))

    # mixed 7: 17 x 17 x 768
    branch1x1 = conv2d_bn(x, 192, 1, 1)

    branch7x7 = conv2d_bn(x, 192, 1, 1)
    branch7x7 = conv2d_bn(branch7x7, 192, 1, 7)
    branch7x7 = conv2d_bn(branch7x7, 192, 7, 1)

    branch7x7dbl = conv2d_bn(x, 192, 1, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 7, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 7, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)

    branch_pool = AveragePooling2D((3, 3), strides=(1, 1),
padding='same')(x)
    branch_pool = conv2d_bn(branch_pool, 192, 1, 1)
    x = layers.concatenate(
        [branch1x1, branch7x7, branch7x7dbl, branch_pool],
        axis=channel_axis,
        name='mixed7')

    # mixed 8: 8 x 8 x 1280
    branch3x3 = conv2d_bn(x, 192, 1, 1)
    branch3x3 = conv2d_bn(branch3x3, 320, 3, 3,
                          strides=(2, 2), padding='valid')

    branch7x7x3 = conv2d_bn(x, 192, 1, 1)
    branch7x7x3 = conv2d_bn(branch7x7x3, 192, 1, 7)
    branch7x7x3 = conv2d_bn(branch7x7x3, 192, 7, 1)
    branch7x7x3 = conv2d_bn(
        branch7x7x3, 192, 3, 3, strides=(2, 2), pad-
ding='valid')

    branch_pool = MaxPooling2D((3, 3), strides=(2, 2))(x)
    x = layers.concatenate(
```

```
        [branch3x3, branch7x7x3, branch_pool],
axis=channel_axis, name='mixed8')

    # mixed 9: 8 x 8 x 2048
    for i in range(2):
        branch1x1 = conv2d_bn(x, 320, 1, 1)

        branch3x3 = conv2d_bn(x, 384, 1, 1)
        branch3x3_1 = conv2d_bn(branch3x3, 384, 1, 3)
        branch3x3_2 = conv2d_bn(branch3x3, 384, 3, 1)
        branch3x3 = layers.concatenate(
            [branch3x3_1, branch3x3_2], axis=channel_axis,
name='mixed9_' + str(i))

        branch3x3dbl = conv2d_bn(x, 448, 1, 1)
        branch3x3dbl = conv2d_bn(branch3x3dbl, 384, 3, 3)
        branch3x3dbl_1 = conv2d_bn(branch3x3dbl, 384, 1, 3)
        branch3x3dbl_2 = conv2d_bn(branch3x3dbl, 384, 3, 1)
        branch3x3dbl = layers.concatenate(
            [branch3x3dbl_1, branch3x3dbl_2],
axis=channel_axis)

        branch_pool = AveragePooling2D(
            (3, 3), strides=(1, 1), padding='same')(x)
        branch_pool = conv2d_bn(branch_pool, 192, 1, 1)
        x = layers.concatenate(
            [branch1x1, branch3x3, branch3x3dbl, branch_pool],
            axis=channel_axis,
            name='mixed' + str(9 + i))
    xt = []
    if include_top:
        # Classification block
        x = GlobalAveragePooling2D(name='avg_pool')(x)
        xt = Dense(classes, activation='softmax',
name='predictions')(x)
    else:
        if pooling == 'avg':
            x = GlobalAveragePooling2D()(x)
        elif pooling == 'max':
            x = GlobalMaxPooling2D()(x)

    # Ensure that the model takes into account
    # any potential predecessors of `input_tensor`.
    if input_tensor is not None:
        inputs = get_source_inputs(input_tensor)
    else:
        inputs = img_input
    # Create model.
    model = Model(inputs, [x, xt], name='inception_v3')
```

```
    # load weights
    if weights == 'imagenet':
        if K.image_data_format() == 'channels_first':
            if K.backend() == 'tensorflow':
                warnings.warn('You are using the TensorFlow
backend, yet you '
                              'are using the Theano '
                              'image data format convention '

'(`image_data_format="channels_first"`). '
                              'For best performance, set '

'`image_data_format="channels_last"` in '
                              'your Keras config '
                              'at ~/.keras/keras.json.')
        if include_top:
            weights_path = get_file(
                'incep-
tion_v3_weights_tf_dim_ordering_tf_kernels.h5',
                WEIGHTS_PATH,
                cache_subdir='models',
                md5_hash='9a0d58056eeedaa3f26cb7ebd46da564')
        else:
            weights_path = get_file(
                'incep-
tion_v3_weights_tf_dim_ordering_tf_kernels_notop.h5',
                WEIGHTS_PATH_NO_TOP,
                cache_subdir='models',
                md5_hash='bcbd6486424b2319ff4ef7d526e38f63')
        model.load_weights(weights_path)
        if K.backend() == 'theano':
            convert_all_kernels_in_model(model)
    return model


def preprocess_input(x):
    x /= 255.
    x -= 0.5
    x *= 2.
    return x


def timeit(method):
    import time
    def timed(*args, **kw):
        ts = time.time()
        result = method(*args, **kw)
        te = time.time()

        print ('%r %2.2f sec' % \
```

```
                (method.__name__, te-ts))
        return result

    return timed


@timeit
def predict(model, np_x):
    preds = model.predict(np_x)
    print('Predicted:', decode_predictions(preds[1]))


def main():
    model = InceptionV3(include_top=True, weights='imagenet')

    img_path = 'test_cat.jpg'
    img = image.load_img(img_path, target_size=(299, 299))
    x = image.img_to_array(img)


    x = preprocess_input(x)

    n = 100
    np_x = np.array([x for _ in xrange(n)])
    # np_x = np.expand_dims(np_x, axis=0)
    predict(model, np_x)


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

Учебное электронное издание комбинированного распространения

Учебное издание

Курочка Константин Сергеевич
Панарин Константин Александрович

НЕЙРОСЕТЕВАЯ ОБРАБОТКА ДАННЫХ

Учебно-методическое пособие для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» дневной и заочной форм обучения

Электронный аналог печатного издания

Редакторы *Т. Н. Мисюрова, Н. В. Гладкова*
Компьютерная верстка *Н. Б. Козловская*

Подписано в печать 21.12.21.
Формат 60x84/16. Бумага офсетная. Гарнитура «Таймс».
Цифровая печать. Усл. печ. л. 15,34. Уч.-изд. л. 16,24.
Изд. № 22.
http://www.gstu.by

Издатель и полиграфическое исполнение
Гомельский государственный
технический университет имени П. О. Сухого.
Свидетельство о гос. регистрации в качестве издателя
печатных изданий за № 1/273 от 04.04.2014 г.
пр. Октября, 48, 246746, г. Гомель