Ф.М. ГАФАРОВ, А.Ф. ГАЛИМЯНОВ

ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ И ИХ ПРИЛОЖЕНИЯ

Учебное пособие

Казань – 2018

Издательство Казанского университета

УДК 004.032.26
ББК 32.973.2-018+32.813
Г12

Печатается по постановлению
Редакционно-издательского совета
Института вычислительной математики и информационных технологий
Казанского (Приволжского) федерального университета;
(протокол №1 от 18 октября 2018 г.)

Научный редактор
кандидат педагогических наук, доцент Ч.Б. Миннегалиева

Рецензенты:
кандидат физико-математических наук,
доцент кафедры теории функций и приближений КФУ **Ю.Р. Агачев;**
кандидат технических наук,
с.н.с. научно-исследовательского института АН РТ «Прикладная семиотика»
А.Р. Гатиятуллин

Гафаров Ф.М
Г12 Искусственные нейронные сети и приложения: учеб. пособие / Ф.М. Гафаров, А.Ф. Галимянов. – Казань: Изд-во Казан. ун-та, 2018. – 121 с.

Учебное пособие посвящено изложению основ теории нейронных сетей и работы с популярным фреймвоком KERAS и TENSORFLOW. Приводятся также все необходимые вводные материалы для дальнейшего понимания.

Адресовано, в первую очередь, студентам-бакалаврам, а также магистрам направления «Информационные системы и технологии», а также широкому кругу читателей, интересующихся нейронными сетями и приложениями.

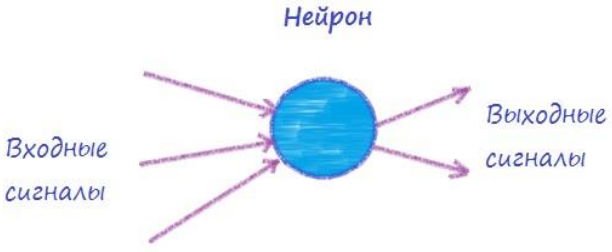
УДК 004.032.26
ББК 32.973.2-018+32.813

ОГЛАВЛЕНИЕ

1. НЕЙРОНЫ И ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ

Изучение и использование искусственных нейронных сетей, в принципе, началось уже достаточно давно – в начале 20 века, но по настоящему широкую известность они получили несколько позже. Связано это, в первую очередь, с тем, что стали появляться продвинутые (для того времени) вычислительные устройства, мощности которых были достаточно велики для работы с искусственными нейронными сетями. По сути, на данный момент можно легко смоделировать нейронную сеть средней сложности на любом персональном компьютере.

Нейронная сеть представляет из себя совокупность нейронов, соединенных друг с другом определенным образом. Рассмотрим один нейрон:



Нейрон представляет из себя элемент, который вычисляет выходной сигнал (по определенному правилу) из совокупности входных сигналов. То есть основная последовательность действий одного нейрона такая:

- Прием сигналов от предыдущих элементов сети
- Комбинирование входных сигналов
- Вычисление выходного сигнала
- Передача выходного сигнала следующим элементам нейронной сети

Между собой нейроны могут быть соединены абсолютно по-разному, это определяется структурой конкретной сети. Но суть работы нейронной сети остается всегда одной и той же. По совокупности поступающих на вход сети сигналов на выходе формируется выходной сигнал (или несколько выходных сигналов). То есть нейронную сеть упрощенно можно представить в виде черного ящика, у которого есть входы и выходы. А внутри этого ящика сидит огромное количество нейронов

Мы перечислили основные этапы работы сети, теперь давайте остановимся на каждом из них в отдельности.

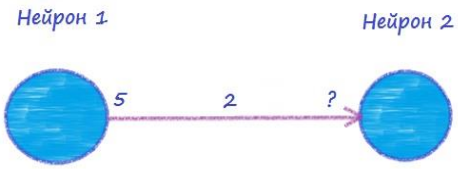
1.1. Комбинирование входных сигналов

Поскольку к каждому нейрону могут приходить несколько входных сигналов, то при моделировании нейронной сети необходимо задать определенное правило комбинирования всех этих сигналов. И довольно-таки часто используется правило суммирования взвешенных значений связей. Что значит взвешенных? Сейчас разберемся…

Каждую связь в сети нейронов можно полностью охарактеризовать при помощи трех факторов:

- первый – элемент, от которого исходит связь
- второй – элемент, к которую связь направлена
- третий – вес связи.

Сейчас нас в большей степени интересует именно третий фактор. Вес связи определяет, будет ли усилен или ослаблен сигнал, передаваемый по данной связи. Если объяснять просто, “на пальцах”, то давайте рассмотрим такой пример:



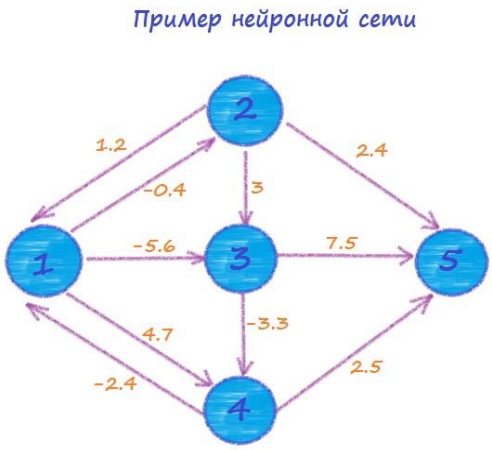
Выходной сигнал нейрона 1 равен 5. Вес связи между нейронами равен 2. Таким образом, чтобы определить входной сигнал нейрона 2, приходящий от нейрона 1, необходимо умножить значение этого сигнала на вес связи (5*2).

А если сигналов много, то они все суммируются. В итоге на входе нейрона мы получаем следующее:

$$net_j = \sum_{i=1}^{N} x_i * w_{ij}$$

В этой формуле net_j – это результат комбинирования всех входных сигналов для нейрона j (комбинированный ввод нейрона). N – количество элементов, передающих свои выходные сигналы на вход сигнала j. А w_{ij}– вес связи, соединяющей нейрон i с нейроном j. Суммируя все взвешенные входные сигналы, мы получаем комбинированный ввод элемента сети.

Чаще всего структура связей между нейронами представляется в виде матрицы W, которую называют весовой матрицей. Элемент матрицы w_{ij}, как и в формуле, определяет вес связи, идущей от элемента i к элементу j. Для того, чтобы понять, как составляются весовые матрицы, давайте рассмотрим простую нейронную сеть:



Весовая матрица такой нейронной сети будет иметь следующий вид:

$$W = \begin{bmatrix} 0 & -0.4 & -5.6 & 4.7 & 0 \\ 1.2 & 0 & 3 & 0 & 2.4 \\ 0 & 0 & 0 & -3.3 & 7.5 \\ -2.4 & 0 & 0 & 0 & 2.5 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Например, от второго элемента к третьему идет связь, вес которой равен 3. Смотрим на матрицу, вторая строка, третий столбец – число 3, все верно.

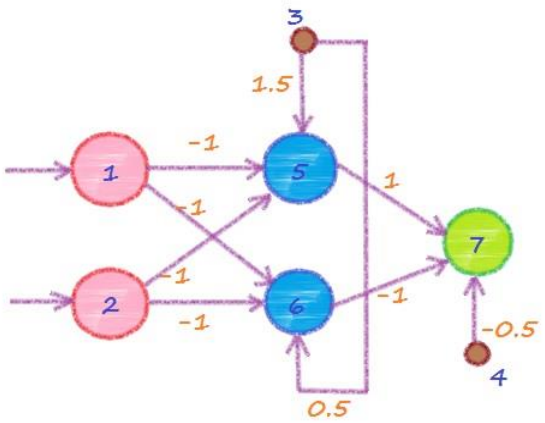
1.2. Функция активации элемента.

Рассмотрим выходные сигналы. Для каждого элемента сети имеется определенное правило, в соответствии с которым из значения комбинированного ввода элемента вычисляется его выходное значение. Это правило называется функцией активации. А само выходное значение называется активностью нейрона. В роли функций активации могут выступать абсолютно любые математические функции, приведем в качестве примера несколько из наиболее часто использующихся:

- пороговая функция – если значение комбинированного ввода ниже определенного значения (порога), то активность равна нулю, если выше – единице.
- логистическая функция.

Рассмотрим еще один небольшой пример, который очень часто используется в литературе для объяснения сути работы нейронных сетей.

Задача примера заключается в том, чтобы при помощи нейронной сети вычислить отношение XOR. То есть на вход мы будем подавать разные варианты сигналов, а на выходе должны получить результат операции XOR для поданных на вход значений:



Элементы 1 и 2 являются входными, а элемент 7 – выходным. Нейроны 5 и 6 называются скрытыми, поскольку они не связаны с внешней средой. Таким образом, мы получили три слоя – входной, скрытый и выходной. Элементы 3 и 4 называют элементами смещения. Их выходной сигнал (активность) всегда равен 1. Для вычисления комбинированного ввода в этой сети мы будем использовать правило суммирования взвешенных связей, а в качестве функции активности будет выступать пороговая функция. Если комбинированный ввод элемента меньше 0, то активность равна 0, если ввод больше 0, то активность – 1.

Подадим на вход нейрона 1 – единицу, а на вход нейрона 2 – ноль. В этом случае на выходе мы должны получить 1 (0 XOR 1 = 1). Рассчитаем выходное значение вручную для демонстрации работы сети.

Комбинированный ввод элемента 5:$net_5 = 1 * (-1) + 0 * (-1) + 1 * 1.5 = 0.5$.

Активность элемента 5: 1 (0.5 > 0).

Комбинированный ввод элемента 6: $net_6 = 1 * (-1) + 0 * (-1) + 1 * 0.5 = -0.5$.

Активность элемента 6: 0.

Комбинированный ввод элемента 7: $net_7 = 1 * (1) + 0 * (-1) + 1 * (-0.5) = 0.5$.

Активность элемента 7, а в то же время и выходное значение сети равно 1. Что и требовалось доказать.

Можно попробовать использовать в качестве входных сигналов все возможные значения (0 и 0, 1 и 0, 0 и 1, 1 и 1), на выходе мы всегда будем видеть значение, соответствующее таблице истинности операции XOR.

В данном случае все значения весовых коэффициентов нам были известны заранее, но главной особенностью нейронных сетей является то, что они могут сами корректировать значения веса всех связей в процессе обучения сети.

2. ИСТОРИЯ НЕЙРОННЫХ СЕТЕЙ

Основные этапы в истории исследования и применения искусственных нейронных сетей:

- 1943 — У. Маккалок и У. Питтс формализуют понятие нейронной сети в фундаментальной статье о логическом исчислении идей и нервной активности.
- 1948 — Н. Винер вместе с соратниками публикует работу о кибернетике. Основной идеей является представление сложных биологических процессов математическими моделями.
- 1949 — Д. Хебб предлагает первый алгоритм обучения.
- В 1958 Ф. Розенблатт изобретает однослойный перцептрон и демонстрирует его способность решать задачи классификации. Перцептрон обрёл популярность — его используют для распознавания образов, прогнозирования погоды и т. д.
- В 1960 году Уидроу совместно со своим студентом Хоффом на основе дельта-правила (формулы Уидроу) разработали Адалин, который сразу начал использоваться для задач предсказания и адаптивного управления.

Сейчас Адалин (адаптивный сумматор) является стандартным элементом многих систем обработки сигналов.

- В 1963 году в Институте проблем передачи информации АН СССР. А. П. Петровым проводится подробное исследование задач «трудных» для перцептрона.
- В 1969 году М. Минский публикует формальное доказательство ограниченности перцептрона и показывает, что он неспособен решать некоторые задачи (проблема «чётности» и «один в блоке»), связанные с инвариантностью представлений. Интерес к нейронным сетям резко спадает.
- В 1972 году Т. Кохонен и Дж. Андерсон независимо предлагают новый тип нейронных сетей, способных функционировать в качестве памяти.
- В 1973 году Б. В. Хакимов предлагает нелинейную модель с синапсами на основе сплайнов и внедряет её для решения задач в медицине, геологии, экологии.
- 1974 — Пол Дж. Вербос и А. И. Галушкин одновременно изобретают алгоритм обратного распространения ошибки для обучения многослойных перцептронов
- 1975 — Фукусима представляет когнитрон — самоорганизующуюся сеть, предназначенную для инвариантного распознавания образов, но это достигается только при помощи запоминания практически всех состояний образа.
- 1982 — 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 — Дэвидом И. Румельхартом, Дж. Е. Хинтоном и Рональдом Дж. Вильямсом и одновременно с С. И. Барцевым и В. А. Охониным (Красноярская группа) переоткрыт и существенно развит метод обратного распространения ошибки. Начался взрыв интереса к обучаемым нейронным сетям.
- 2007 Джеффри Хинтоном в университете Торонто созданы алгоритмы глубокого обучения многослойных нейронных сетей. Успех обусловлен тем, что Хинтон при обучении нижних слоев сети использовал ограниченную машину Больцмана (RBM — Restricted Boltzmann Machine).

3. КЛАССИФИКАЦИЯ НЕЙРОННЫХ СЕТЕЙ

Классификация нейронных сетей по характеру обучения делит их на:

- нейронные сети, использующие обучение с учителем;
- нейронные сети, использующие обучение без учителя.

Нейронные сети, использующие обучение с учителем. Обучение с учителем предполагает, что для каждого входного вектора существует целевой вектор, представляющий собой требуемый выход. Вместе они называются обучающей парой. 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.

Нейронные сети, использующие обучение без учителя. 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.

Классификация нейронных сетей по типу настройки весов делит их на:

- сети с фиксированными связями – весовые коэффициенты нейронной сети выбираются сразу, исходя из условий задачи;
- сети с динамическими связями – для них в процессе обучения происходит настройка синаптических весов.

Классификация нейронных сетей по типу входной информации делит их на:

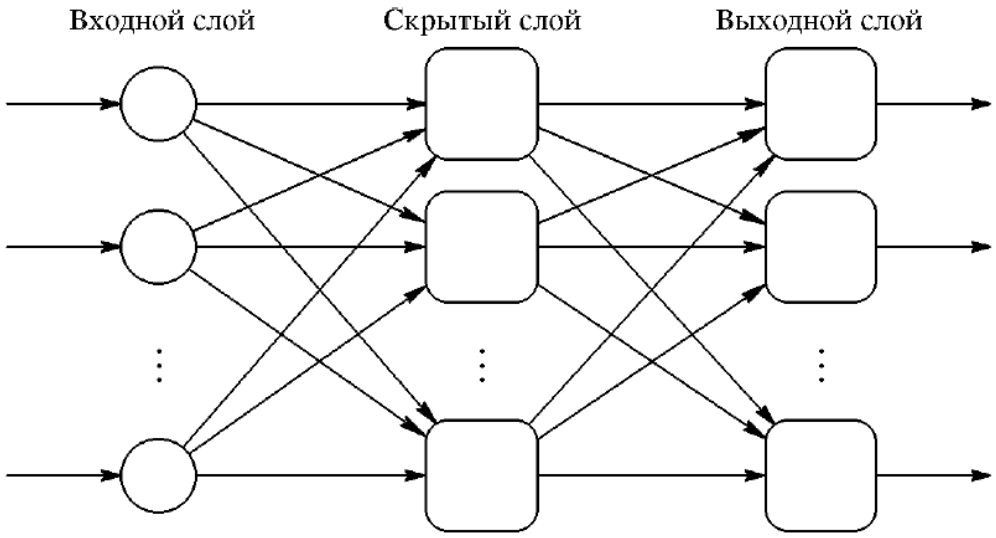
- аналоговые – входная информация представлена в форме действительных чисел;
- двоичные – вся входная информация в таких сетях представляется в виде нулей и единиц.

4. АРХИТЕКТУРЫ НЕЙРОННЫХ СЕТЕЙ

В **полносвязных нейронных сетях** каждый нейрон передает свой выходной сигнал остальным нейронам, в том числе и самому себе. Все входные сигналы подаются всем нейронам. Выходными сигналами сети могут быть все или некоторые выходные сигналы нейронов после нескольких тактов функционирования сети.

В **многослойных (слоистых) нейронных сетях** 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 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.



4.1 Типы многослойных нейронных сетей

Монотонные. Это частный случай слоистых сетей с дополнительными условиями на связи и нейроны. Каждый слой, кроме последнего (выходного), разбит на два блока: возбуждающий и тормозящий. Связи между блоками тоже разделяются на тормозящие и возбуждающие. Если от нейронов блока к нейронам блока ведут только возбуждающие связи, то это означает, что любой выходной сигнал блока является монотонной неубывающей функцией любого выходного сигнала блока. Если же эти связи только тормозящие, то любой выходной сигнал блока является невозрастающей функцией любого выходного сигнала блока. Для нейронов монотонных сетей необходима монотонная зависимость выходного сигнала нейрона от параметров входных сигналов.

Сети без обратных связей. В таких сетях нейроны входного слоя получают входные сигналы, преобразуют их и передают нейронам первого скрытого слоя, и так далее вплоть до выходного, который выдает сигналы для интерпретатора и пользователя. Если не оговорено противное, то каждый

выходной сигнал q-го слоя подается на вход всех нейронов (q+1)-го слоя; однако возможен вариант соединения q-го слоя с произвольным -м слоем.

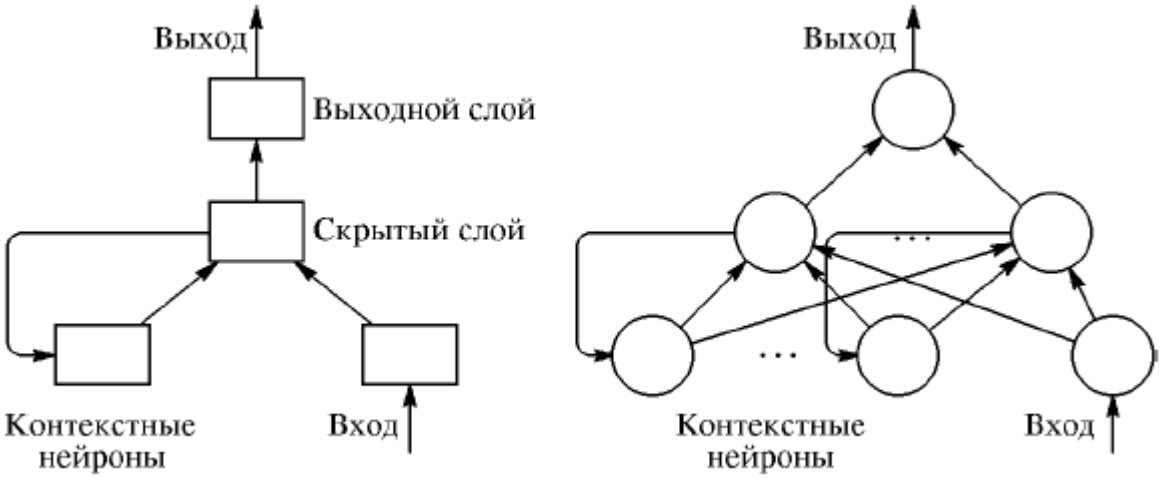
Среди многослойных сетей без обратных связей различают **полносвязные** (выход каждого нейрона q–го слоя связан с входом каждого нейрона (q+1)–го слоя) и частично **полносвязные.**

4.2. Сети с обратными связями

В **сетях с обратными связями** информация с последующих слоев передается на предыдущие. Различают следующие типы нейронных сетей с обратными связями:

- **слоисто-циклические**, отличающиеся тем, что слои замкнуты в кольцо: последний слой передает свои выходные сигналы первому; все слои равноправны и могут как получать входные сигналы, так и выдавать выходные;
- **слоисто-полносвязные** состоят из слоев, каждый из которых представляет собой полносвязную сеть, а сигналы передаются как от слоя к слою, так и внутри слоя; в каждом слое цикл работы распадается на три части: прием сигналов с предыдущего слоя, обмен сигналами внутри слоя, выработка выходного сигнала и передача к следующему слою;
- **полносвязно-слоистые**, по своей структуре аналогичные слоисто-полно-связным, но функционирующим по-другому: в них не разделяются фазы обмена внутри слоя и передачи следующему, на каждом такте нейроны всех слоев принимают сигналы от нейронов как своего слоя, так и последующих.

На картике изображены сеть Элмана и сеть Жордана

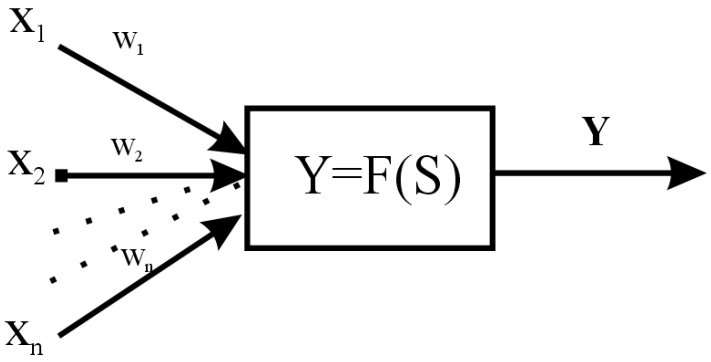


Нейронные сети можно разделить по типам структур нейронов на гомогенные (однородные) и гетерогенные. **Гомогенные** сети состоят из нейронов одного типа с единой функцией активации, а в **гетерогенную** сеть входят нейроны с различными функциями активации.

Еще одна классификация делит нейронные сети на **синхронные** и **асинхронные**. В первом случае в каждый момент времени лишь один нейрон меняет свое состояние, во втором – состояние меняется сразу у целой группы нейронов, как правило, у всего слоя. Алгоритмически ход времени в нейронных сетях задается итерационным выполнением однотипных действий над нейронами.

5. ФОРМАЛЬНЫЙ НЕЙРОН

У нейрона есть несколько входных каналов и только один выходной канал. По входным каналам на нейрон поступают данные задачи, а на выходе формируется результат работы. Нейрон вычисляет взвешенную сумму входных сигналов, а затем преобразует полученную сумму с помощью заданной нелинейной функции. Множество, состоящее из порогового уровня и всех весов, называют параметрами нейрона.



Здесь введены следующие обозначения: $X_1, X_2, \ldots, X_n$ - входной сигнал (паттерн), $w_1, w_2, \ldots, w_n$ – весовые коэффициенты, b- порог нейрона

Сначала нейрон вычисляет взвешенную сумму $S = \sum_i w_i X_i - b$, далее применяя **функцию активации** F(S) вычисляет выходной сигнал Y.

Функция активации нейрона - это функция, которая вычисляет выходной сигнал нейрона. На вход этой функции подается сумма всех произведений сигналов и весов этих сигналов.

Рассмотрим наиболее часто используемые функции активации.

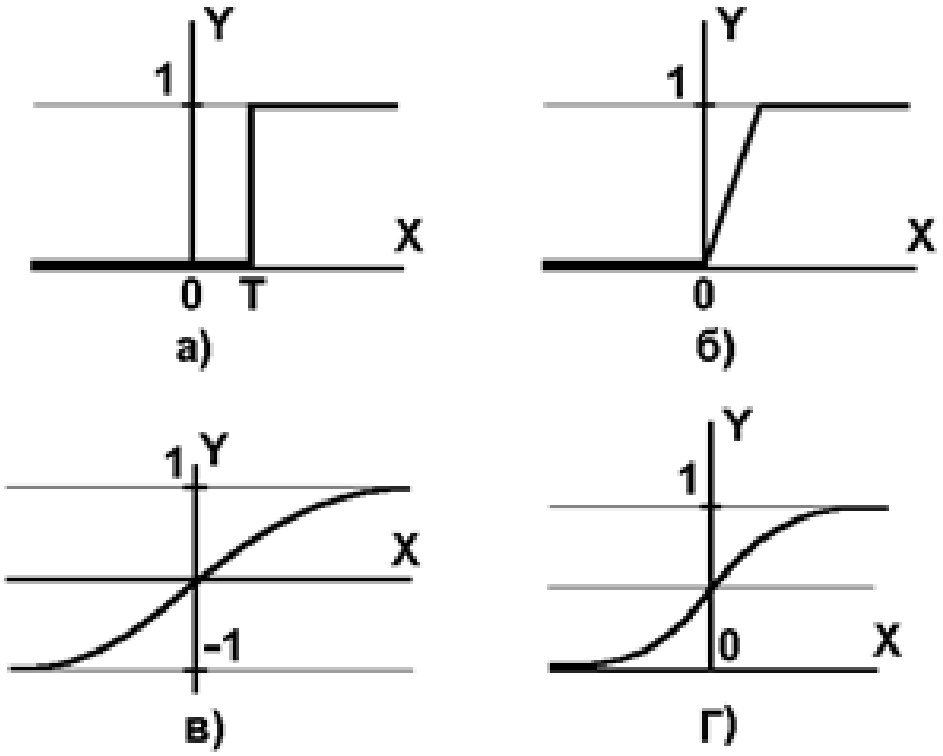
а) Пороговая функция. Это простая кусочно-линейная функция. Если входное значение меньше порогового, то значение функции активации равно минимальному допустимому, иначе – максимально допустимому.

б) Линейный порог. Это несложная кусочно-линейная функция. Имеет два линейных участка, где функция активации тождественно равна минимально допустимому и максимально допустимому значению и есть участок, на котором функция строго монотонно возрастает.

в) Сигмоидальная функция или **сигмоида (sigmoid)**. Это монотонно возрастающая дифференцируемая S-образная нелинейная функция. Сигмоида позволяет усиливать слабые сигналы и не насыщаться от сильных сигналов.

г) Гиперболический тангенс (hyperbolic tangent, tanh). Эта функция принимает на входе произвольное вещественное число, а на выходе дает вещественное число в интервале от –1 до 1. Подобно сигмоиде,

гиперболический тангенс может насыщаться. Однако, в отличие от сигмоиды, выход данной функции центрирован относительно нуля.



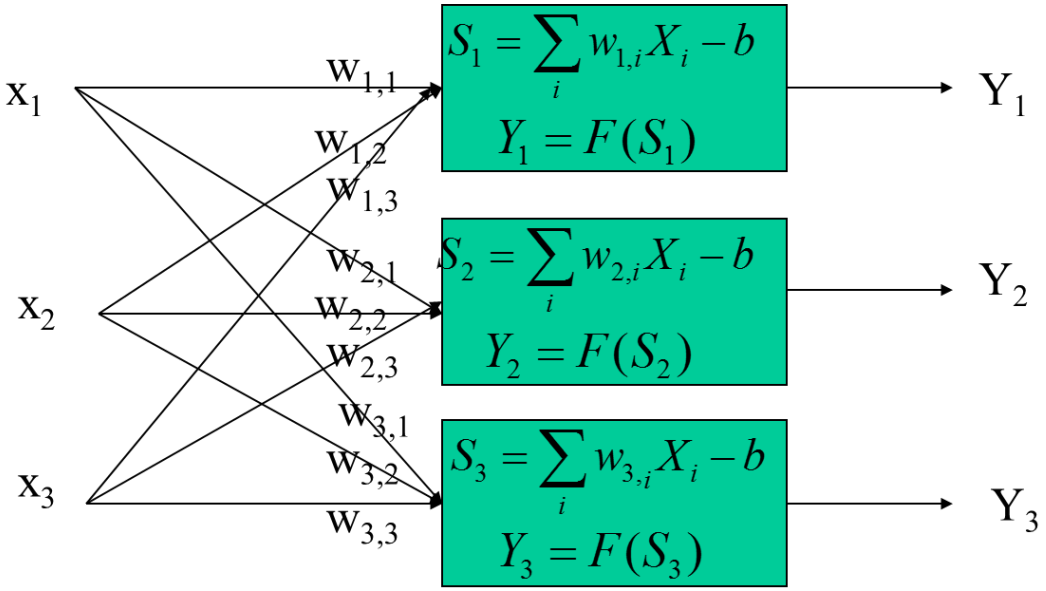
Недостатки формального нейрона:

- Предполагается, что нейрон мгновенно вычисляет свой выход, поэтому с помощью таких нейронов нельзя моделировать непосредственно системы с внутренним состоянием.
- Формальные нейроны, в отличие от биологических, не могут обрабатывать информацию синхронно.
- Нет четких алгоритмов выбора функции активации.
- Невозможно регулировать работу всей сети.
- Излишняя формализация понятий «порог» и «весовые коэффициенты». У реальных нейронов порог меняется динамически, в зависимости от активности нейрона и общего состояния сети, а весовые коэффициенты изменяются в зависимости от проходящих сигналов.

6. ОДНОСЛОЙНАЯ НЕЙРОННАЯ СЕТЬ

Один нейрон может выполнять простейшие вычисления, но основные функции нейросети обеспечиваются не отдельными нейронами, а

соединениями между ними. Однослойный перцептрон представляет собой простейшую сеть, которая состоит из группы нейронов, образующих слой. Входные данные кодируются вектором значений, каждый элемент подается на соответствующий вход каждого нейрона в слое. В свою очередь, нейроны вычисляют выход независимо друг от друга. Размерность выхода (то есть количество элементов) равна количеству нейронов, а количество синапсов у всех нейронов должно быть одинаково и совпадать с размерностью входного сигнала.

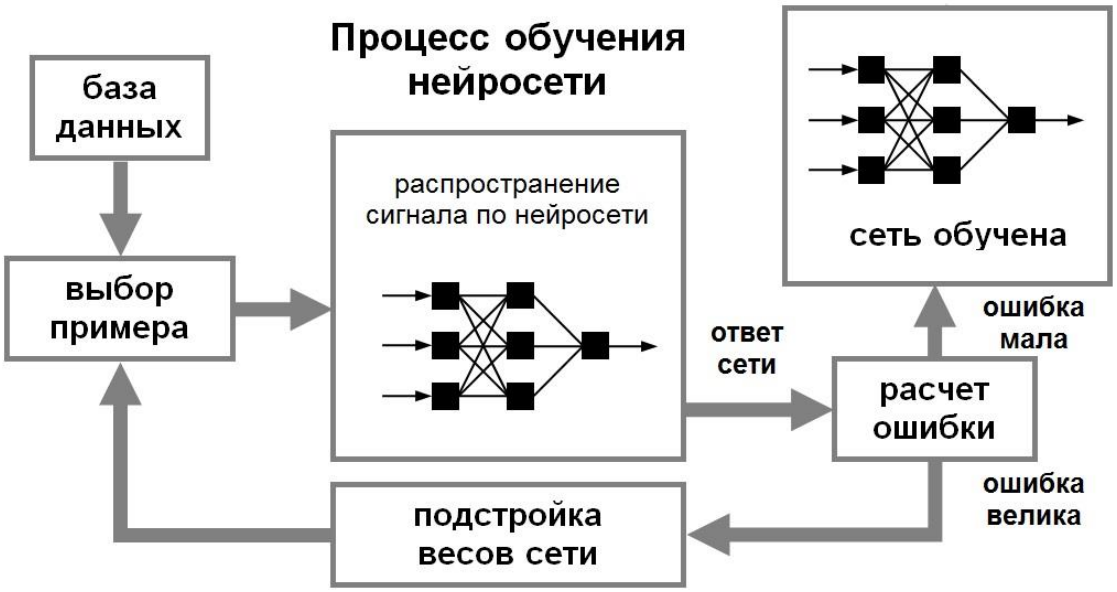


Здесь X_1, X_2, X_3- называется входной паттерн, Y_1, Y_2, Y_3- выходной паттерн, а $w_{i,j}$- это j-ый весовой коэффициент i-го нейрона

7. ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ

Обучение нейронной сети- это процесс, в котором параметры нейронной сети настраиваются посредством моделирования среды, в которую эта сеть встроена. Тип обучения определяется способом подстройки параметров. Различают алгоритмы обучения с учителем и без учителя. Процесс обучения с учителем представляет собой предъявление сети выборки обучающих примеров. Каждый образец подается на входы сети, затем проходит обработку внутри структуры НС, вычисляется выходной сигнал сети, который сравнивается с соответствующим значением целевого вектора, представляющего собой требуемый выход сети.

Для того, чтобы нейронная сети была способна выполнить поставленную задачу, ее необходимо обучить. Различают алгоритмы обучения с учителем и без учителя. Процесс обучения с учителем представляет собой предъявление сети выборки обучающих примеров. Каждый образец подается на входы сети, затем проходит обработку внутри структуры НС, вычисляется выходной сигнал сети, который сравнивается с соответствующим значением целевого вектора, представляющего собой требуемый выход сети. Затем по определенному правилу вычисляется ошибка, и происходит изменение весовых коэффициентов связей внутри сети в зависимости от выбранного алгоритма. Векторы обучающего множества предъявляются последовательно, вычисляются ошибки и веса подстраиваются для каждого вектора до тех пор, пока ошибка по всему обучающему массиву не достигнет приемлемо низкого уровня.



При обучении без учителя 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. Предъявление на вход вектора из данного класса даст определенный выходной вектор, но до обучения

невозможно предсказать, какой выход будет производиться данным классом входных векторов. Следовательно, выходы подобной сети должны трансформироваться в некоторую понятную форму, обусловленную процессом обучения. Это не является серьезной проблемой. Обычно не сложно идентифицировать связь между входом и выходом, установленную сетью. Для обучения нейронных сетей без учителя применяются сигнальные метод обучения Хебба и Ойа.

Математически процесс обучения можно описать следующим образом. В процессе функционирования нейронная сеть формирует выходной сигнал Y, реализуя некоторую функцию Y = G(X). Если архитектура сети задана, то вид функции G определяется значениями синаптических весов и смещенной сети.

Пусть решением некоторой задачи является функция Y = F(X), заданная параметрами входных-выходных данных (X^1, Y^1), (X^2, Y^2), …, (X^N, Y^N), для которых $Y^k = F(X^k)$ (k = 1, 2, …, N).

Обучение состоит в поиске (синтезе) функции G, близкой к F в смысле некоторой функции ошибки E.

Если выбрано множество обучающих примеров – пар (X^N, Y^N) (где k = 1, 2, …, N) и способ вычисления функции ошибки E, то обучение нейронной сети превращается в задачу многомерной оптимизации, имеющую очень большую размерность, при этом, поскольку функция E может иметь произвольный вид обучение в общем случае – многоэкстремальная невыпуклая задача оптимизации.

Для решения этой задачи могут использоваться следующие (итерационные) алгоритмы:

1. алгоритмы локальной оптимизации с вычислением частных производных первого порядка:
   - градиентный алгоритм (метод наискорейшего спуска),
   - методы с одномерной и двумерной оптимизацией целевой функции в направлении антиградиента,
   - метод сопряженных градиентов,

- методы, учитывающие направление антиградиента на нескольких шагах алгоритма;

2. алгоритмы локальной оптимизации с вычислением частных производных первого и второго порядка:

- метод Ньютона,
- методы оптимизации с разреженными матрицами Гессе,
- квазиньютоновские методы,
- метод Гаусса-Ньютона,
- метод Левенберга-Марквардта и др.;

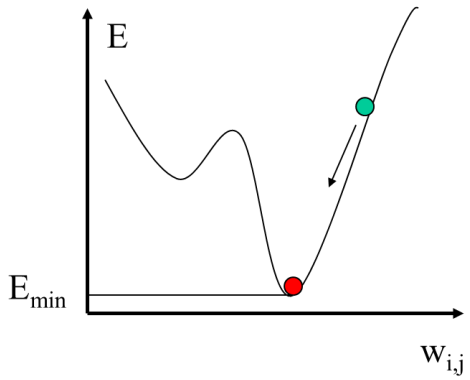
3. стохастические алгоритмы оптимизации:

- поиск в случайном направлении,
- имитация отжига,
- метод Монте-Карло (численный метод статистических испытаний);

4. алгоритмы глобальной оптимизации (задачи глобальной оптимизации решаются с помощью перебора значений переменных, от которых зависит целевая функция).

7.1. Метод градиентного спуска в пространстве весовых коэффициентов

Градиентный спуск — метод нахождения локального экстремума (минимума или максимума) функции с помощью движения вдоль градиента.



Весовые коэффициенты и смещения вычисляются по формулам:

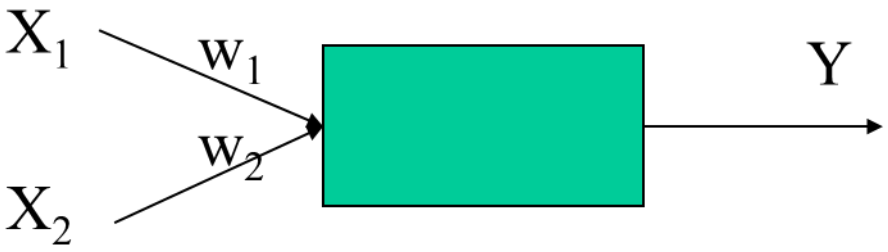
$$w_{i,j}(t+1) = w_{i,j}(t) - \alpha \frac{\partial E}{\partial w_{i,j}}$$

$$b_i(t+1) = b_i(t) - \alpha \frac{\partial E}{\partial b_i}$$

Здесь E- функционал ошибки, α- скорость обучения.

7.2. Правило обучения Уидроу-Хоффа

Правило обучения Видроу-Хоффа известно под названием дельта-правило. Оно предполагает минимизацию среднеквадратичной ошибки нейронной сети, которая для входных образов определяется по формуле: $E = \frac{1}{2}(Y-d)^2$, где d- целевое выходное значение



Каждый нейрон вычисляет взвешенную сумму по формуле: $S=w_1X_1+w_2X_2-b$. Если использовать линейную функцию активацииY=x, то функционал ошибок будет равен:

$$E = \frac{1}{2}(w_1X_1 + w_2X_2 - b - d)^2$$

А производные от функционала ошибок будут выражаться следующим образом

$$\frac{\partial E}{\partial w_1} = (Y-d)X_1$$

$$\frac{\partial E}{\partial w_2} = (Y - d)X_2$$

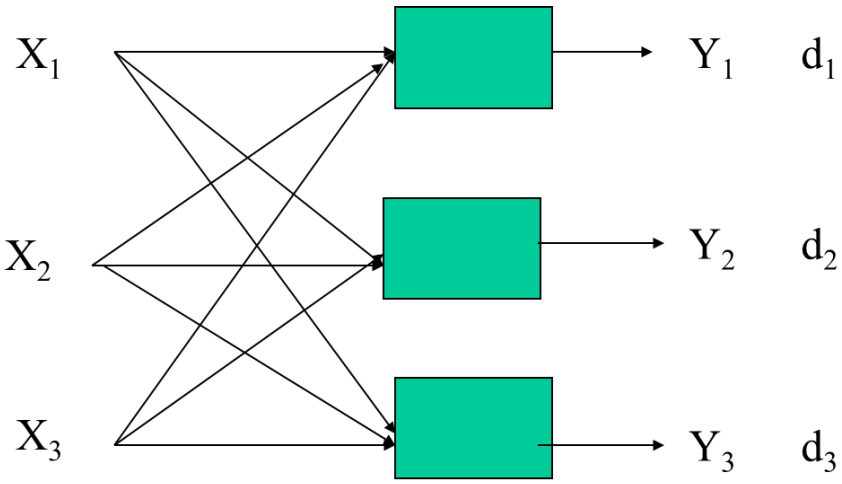
$$\frac{\partial E}{\partial b} = -(Y - d)$$

Весовые коэффициенты и смещение нейрона вычисляются по формулам:

$$w_i(t+1) = w_i(t) - \alpha(Y - d)X_i$$

$$b(t+1) = b(t) + \alpha(Y - d)$$

Рассмотрим нейронную сеть, состоящую из одного слоя с тремя нейронами.



Здесь вектора$\{X_1,X_2,X_3\}$ и $\{d_1, d_2, d_3\}$ представляют собой обучающую пару.

В этом случае функционал ошибки будет равен $E = \frac{1}{2}\sum_j (Y_j - d_j)^2$,

а весовые коэффициенты и смещения нейронов вычисляются по формулам:

$$w_{ij}(t+1) = w_{ij}(t) - \alpha(Y_j - d_j)X_i$$

$$b_j(t+1) = b_j(t) + \alpha(Y_j - d_j)$$

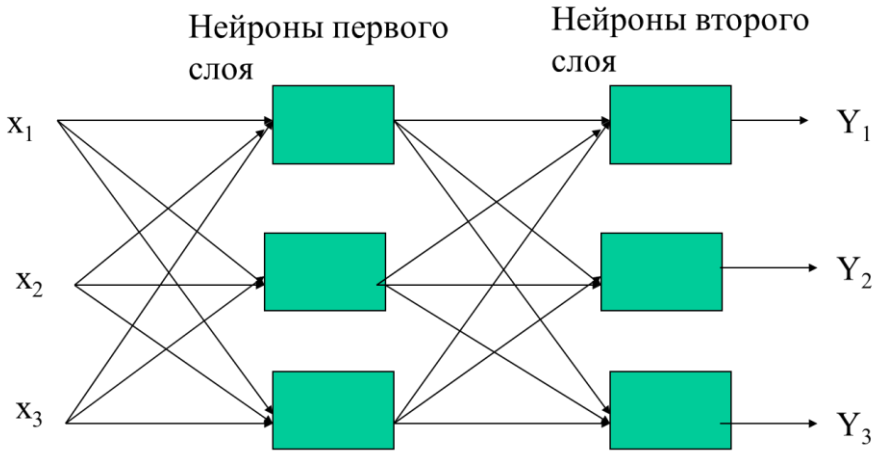
7.3. Алгоритм обучения однослойной НС

Рассмотрим последовательность шагов при обучении однослойной нейронной сети по правилу обучения Уидроу-Хоффа:

1. Задаются шаг обучения α(0< α <1) и желаемая среднеквадратичная ошибка сети E_m.
2. Инициализируются случайным образом весовые коэффициенты $w_{i,j}$ и пороговые b_j значения нейронов.
3. Подаются последовательно образы из обучающей выборки на вход нейронной сети. Вычисляется выходные значения нейронов.
4. Производится изменение весовых коэффициентов и порогов нейронных элементов.
5. Вычисляется суммарная ошибка нейронной сети E
6. Если $E>E_m$, то происходит переход к шагу 3, иначе выполнение алгоритма завершается

8. МНОГОСЛОЙНАЯ НЕЙРОННОЙ СЕТЬ

Многослойная нейронная сеть (персептрон) — это нейронная сеть, состоящая из входного, выходного и расположенных между ними одного (или нескольких) скрытых слоев нейронов.



Чтобы построить многослойный персептрон, необходимо выбрать его параметры по следующему алгоритму:

- Определить, какой смысл вкладывается в компоненты входного вектора X. Входной вектор должен содержать формализованное условие задачи, то есть всю информацию, необходимую для того, чтобы получить ответ.

- Выбрать выходной вектор Y таким образом, чтобы его компоненты содержали полный ответ для поставленной задачи.
- Выбрать вид функции активации нейронов. При этом желательно учесть специфику задачи, так как удачный выбор увеличит скорость обучения.
- Выбрать количество слоев и нейронов в слое.
- Задать диапазон изменения входов, выходов, весов и пороговых уровней на основе выбранной функции активации.
- Присвоить начальные значения весам и пороговым уровням. Начальные значения не должны быть большими, чтобы нейроны не оказались в насыщении (на горизонтальном участке функции активации), иначе обучение будет очень медленным. Начальные значения не должны быть и слишком малыми, чтобы выходы большей части нейронов не были равны нулю, иначе обучение тоже замедлится.
- Провести обучение, то есть подобрать параметры сети так, чтобы задача решалась наилучшим образом. По окончании обучения сеть сможет решать задачи того типа, которым она обучена.
- Подать на вход сети условия задачи в виде вектора X. Рассчитать выходной вектор Y, который и даст формализованное решение задачи.

8.1. Алгоритм обратного распространения ошибки

Алгоритм обратного распространения ошибки является одним из методов обучения многослойных нейронных сетей прямого распространения. Обучение алгоритмом обратного распространения ошибки предполагает два прохода по всем слоям сети: прямого и обратного. При прямом проходе входной вектор подается на входной слой нейронной сети, после чего распространяется по сети от слоя к слою. В результате генерируется набор выходных сигналов, который и является фактической реакцией сети на данный входной образ. Во время прямого прохода все синаптические веса сети фиксированы. Во время обратного прохода все синаптические веса настраиваются в соответствии с правилом коррекции ошибок, а именно: фактический выход сети вычитается из желаемого, в результате чего формируется сигнал ошибки. Этот сигнал

впоследствии распространяется по сети в направлении, обратном направлению синаптических связей. Отсюда и название – алгоритм обратного распространения ошибки. Синаптические веса настраиваются с целью максимального приближения выходного сигнала сети к желаемому.

Введем следующие обозначения: X_i- входной вектор, Y_i- выходной вектор, $w^k_{i,j}$- i-ый весовой коэффициент j-го нейрона k-го слоя, b^k_i- порог i-го нейрона k-го слоя, d_i- эталонное выходное значение i-го нейрона.

Выходное значение j-го нейрона k-го слоя вычисляется по формуле:

$$Y_j^k = F(\sum w_{i,j}^k Y_i^{k-1} - b_j^k)$$

Выходное значение j-го нейрона выходного слоя вычисляется по формуле:

$$Y_j = F(\sum w_{i,j} Y_i^{n-1} - b_j)$$

Функционал ошибки сети равен $E = \frac{1}{2}\sum_j (Y_j - d_j)^2$, где $\gamma_j = Y_j - d_j$ ошибка j-го нейрона выходного слоя. Ошибка j-го элемента k- го скрытого слоя

$$\gamma_j^k = \frac{\partial E}{\partial Y_j^k} = \sum_j \frac{\partial E}{\partial Y_j}\frac{\partial Y_j}{\partial S_j}\frac{\partial S_j}{\partial Y_j^k} = \sum_j \frac{\partial E}{\partial Y_j}\frac{\partial Y_j}{\partial S_j} w_{i,j} =$$

$$= \sum_j (Y_j - d_j) F'(S_j) w_{i,j} = \sum_j \gamma_j F'(S_j) w_{i,j}$$

Градиенты ошибок равны:

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial Y_j}\frac{\partial Y_j}{\partial S_j}\frac{\partial S_j}{\partial w_{i,j}} = \gamma_j F'(S_j) Y_j^k$$

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial Y_j}\frac{\partial Y_j}{\partial S_j}\frac{\partial S_j}{\partial b_j} = -\gamma_j F'(S_j)$$

$$\frac{\partial E}{\partial w_{i,j}^k} = \sum_j \frac{\partial E}{\partial Y_j}\frac{\partial Y_j}{\partial S_j}\frac{\partial S_j}{\partial Y_j^{k-1}}\frac{\partial Y_j^{k-1}}{\partial S_j^{k-1}}\frac{\partial S_j^{k-1}}{\partial w_{ij}^k} = \gamma_j F'(S_j^k) Y_j^k$$

Весовые коэффициенты и смещения нейронов вычисляются по формулам:

$$w_{i,j}^{k}(t+1) = w_{i,j}^{k} - \alpha\gamma_{j}^{k}F'(S_{j}^{k})Y_{j}^{k}$$

$$b_{j}^{k}(t+1) = b_{j}^{k} + \alpha\gamma_{j}^{k}F'(S_{j}^{k})$$

8.2. Алгоритм обучения многослойной НС

1. Задаются шаг обучения α (0< α <1) и желаемая среднеквадратичная ошибка сети E_m.
2. Инициализируются случайным образом весовые коэффициенты $w_{i,j}^{k}$ и пороговые b_j^k значения НС.
3. Подаются последовательно образы из обучающей выборки на вход нейронной сети. При этом для каждого образа выполняются следующие действия:
    a. Производится фаза прямого распространения входного образа по нейронной сети. Вычисляется выходное значение всех нейронов Y_j^k.
    b. Вычисляются ошибки γ_j нейронов выходного и скрытого слоев.
    c. Производится изменение весовых коэффициентов и порогов нейронных элементов для каждого слоя нейронной сети.
4. Вычисляется суммарная ошибка нейронной сети E
5. Если $E>E_m$, то происходит переход к шагу 3, иначе выполнение алгоритма завершается

9. ВВЕДЕНИЕ В KERAS И ЕГО ОСНОВНЫЕ ПРИНЦИПЫ

В современном мире, начиная со здравоохранения и заканчивая мануфактурным производством, повсеместно используется глубинное обучение. Компании обращаются к этой технологии для решения сложных проблем, таких как распознавание речи и объектов, машинный перевод и так далее.

Одним из самых впечатляющих достижений этого года был AlphaGo, обыгравший лучшего в мире игрока в го. Кроме как в го, машины обошли людей и в других играх: шашки, шахматы, реверси, и джеопарди.

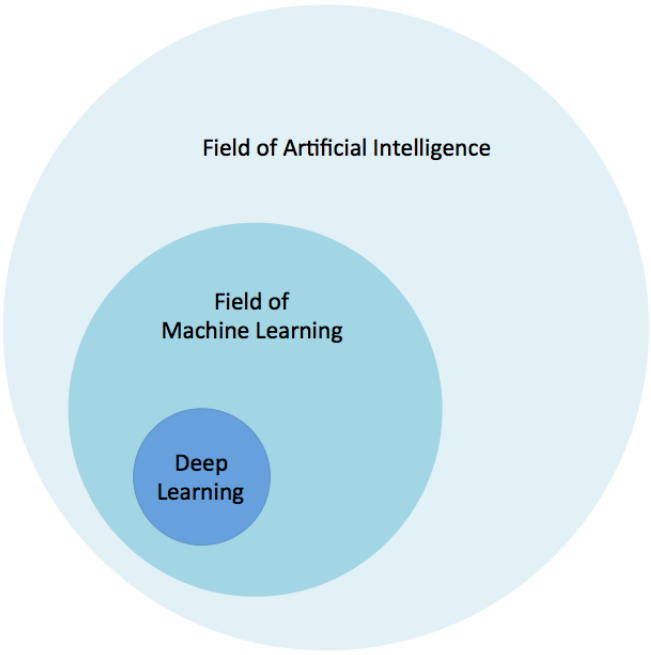
Возможно, победа в настольной игре кажется неприменимой в решении реальных проблем, однако это совсем не так. Го был создан так, чтобы в нем не мог победить искусственный интеллект. Для этого ему необходимо было бы научиться одной важной для этой игры вещи – человеческой интуиции. Теперь с помощью данной разработки возможно решить множество проблем, недоступных компьютеру раньше.

Очевидно, глубинное обучение еще далеко от совершенства, но оно уже близко к тому, чтобы приносить коммерческую пользу. Например, эти самоуправляемые машины. Известные компании вроде Google, Tesla и Uber уже пробуют внедрить автономные автомобили на улицы города. Ford предсказывает значительное увеличение доли беспилотных транспортных средств уже к 2021 году. Правительство США также успело разработать для них свод правил безопасности.

Keras является высокоуровневыми нейронными сетями API, написанный на Python и могут работать поверх TensorFlow, CNTK или Теано. Он был разработан с упором на возможность быстрого экспериментирования. Способность идти от идеи к результату с наименьшей возможной задержкой является ключом к проведению хороших исследований.

9.1 Что такое глубинное обучение?

Чтобы ответить на этот вопрос, нужно понять, как оно взаимодействует с машинным обучением, нейросетями и искусственным интеллектом. Для этого используем метод визуализации с помощью концентрических кругов:



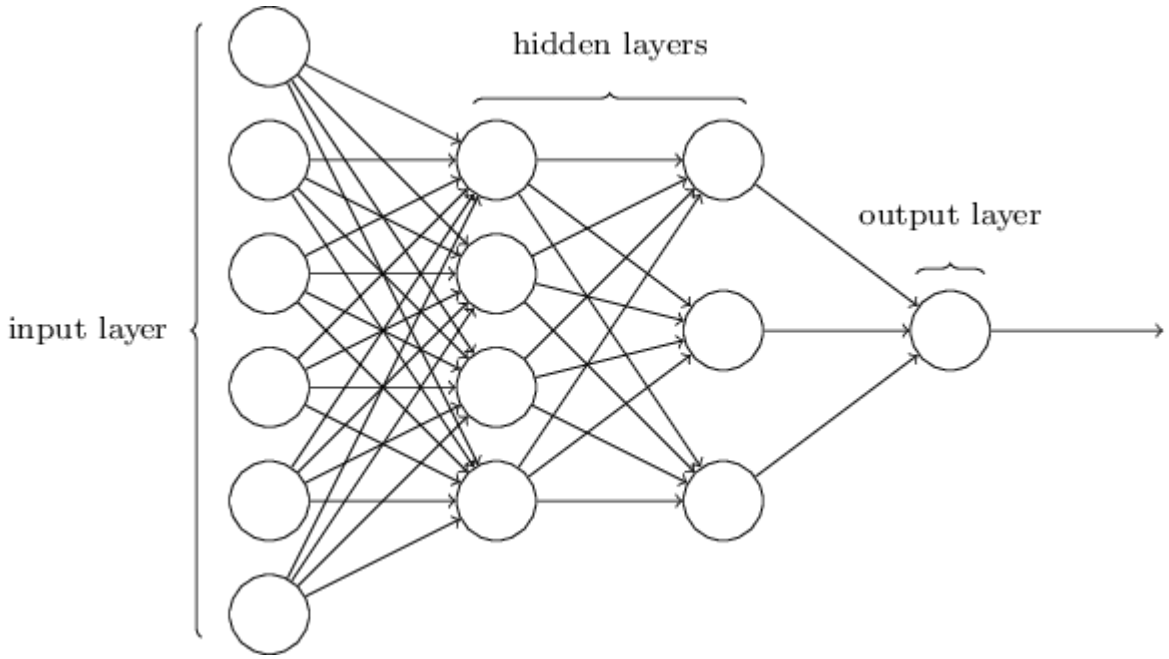
Внешний круг – это искусственный интеллект в целом (например, компьютеры). Чуть дальше – машинное обучение, а совсем в центре – глубинное обучение и искусственные нейросети.

Грубо говоря, глубинное обучение – просто более удобное название для искусственных нейросетей. «Глубинное» в этом словосочетании обозначает степень сложности (глубины) нейросети, которая зачастую может быть весьма поверхностной.

Создатели первой нейросети вдохновлялись структурой коры головного мозга. Базовый уровень сети, перцептрон, является по сути математическим аналогом биологического нейрона. И, как и в головном мозге, в нейросети могут появляться пересечённые друг с другом перцептроны.

Первый слой нейросети называется входным. Каждый узел этого слоя получает на вход какую-либо информацию и передает ее на последующие узлы в других слоях. Чаще всего между узлами одного слоя нет связей, а последний узел цепочки выводит результат работы нейросети.

Узлы посередине называются скрытыми, поскольку не имеют соединений с внешним миром, как узлы вывода и ввода. Они вызываются только в случае активации предыдущих слоев.



Глубинное обучение – это по сути техника обучения нейросети, которая использует множество слоев для решения сложных проблем (например, распознавания речи) с помощью шаблонов. В восьмидесятых годах большинство нейросетей были однослойными в силу высокой стоимости и ограниченности возможностей данных.

Если рассматривать машинное обучение как ответвление или вариант работы искусственного интеллекта, то глубинное обучение – это специализированный тип такого ответвления.

Машинное обучение использует компьютерный интеллект, который не дает ответа сразу. Вместо этого код будет запускаться на тестовых данных и, исходя из правильности их результатов, корректировать свой ход. Для успешности этого процесса обычно используются разнообразные техники, специальное программное обеспечение и информатика, описывающая статические методы и линейную алгебру.

9.2. Методы глубинного обучения

Методы глубинного обучения делятся на два основных типа:

- Обучение с учителем
- Обучение без учителя

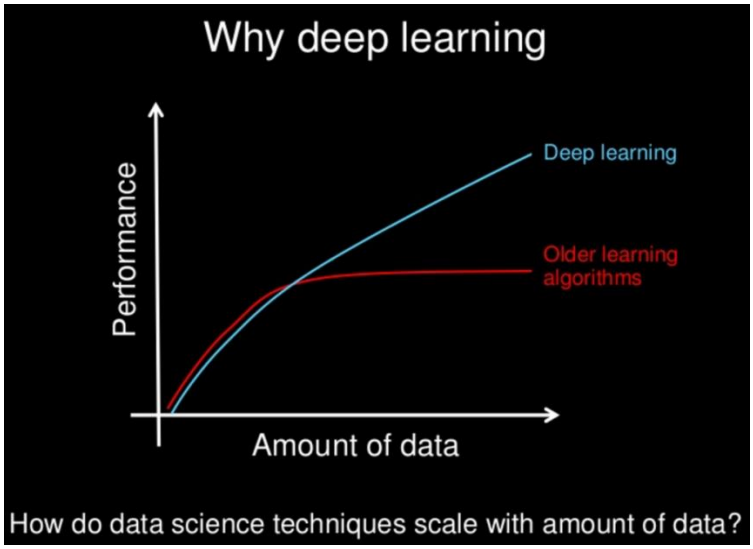
Первый способ использует специально отобранные данные, чтобы добиться желаемого результата. Он требует довольно много человеческого вмешательства, ведь данные приходится выбирать вручную. Однако он удобен для классификации и регрессии.

Представьте, что вы владелец компании и хотите определить влияние премий на продолжительность контрактов с вашими подчиненными. При наличии заранее собранных данных, метод обучения с учителем был бы незаменим и очень эффективен.

Второй же способ не подразумевает заранее заготовленных ответов и алгоритмов работы. Он направлен на выявление в данных скрытых шаблонов. Обычно его используют для кластеризации и ассоциативных задач, например для группировки клиентов по поведению. «С этим также выбирают» на Amazon – вариант ассоциативный задачи.

В то время как метод обучения с учителем довольно часто вполне удобен, его более сложный вариант все же лучше. Глубинное обучение зарекомендовало себя как нейросеть, не нуждающаяся в надзоре человека.

9.3. Важность глубинного обучения



Компьютеры уже давно используют технологии распознавания определенных черт на изображении. Однако результаты были далеки от успеха.

Компьютерное зрение оказало на глубинное обучение невероятное влияние. Именно эти две техники в данный момент решают все задачи на распознавание.

В частности, в распознавании лиц на фотографиях с помощью глубинного обучения преуспел Facebook. Это не простое улучшение технологии, а поворотный момент, изменяющий все более ранние представления: «Человек может с вероятностью в 97.53% определить, один ли человек представлен на двух разных фотографиях. Программа, разработанная командой Facebook, может делать это с вероятностью в 97.25% вне зависимости от освещения или того, смотрит ли человек прямо в камеру или повернут к ней боком».

Распознавание речи тоже претерпело значительные изменения. Команда Baidu – одного из лидирующих поисковиков Китая – разработала систему распознавания речи, сумевшую опередить человека в скорости и точности написания текста на мобильных устройствах. На английском и мандаринском.

Что особенно занимательно – написание общей нейросети для двух абсолютно разных языков не потребовало особенного труда: «Так исторически сложилось, что люди видели Китайский и Английский, как два совершенно разных языка, поэтому и подход к каждому из них требовался различный», — говорит начальник исследовательского центра Baidu, Andrew Ng. «Алгоритмы обучения сейчас настолько обобщены, что вы можете *просто* обучаться».

Google использует глубинное обучение для управления энергией в дата-центрах компании. Они смогли сократить затраты ресурсов для охлаждения на 40%. Это около 15% повышения эффективности энергопотребления и миллионы долларов экономии.

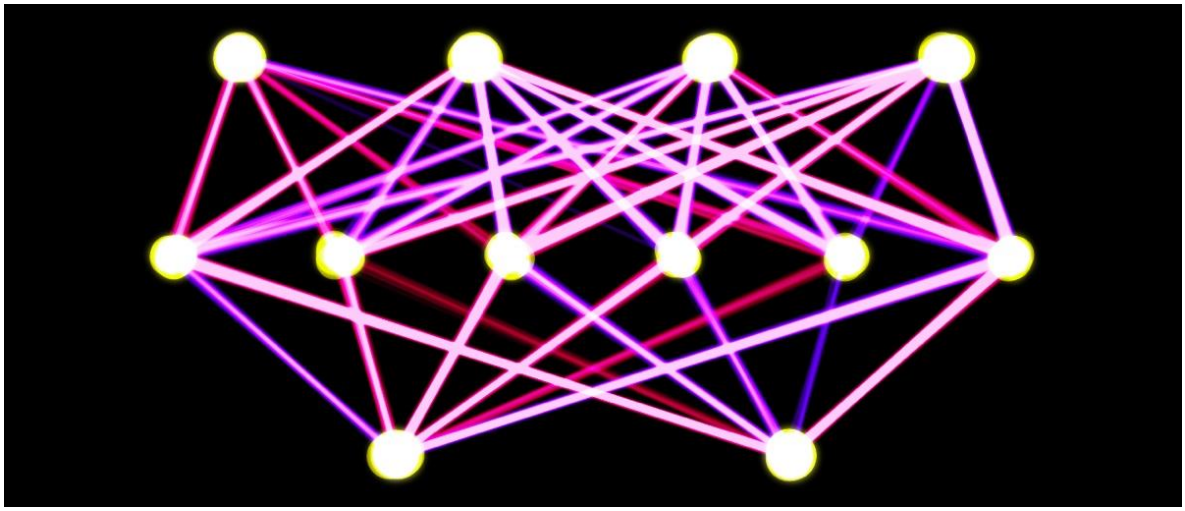
9.4 Микросервисы глубинного изучения

Вот краткий обзор сервисов, связанных с глубинным обучением.

Illustration Tagger. Дополненный Illustration2Vec, этот сервис позволяет отмечать изображения с рейтингом «защищенный», «сомнительный», «опасный», «копирайт» или «общий» для того, чтобы заранее понять содержание картинки.

Классификатор возраста использует технологии анализа фотографии для определения возраста человека. **Places 365 Classifier** использует заранее натренированную нейросеть в сочетании с базой данных за 2016 год для определения местоположение человека по фотографии (например, деревня, аптека, номер гостиной, горы и так далее). Не стоит забывать и о **InceptionNet** – прямом наследнике InceptionNet от Google. Эта нейросеть на основе анализа фотографии машины выдает пять лучших моделей, соответствующих этому автомобилю.

9.5. Open Source фреймворки о глубинном обучении



Доступность глубинного обучения обеспечена несколькими проектами с открытым исходным кодом. В этом списке есть как известные технологии, так и менее популярные. Он составлялся на основе направленности нейросети, сложности и акдемичности. Вот этот список:

DeepLearning4j(DL4J):

- Основана на JVM
- Свободное распространение
- Интегрируется с Hadoop и Spark

Theano:

- Популярна на Academia
- Сказочно простая

- Редактируется на Python и Numpy

Torch:

- Основана на Lua
- Домашняя версия используется компаниями Facebook и Twitter
- Содержит заранее натренированные модели

TensowFlow:

- Дополнение для Theano от Google
- Редактируется на Python и Numpy
- Зачастую применяется для решения определенного спектра проблем

Caffe:

- Не общего назначения. Основной упор на машинное зрение
- Редактируется на C++
- Есть интерфейс на Python

Здесь мы подробно рассматриваем Keras. Используйте Keras, если вам нужна библиотека глубокого обучения, которая:

1. Позволяет легко и быстро создавать прототипы (благодаря удобству, модульности и расширяемости).
2. Поддерживает как сверточные сети, так и повторяющиеся сети, а также комбинации этих двух.
3. Легко работает на процессоре и графическом процессоре.

9.6. Основные принципы Keras

Установка и настройка. Изначально Keras вырос как удобная надстройка над Theano. Отсюда и его греческое имя — κέρας, что значит "рог" по-гречески, что, в свою очередь, является отсылкой к Одиссее Гомера. Хотя, с тех пор утекло много воды, и Keras стал сначала поддерживать Tensorflow, а потом и вовсе стал его частью.

Keras устанавливается как обычный питоновский пакет:

pip install keras

ВНИМАНИЕ: Чтобы работать с Keras, у вас уже должен быть установлен хотя бы один из фреймворков — Theano или Tensorflow.

Бэкенды — это то, из-за чего Keras стал известен и популярен. Фронтенд (англ. front-end) — клиентская сторона пользовательского интерфейса к программно-аппаратной части сервиса. Бекенд (англ. back-end) — программно-аппаратная часть сервиса. Фронт- и бекенд — это вариант архитектуры программного обеспечения. Термины появились в программной инженерии вследствие развития принципа разделения ответственности между внешним представлением и внутренней реализацией. Back-end создает некоторое API, которое использует front-end. Таким образом front-end разработчику не нужно знать особенностей реализации сервера, а back-end разработчику — реализацию front-end. Keras позволяет использовать в качестве бэкенда разные другие фреймворки. При этом написанный код будет исполняться независимо от используемого бэкенда. Начиналась разработка, как уже было сказано, с Theano, но со временем добавился Tensorflow. Сейчас Keras по умолчанию работает именно с ним, но если нужно использовать Theano, то есть два варианта, как это сделать:

1. Отредактировать файл конфигурации keras.json, который лежит по пути $HOME/.keras/keras.json(или %USERPROFILE%\.keras\keras.json в случае операционных систем семейства Windows). Нам нужно поле backend:

```
{
"image_data_format": "channels_last",
"epsilon": 1e-07,
"floatx": "float32",
"backend": "theano"
}
```

2. Второй путь — это задать переменную окружения KERAS_BACKEND, например, так:

```
KERAS_BACKEND=theano python -c "from keras import backend"
Using Theano backend.
```

Удобство для пользователя. Keras - это API, предназначенный для людей, а не для машин. Он ставит пользовательский интерфейс спереди и в центре. Keras следует наилучшим методам снижения когнитивной нагрузки: он предлагает последовательные и простые API, он минимизирует количество действий пользователя, необходимых для случаев общего использования, и обеспечивает четкую и эффективную обратную связь с ошибкой пользователя.

Модульность. Под моделью понимается последовательность или график автономных полностью настраиваемых модулей, которые могут быть подключены вместе с минимальными ограничениями. В частности, нейронные слои, функции затрат, оптимизаторы, схемы инициализации, функции активации, схемы регуляризации - это автономные модули, которые вы можете комбинировать для создания новых моделей.

Легкая масштабируемость. Новые модули просто добавлять (как новые классы и функции), а существующие модули предоставляют множество примеров. Чтобы иметь возможность легко создавать новые модули, вы можете полностью выразить свою выразительность, что делает Keras подходящим для передовых исследований.

Работа с Python. Нет отдельных файлов конфигурации моделей в декларативном формате. Модели описаны в коде Python, который компактен, легче отлаживается и обеспечивает простоту расширяемости.

10. МОДЕЛИ KERAS

Основная структура данных Keras - это модель, способ организации слоев. В Keras доступны два основных типа моделей: последовательная модель **Sequential** и класс **Model**, используемый с функциональным API. Простейшим типом модели является **Sequential** модель, которая представляет собой линейную совокупность слоев. Для более сложных архитектур необходимо использовать функциональный API Keras, который позволяет создавать произвольные графики слоев.

Эти модели имеют ряд общих свойств и общих методов:

- **model.layers** - представляет собой список слоев, содержащихся в модели.
- **model.inputs** - представляет собой список входных тензоров модели.
- **model.outputs** - это список выходных тензоров модели.
- **model.summary()** - печатает сводное представление о модели.
- **model.get_config()** - возвращает словарь, содержащий конфигурацию модели.
- **model.get_weights()** - возвращает список всех весовых тензоров в модели.
- **model.set_weights(weights)** - устанавливает значения весов модели, из массива. Массивы в списке должны иметь ту же форму, что и возвращаемые **get_weights().**
- **model.to_json()** - возвращает представление модели как в виде строки JSON. Это представление не включает веса, а только архитектуру.

Приведем пример использования метода **model.to_json():**

```
from keras.models import model_from_json
json_string = model.to_json()
model = model.from_json(json_string)
```

10.1. API класса Model

Используя функциональный API можно создать экземпляр класса Model для некоторого входного тензора и выходной тензора используя следующий код:

```
from keras.models import Model
from keras.layers import Input, Dense
a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

Эта модель будет включать все уровни, необходимые для вычисления **b** на основе **a.**

В случае моделей с несколькими входами или с несколькими выходами также можно использовать списки:

model = Model(inputs=[a1, a2], outputs=[b1, b2, b3])

10.2. Основные методы класса Model

Рассмотрим наиболее важные методы класса **Model**, необходимые для организации процесса обучения нейронных сетей.

1. Метод настройки модели для обучения:

compile(self, optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_mode=None, weighted_metrics=None, target_tensors=None)

2. Обучение модели для определенного количества эпох:

fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None)

Основные аргументы этого метода:

- x: массив данных обучения (если модель имеет один вход) или список массивов (если модель имеет несколько входов).
- y: массив целевых данных (если модель имеет один вывод) или список массивов (если модель имеет несколько выходов).
- batch_size: количество выборок на обновление градиента. Если не указано, batch_size будет по умолчанию установлено значение 32.
- epochs: Количество эпох для обучения модели.
- validation_split: Float между 0 и 1. Доля данных обучения, которые будут использоваться в качестве данных валидации. Модель будет выделять эту часть данных обучения, не будет тренироваться на ней и будет оценивать ошибку и любые модельные показатели по этим данным в конце каждой эпохи.
- initial_epoch: эпоха, с которой начать обучение (полезно для возобновления предыдущего цикла обучения).

3. Метод для оценки качества обученности модели. Этот метод возвращает значения ошибок и показателей для модели в тестовом режиме.

evaluate(self, x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None)

4. Метод для создания выходных прогнозов для входных выборок.

predict(self, x, batch_size=None, verbose=0, steps=None)

Основные аргументы:

- x: входные данные, как в виде массива (или список массивов Numpy, если модель имеет несколько входов).
- steps: общее количество шагов (партии выборок) до объявления раунда прогнозирования.

5. Метод извлечения слоя на основе его имени или индекса. Этот метод возвращает экземпляр слоя.

get_layer(self, name=None, index=None)

Основные аргументы:

- name: String, имя слоя.
- index: Integer, индекс слоя.

11. СЛОИ В KERAS

Все слои Keras имеют ряд общих методов:

- **layer.get_weights()**- возвращает веса слоя в виде списка массивов Numpy.
- **layer.set_weights(weights)-** устанавливает веса слоя из списка массивов (с теми же формами, что и выход **get_weights**).
- **layer.get_config() -** возвращает словарь, содержащий конфигурацию слоя.

Слой может быть восстановлен из его конфигурации используя следующий:

layer = Dense(32)

config = layer.get_config()

reconstructed_layer = Dense.from_config(config)

Если слой имеет один узел (т. е. если он не является общим слоем), то можно получить его входной тензор, выходной тензор, размерность входного массива и размерность выходного массива через свойства:

- layer.input
- layer.output
- layer.input_shape
- layer.output_shape

11.1. Плотный слой Dense

Слой **Dense** реализует операцию**:** output = activation(dot(input, kernel) + bias) где **activation** функция активации, переданная в качестве **activation** аргумента, **kernel** является матрицей слоя весов, и **bias** представляет собой вектор смещения, созданный слоем.

Плотный слой создается использованием метода:

keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

Рассмотрим пример создания плотного слоя.

#сначала создаем последовательную модель

model = Sequential()

добавляем первый плотный слой

модель будет принимать на входе массив (*, 16) и выходной массив (*, 32)

model.add(Dense(32, input_shape=(16,)))

при добавлении следующих слоев нет необходимости указывать размеры входных массивов

model.add(Dense(32))

Чтобы указать функцию активации, которая будет применена к выходу необходимо использовать метод:

keras.layers.Activation(activation)

В качестве аргументы activation необходимо указать имя используемой функции активации.

Переобучение (**overfitting**) — одна из проблем глубоких нейронных сетей, состоящая в том, что модель хорошо распознает только примеры из обучающей выборки, адаптируясь к обучающим примерам, вместо того чтобы учиться классифицировать примеры, не участвовавшие в обучении (теряя способность к обобщению). Наиболее эффективным решением проблемы переобучения является метод исключения (Dropout).

keras.layers.Dropout(rate, noise_shape=None, seed=None)

Сети для обучения получаются с помощью исключения из сети (dropping out) нейронов с вероятностью **rate**, таким образом, вероятность того, что нейрон останется в сети, составляет **1- rate**. “Исключение” нейрона означает, что при любых входных данных или параметрах он возвращает 0.

Для преобразования результата в определенную форму необходимо использовать метод:

keras.layers.Reshape(target_shape)

В качестве аргумента target_shape указывается кортеж целых чисел.

Рассмотрим пример:

model.add(Reshape((3, 4), input_shape=(12,)))

размерность массива выходного слоя: model.output_shape == (None, 3, 4)

model.add(Reshape((6, 2)))

размерность массива выходного слоя: model.output_shape == (None, 6, 2)

Для изменения размеров входного массива можно использовать метод:

keras.layers.**Permute**(dims)

Этот метод полезен, например, для соединения RNN и коннектов вместе.

Пример

model = Sequential()

model.add(Permute((2, 1), input_shape=(10, 64)))

11.2. Сверточные слои

Слой свёртки — это основной блок свёрточной нейронной сети. Слой свёртки включает в себя для каждого канала свой фильтр, ядро свёртки которого обрабатывает предыдущий слой по фрагментам (суммируя результаты матричного произведения для каждого фрагмента). Весовые коэффициенты ядра свёртки (небольшой матрицы) неизвестны и устанавливаются в процессе обучения. Особенностью свёрточного слоя является сравнительно небольшое количество параметров, устанавливаемое при обучении.

1. Conv1D - Этот слой создает сверточное ядро, по одному пространственному (или временному) измерению:

keras.layers.Conv1D(filters, kernel_size, strides=1, padding='valid', data_format='channels_last', dilation_rate=1, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

Основные аргументы:

- filters: размерность выходного пространства (т. е. количество выходных фильтров в свертке).
- kernel_size: целое или список целых чисел, определяющий длину окна свертки.
- strides: целое или список целых чисел, определяющий длину шага свертки.
- activation: функция активации слоя. Если этот параметр не указан, то активация не применяется (т.е. «линейная» функция активации a(x) = x).

2. Conv2D – это 2D сверточный слой (например, пространственная свертка над изображениями). Этот слой создает ядро свертки для создания тензора выходов.

keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True,

kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

3. Conv3D - 3D сверточный слой (например, пространственная свертка над объемами). Этот слой создает ядро свертки, которое свернуто со слоем ввода для создания тензора выходов.

keras.layers.Conv3D(filters, kernel_size, strides=(1, 1, 1), padding='valid', data_format=None, dilation_rate=(1, 1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None).

При создании этого слоя если use_bias= True, тогда вектор смещения создается и добавляется к выходу. При использовании этого слоя в качестве первого слоя в модели в качестве аргумента input_shape необходимо указать кортеж целых чисел, который не включает ось выборки, например, input_shape=(128, 128, 128, 1).

11.3. Слой пулинга

Слой пулинга представляет собой нелинейное уплотнение карты признаков, при этом группа пикселей (обычно размера 2×2) уплотняется до одного пикселя, проходя нелинейное преобразование. Преобразования затрагивают непересекающиеся прямоугольники или квадраты, каждый из которых ужимается в один пиксель, при этом выбирается пиксель, имеющий максимальное значение. Операция пулинга позволяет существенно уменьшить пространственный объём изображения. Пулинг интерпретируется так. Если на предыдущей операции свёртки уже были выявлены некоторые признаки, то для дальнейшей обработки настолько подробное изображение уже не нужно, и оно уплотняется до менее подробного. К тому же фильтрация уже ненужных деталей помогает не переобучаться. Слой пулинга, как правило, вставляется после слоя свёртки перед слоем следующей свёртки.

Наиболее употребительна при этом функция максимума.

1. MaxPooling1D

keras.layers.MaxPooling1D(pool_size=2, strides=None, padding='valid')

Основные аргументы:

- pool_size: Integer, размер максимальных окон объединения.
- strides: параметр, с помощью которого можно уменьшить масштаб. Например, 2 уменьшит вдвое вход.

2. MaxPooling2D

keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None)

Операция объединения для пространственных данных.

3. MaxPooling3D

keras.layers.MaxPooling3D(pool_size=(2, 2, 2), strides=None, padding='valid', data_format=None)

Операция объединения трехмерных данных (пространственное или пространственно-временное объединение).

12. ОСНОВЫ РАБОТЫ С ПОСЛЕДОВАТЕЛЬНОЙ МОДЕЛЬЮ KERAS

Создать последовательную модель (**Sequential модель**) можно передав список экземпляров слоя в конструктор класса **Sequential**:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
model = Sequential([Dense(32, input_shape=(784,)), Activation('relu'), Dense(10), Activation('softmax'),])
```

Также можно просто добавить слои с помощью метода **add()**.

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

12.1. Указание размерности входных данных

Модель должна знать размерность входного массива данных. Поэтому первый слой в **Sequential** модели (и только первый, потому что следующие слои могут получать автоматически эту информацию из предыдущего слоя) должен получать информацию о размерность входного массива. Существует несколько способов сделать это:

- Передать аргумент **input_shape** первому слою.
- Некоторые 2D-слои, например, **Dense**, поддерживают спецификацию их формы ввода через аргумент **input_dim**, а некоторые 3D-слои поддерживают аргументы **input_dim** и **input_length**.

Таким образом, следующие фрагменты кода строго эквивалентны:

model.add(Dense(32, input_shape=(784,)))

model.add(Dense(32, input_dim=784))

12.2. Компиляция

Перед подготовкой модели необходимо настроить процесс обучения, который выполняется с помощью метода **compile**. Этом метод имеет три аргумента:

- **Оптимизатор.** Это может быть строковый идентификатор существующего оптимизатора (например, **rmsprop** или **adagrad**) или экземпляр класса **Optimizer**.
- **Функция вычисления ошибок.** Это цель, которую модель попытается свести к минимуму. Он может быть строковым идентификатором существующей функции ошибок (например, **categorical_crossentropy** или **mse**) или может быть целевой функцией.
- **Список метрик.** Для любой проблемы классификации вы захотите установить это metrics=['accuracy']. Метрика может быть строковым идентификатором существующей метрики или специальной метрической функцией.

Примеры:

1. Для задачи классификации по нескольким классам

model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

2. Для задачи бинарной классификации

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

3. Для регрессионный задачи со среднеквадратичной ошибкой

model.compile(optimizer='rmsprop', loss='mse')

Пример создания и использования собственной метрики:

```
import keras.backend as K
def mean_pred(y_true, y_pred):
    return K.mean(y_pred)
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy', mean_pred])
```

12.3. Обучение

Модели Keras обучаются массивам входных данных и целевых значении. Для этого необходимо использовать функцию **fit**.

Рассмотрим пример создания модели для бинарной классификации.

```
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
# Создаем фиктивные данные
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))
# Обучаем модель в 10 эпох по 32 примера в пакете
model.fit(data, labels, epochs=10, batch_size=32)
```

12.4. Пример многослойного перцептрона (MLP) для многоклассовой классификации

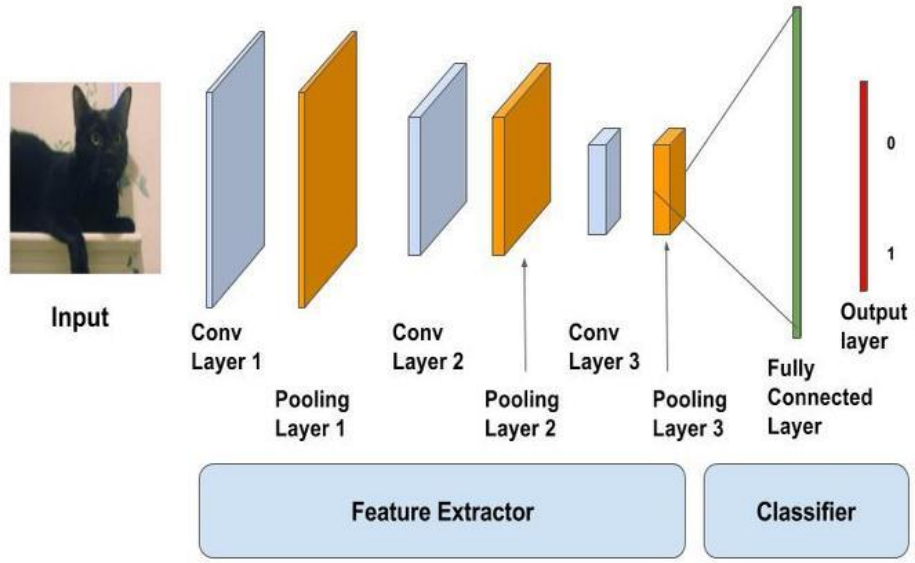
```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD
import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)),
num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)),
num_classes=10)
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',  optimizer=sgd,
metrics=['accuracy'])
model.fit(x_train, y_train,  epochs=20,  batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

13. КЛАССИФИКАЦИЯ ИЗОБРАЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ СВЕРТОЧНЫХ НЕЙРОННЫХ СЕТЕЙ В KERAS

13.1. Сверточная нейронная сеть

Сверточные нейронные сети являются одной из форм многослойных нейронных сетей. Здесь приведена схема типичного CNN. Первая часть состоит

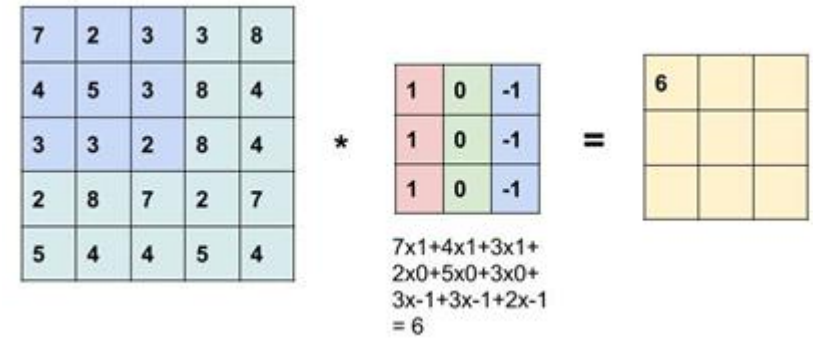
из слоев свертки и максимального пула, которые выступают в качестве экстрактора признаков. Вторая часть состоит из полносвязного слоя, который выполняет нелинейные преобразования извлеченных признаков и действует как классификатор.



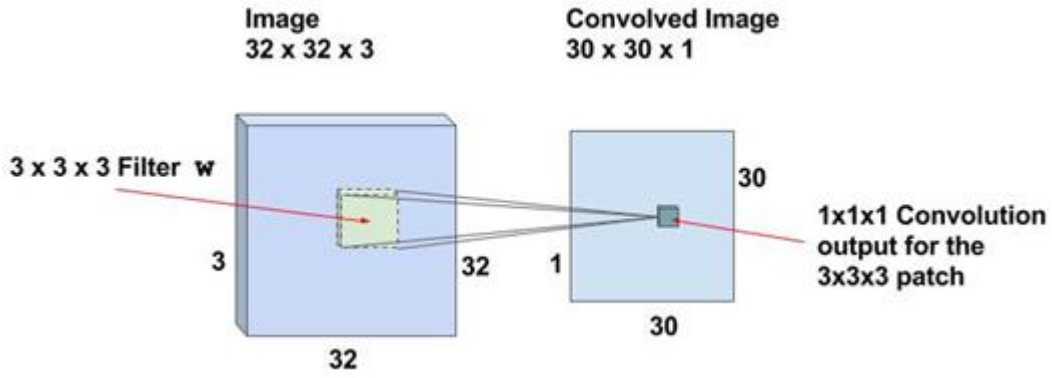
На приведенной выше диаграмме вход подается в сеть последовательных слоев **Conv**, **Pool** и **Dense**. Выходной сигнал может быть слоем **softmax**, указывающим, есть ли кошка или что-то еще. Также, а качестве выходного может быть использован сигмоидный слой, на выходе которого будет вероятность того, что изображение будет кошкой. Рассмотрим слои более подробно.

Сверточный слой можно рассматривать как глаза сверточной нейронной сети. Нейроны в этом слое ищут определенные особенности. Свертку можно рассматривать как взвешенную сумму между двумя сигналами или функциями. Пример операции свертки на матрице размером 5 × 5 с ядром размером 3 × 3 показан ниже. Ядро свертки скользит по всей матрице для

получения карты активации.



Предположим, что входное изображение имеет размер 32x32x3, т.е. это трехмерный массив глубины 3. Любой фильтр свертки, который мы определяем на этом слое, должен иметь глубину, равную глубине ввода. Поэтому мы можем выбрать фильтры свертки глубины 3 (например, 3x3x3 или 5x5x3 или 7x7x3 и т. Д.). Выберем фильтр свертки размера 3x3x3, т.е сверточное ядро будет кубом вместо квадрата.



Если мы сможем выполнить операцию свертки, сдвинув фильтр 3x3x3 на все изображение размером 32x32x3, то мы получим изображение с разрешением 30x30x1. Это связано с тем, что операция свертки невозможна для полосы шириной 2 пикселя вокруг изображения. Фильтр всегда находится внутри изображения и поэтому 1 пиксель удаляется от левой, правой, верхней и нижней части изображения.

Для входного изображения 32x32x3 и размера фильтра 3x3x3 у нас есть 30x30x1 местоположения, и для каждого местоположения существует нейрон. Тогда выходы 30x30x1 или активации всех нейронов называются

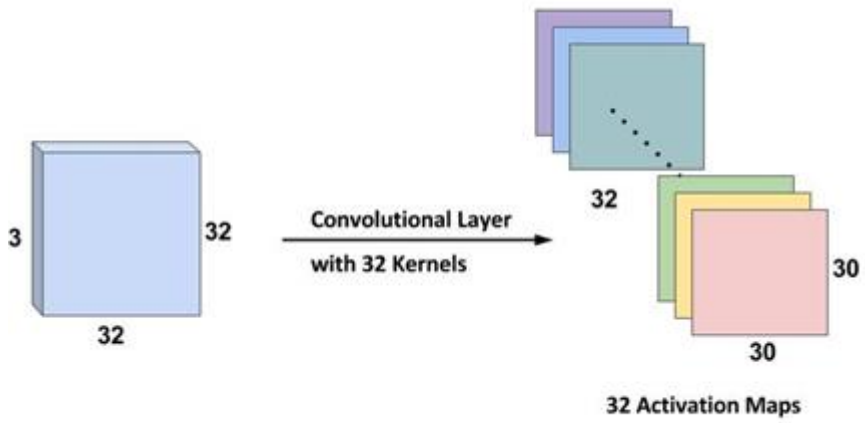
картами активации. Карта активации одного уровня служит входом для следующего слоя.

В нашем примере есть 30 × 30 = 900 нейронов, потому что есть много мест, где может применяться фильтр 3x3x3. В отличие от традиционных нейронных сетей, где веса и смещения нейронов независимы друг от друга, в случае сверточных нейронных сетей, нейроны, соответствующие одному фильтру в слое, имеют одинаковые веса и смещения. В приведенном выше случае мы сдвигаем окно на 1 пиксель за раз. Мы также можем сдвинуть окно более чем на 1 пиксель. Это число называется шагом.

Как правило, используют более одного фильтра в одном слое свертки. Если мы используем 32 фильтра, у нас будет карта активации размером 30x30x32.

Обратите внимание, что все нейроны, связанные с одним и тем же фильтром, имеют одинаковые веса и смещения. Таким образом, количество весов при использовании 32 фильтров - это 3x3x3x32 = 288, а число смещений - 32.

На картинке показаны 32 карты активации, полученные от применения

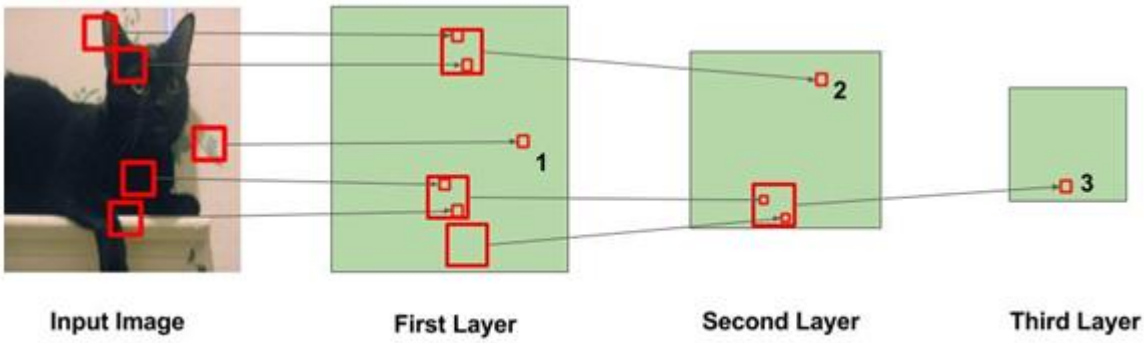


сверточных ядер.

Как вы можете видеть, после каждой свертки результат уменьшается по размеру (так как в этом случае мы переходим от 32 × 32 до 30 × 30). Для удобства стандартная практика заключается в том, чтобы накладывать нули на границу входного слоя таким образом, чтобы выход был такого же размера, как и входной. Итак, в этом примере, если мы добавим дополнение размером 1 по

обе стороны от входного слоя, размер выходного уровня будет 32x32x32, что упростит реализацию.

Рассмотрим, как сверточные нейронные сети анализируют изображения.



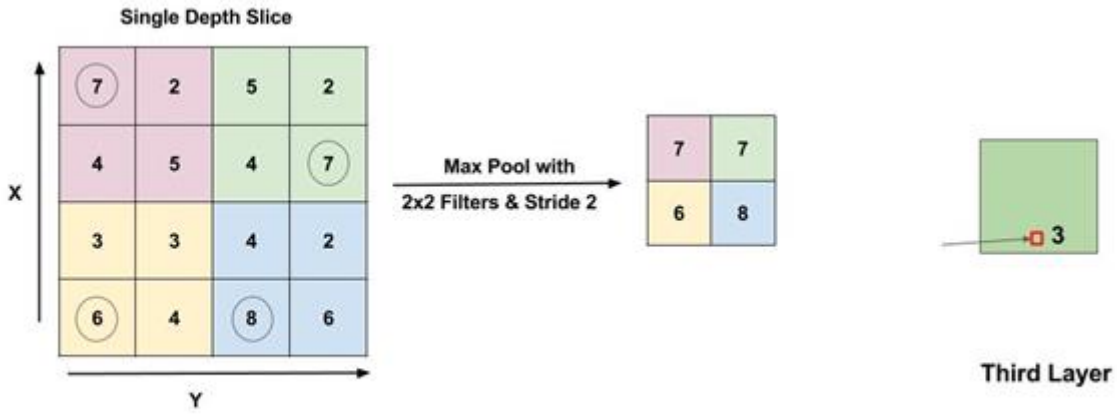
На приведенном выше рисунке большие квадраты указывают область, в которой выполняется операция свертки, а малые квадраты указывают выход операции, которая является просто числом. Следует отметить следующие замечания:

- В первом слое квадрат, обозначенный 1, получается из области изображения, на которой окрашены листья.
- Во втором слое квадрат с меткой 2 получается из большего квадрата в первом слое. Числа в этом квадрате получены из нескольких областей из входного изображения. В частности, вся площадь вокруг левого уха кошки отвечает за значение на квадрате, отмеченном 2.
- Аналогично, в третьем слое этот каскадный эффект приводит к тому, что квадрат, обозначенный 3, получается из большой области вокруг области ноги.

Из сказанного выше можно сказать, что начальные слои анализируют более мелкие области изображения и, следовательно, могут обнаруживать только простые признаки, такие как края / углы и т. д. По мере того как мы идем глубже в сеть, нейроны получают информацию из более крупных частей изображения и от различных других нейронов. Таким образом, нейроны на более поздних слоях могут изучить более сложные функции, такие как глаза, ноги, и т.д

Слой пулинга в основном используется сразу после сверточного слоя для уменьшения пространственного размера (только по ширине и высоте, а не по глубине). Это уменьшает количество параметров, поэтому вычисление уменьшается. Использование меньшего количества параметров позволяет избежать переобучения. **Переобучение** - это условие, когда обученная модель отлично работает с данными обучения, но не очень хорошо работает в тестовых данных.

Наиболее распространенной формой пулинга является максимальный пулинг, в котором мы берем фильтр размера p и применяем максимальную операцию **max** с определенной частью изображения.



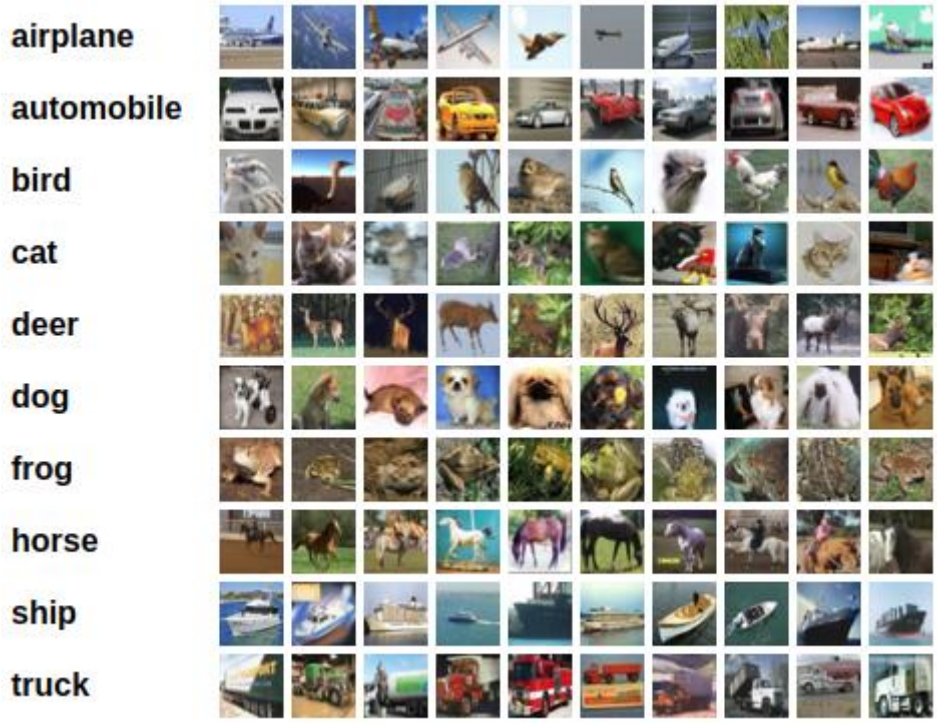
На рисунке показан максимальный пул с размером фильтра 2 × 2 и шагом 2. Выход представляет собой максимальное значение в области 2 × 2, показанной с использованием окруженных цифр. Наиболее распространенная операция пулинга выполняется с фильтром размером 2 × 2 с шагом 2. Это существенно уменьшает размер ввода на половину.

13.2. Набор данных - CIFAR10

Набор данных CIFAR10 поставляется вместе с Keras. Он имеет 50 000 учебных образов и 10000 тестовых изображений 10 классов, таких как самолеты, автомобили, птицы, кошки, олени, собаки, лягушки, лошади, корабли и грузовики. С помощью следующего програмного кода можно осуществить загрузку и подготовку данных CIFAR10 для дальнейшей обработки с помощью нейронных сетей:

```
from __future__ import print_function
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
import os
batch_size = 32
num_classes = 10
epochs = 100
num_predictions = 20
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'keras_cifar10_trained_model.h5'
# Разделяем данные на обучающий и тестовый наборы:
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
# Преобразование векторов классов в двоичные матрицы
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

Изображения имеют размер 32 × 32. На рисунке приведены несколько примеров.



Сверточная нейронная сеть будет состоять из сверточных слоев, и слоев **MaxPooling**. Мы также включим **Dropout** слой для избежания переобучения. На выходе сети мы добавим полносвязный слой (**Dense**), за которым следует слой **softmax**. Здесь приведен программный код создания структуры модели. В приведенном выше коде мы используем 6 сверточных слоев и 1 полносвязный слой. Сначала в модель добавляем сверточные слои с 32 фильтрами с размером окна 3 × 3. Далее мы добавляем сверточный слой с 64 фильтрами. За каждым слоем добавлен слой максимального пуллинга с размером окна 2 × 2. Также добавлены слои **Dropout** с коэффициентами 0,25 и 0.5 для того чтобы не произошло переобучение сети. В заключительных строках мы добавляем плотный слой **Dense**, который выполняет классификацию среди 10 классов с использованием функции активации **softmax.**

model = Sequential()

model.add(Conv2D(32,(3,3),padding='same', input_shape=x_train.shape[1:]))

model.add(Activation('relu'))

model.add(Conv2D(32, (3, 3)))

model.add(Activation('relu'))

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
model.summary()
```

Если мы выведем информацию о структуре модели, мы увидим следующую таблицу с подробным описанием каждого слоя.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
activation_1 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
activation_2 (Activation)    (None, 30, 30, 32)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 15, 32)        0
```

_____

dropout_1 (Dropout) (None, 15, 15, 32) 0

_____

conv2d_3 (Conv2D) (None, 15, 15, 64) 18496

_____

activation_3 (Activation) (None, 15, 15, 64) 0

_____

conv2d_4 (Conv2D) (None, 13, 13, 64) 36928

_____

activation_4 (Activation) (None, 13, 13, 64) 0

_____

max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64) 0

_____

dropout_2 (Dropout) (None, 6, 6, 64) 0

_____

flatten_1 (Flatten) (None, 2304) 0

_____

dense_1 (Dense) (None, 512) 1180160

_____

activation_5 (Activation) (None, 512) 0

_____

dropout_3 (Dropout) (None, 512) 0

_____

dense_2 (Dense) (None, 10) 5130

_____

activation_6 (Activation) (None, 10) 0

==========

====

Total params: 1,250,858

Trainable params: 1,250,858

Non-trainable params: 0

_____

13.3. Обучение сети

Поскольку это проблема классификации по 10 классам, мы будем использовать категорическую потерю энтропии и использовать оптимизатор **RMSProp** для обучения сети.

```
# initiate RMSprop optimizer
opt = keras.optimizers.rmsprop(lr=0.0001, decay=1e-6)
# Let's train the model using RMSprop
model.compile(loss='categorical_crossentropy',optimizer=opt,metrics=['accuracy'])
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

//Запустим его на количество эпох **epochs**.

```
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test, y_test), shuffle=True)

# Сохраняем модель и веса
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
model_path = os.path.join(save_dir, model_name)
model.save(model_path)
print('Saved trained model at %s ' % model_path)

# Проверяем точность работы модели.
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

14. РАСПОЗНАВАНИЕ РУКОПИСНЫХ ЦИФР С ИСПОЛЬЗОВАНИЕМ СВЕРТОЧНЫХ НЕЙРОННЫХ СЕТЕЙ

MNIST - это набор данных, разработанный Янном ЛеКуном, Коринной Кортес и Кристофер Бургес для оценки моделей машинного обучения по проблеме классификации рукописных цифр. Набор данных был построен из ряда отсканированных наборов документов, доступных в Национальном институте стандартов и технологий (NIST). Изображения цифр были взяты из множества отсканированных документов, нормированных по размеру и по центру. Это делает его отличным набором данных для оценки моделей, позволяя разработчику сосредоточиться на механизме обучения с очень небольшой очисткой данных или необходимой подготовкой.

Каждое изображение представляет собой квадрат размером 28 на 28 пикселей (всего 784 пикселя). В этом наборе 60 000 изображений используются для обучения модели, и для ее тестирования используется отдельный набор из 10 000 изображений. Это задача распознавания 10 цифр (от 0 до 9) или классификация на 10 классов.

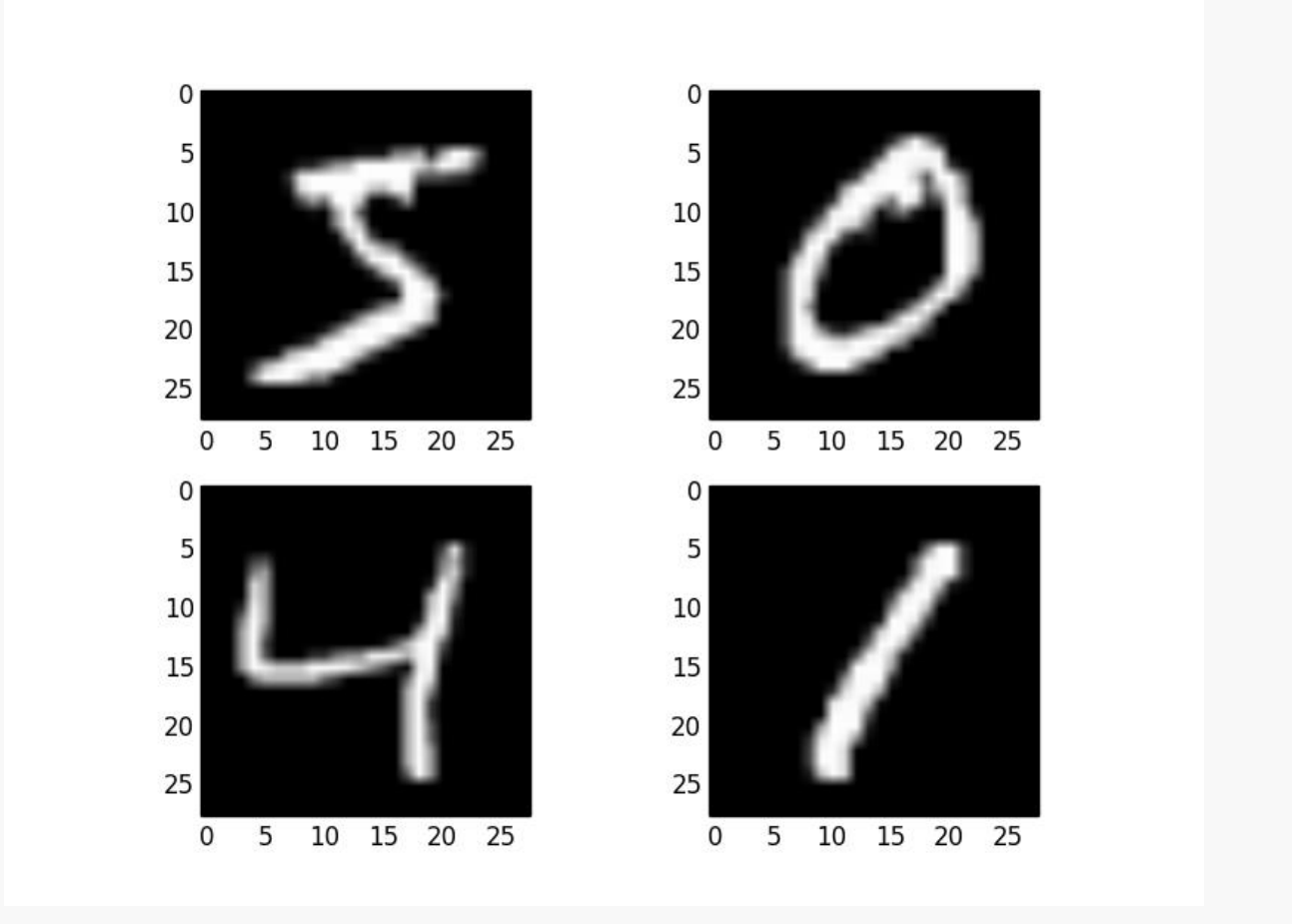
Библиотека глубокого обучения Keras предоставляет удобный метод **mnist.load_data()** для загрузки набора данных MNIST. Набор данных загружается автоматически при первом вызове этой функции и сохраняется в вашем домашнем каталоге в ~ / .keras /datasets /mnist.pkl.gz в виде файла 15 МБ. Это очень удобно для разработки и тестирования моделей глубокого обучения.

Чтобы продемонстрировать, насколько легко загружать набор данных MNIST, мы сначала напишем небольшой скрипт для загрузки и визуализации первых четырех изображений в наборе учебных материалов.

```
from keras.datasets import mnist
import matplotlib.pyplot as plt
# load (downloaded if needed) the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# plot 4 images as gray scale
plt.subplot(221)
```

```
plt.imshow(X_train[0], cmap=plt.get_cmap('gray'))
plt.subplot(222)
plt.imshow(X_train[1], cmap=plt.get_cmap('gray'))
plt.subplot(223)
plt.imshow(X_train[2], cmap=plt.get_cmap('gray'))
plt.subplot(224)
plt.imshow(X_train[3], cmap=plt.get_cmap('gray'))
# show the plot
 plt.show()
```

Запустив приведенный выше пример, вы должны увидеть изображение ниже.



14.1. Базовая модель с многослойным перцептроном

Чтобы понять действительно ли нам нужна сложная модель, такая как сверточная нейронная сеть сначала попробуем использовать очень простую модель нейронной сети с одним скрытым слоем. Мы будем использовать эту

сеть как основу для сравнения более сложных сверточных моделей нейронных сетей.

Начнем с импорта классов и функций, которые нам понадобятся.

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
 from keras.utils import np_utils
```

Учебный набор данных структурирован как трехмерный массив. Чтобы подготовить данные, сперва мы представим изображения в виде одномерных массивов (так как считаем каждый пиксель отдельным входным признаком). В этом случае изображения размером 28 × 28 будут преобразованы в массивы, содержащие 784 элементов.

Мы можем сделать это преобразование, используя функцию **reshape()** библиотеки **NumPy**. Для уменьшения потребления оперативной памяти преобразуем точность значений пикселей в 32.

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
 X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
```

Значения пикселей заданы в оттенках серого со значениями от 0 до 255. Для эффективного обучения нейронных сетей практически всегда рекомендуется выполнять некоторое масштабирование входных значений. Мы можем нормализовать значения пикселей в диапазоне 0 и 1, разделив каждое значение на максимальные значения 255.

```
X_train = X_train / 255
 X_test = X_test / 255
```

Выходная переменная представляет собой целое число от 0 до 9, т.к. это задача классификации с несколькими классами. Хорошей практикой является

использование кодирования значений класса преобразованием вектора целых чисел класса в двоичную матрицу.

Мы можем легко сделать это, используя встроенную вспомогательную функцию **np_utils.to_categorical ()** в Keras.

```
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Теперь создадим нашу простую модель однослойной нейронной сети и определим ее в функции.

```
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels,input_dim=num_pixels,
kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_classes,kernel_initializer='normal',
activation='softmax'))
    model.compile(loss='categorical_crossentropy',optimizer='adam',
metrics=['accuracy'])
  return model
```

Модель представляет собой простую нейронную сеть с одним скрытым слоем с таким же количеством нейронов, что и количество входов (784). В скрытом слое используем полулинейную функцию активации **relu**.

На выходном слое используется функция активации **softmax** для преобразования выходов в вероятностные значения и позволяет выбрать один класс из 10 в качестве выходного значения модели.
Теперь нам осталось только определить функцию потерь, алгоритм оптимизации и метрики, которые мы будет собирать. В задачах с вероятностной классификацией, в качестве функции потерь лучше всего использовать не квадратичную ошибку, а перекрестную энтропию. Потери будут меньше для вероятностных задач (например, с логистической/**softmax**

функцией для выходного слоя), в основном из-за того, что данная функция предназначена для максимизации уверенности модели в правильном определении класса, и ее не заботит распределение вероятностей попадания образца в другие классы. Используемый алгоритм оптимизации будет напоминать какую-то форму алгоритма градиентного спуска, отличие будет лишь в том, как выбирается скорость обучения. В нашем случае мы будем использовать оптимизатор Адама, который обычно показывает хорошую производительность. Так как наши классы сбалансированы (количество рукописных цифр, принадлежащих каждому классу, одинаково), подходящей метрикой будет точность (**accuracy**) — доля входных данных, отнесенных к правильному классу.

Теперь мы можем обучить и оценить качество обученности модели.

```
model = baseline_model()
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200, verbose=2)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

Модель подходит 10 эпох обучения, при каждом обновлении весов используется 200 изображений. Тестовые данные которые используются в качестве набора данных валидации, позволяют вам видеть качество распознавания модели по мере ее обучения. Значение **verbose** =2 используется для уменьшения вывода на одну строку для каждой учебной эпохи. Наконец, тестовый набор данных используется для оценки модели и печатается ошибка классификации.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10  - 21s - loss: 0.2781 - acc: 0.9213 - val_loss: 0.1443 - val_acc: 0.9585
Epoch 2/10  - 21s - loss: 0.1100 - acc: 0.9686 - val_loss: 0.0943 - val_acc: 0.9709
Epoch 3/10 - 18s - loss: 0.0709 - acc: 0.9798 - val_loss: 0.0809 - val_acc: 0.9739
Epoch 4/10  - 18s - loss: 0.0511 - acc: 0.9855 - val_loss: 0.0679 - val_acc: 0.9781
Epoch 5/10  - 18s - loss: 0.0361 - acc: 0.9898 - val_loss: 0.0650 - val_acc: 0.9801
```

Epoch 6/10 - 18s - loss: 0.0265 - acc: 0.9936 - val_loss: 0.0640 - val_acc: 0.9790

Epoch 7/10 - 18s - loss: 0.0191 - acc: 0.9953 - val_loss: 0.0624 - val_acc: 0.9810

Epoch 8/10 - 18s - loss: 0.0145 - acc: 0.9965 - val_loss: 0.0592 - val_acc: 0.9822

Epoch 9/10 - 18s - loss: 0.0109 - acc: 0.9977 - val_loss: 0.0554 - val_acc: 0.9827

Epoch 10/10 - 18s - loss: 0.0079 - acc: 0.9986 - val_loss: 0.0596 - val_acc: 0.9814

Baseline Error: 1.86%

Как видно, наша модель достигает точности приблизительно 98.14% и ошибки 1.86% на тестовом наборе данных, это вполне достойно для такой простой модели.

14.2. Простая сверточная нейронная сеть для MNIST

Мы узнали, как загрузить набор данных MNIST и как запрограммировать простую многослойную модель персептрона, и теперь настало время разработать более сложную сверточную нейронную сеть.

В этом разделе мы создадим простую CNN для MNIST, которая продемонстрирует, как использовать все аспекты современной реализации CNN.

Первый шаг - импортировать необходимые классы и функции.

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
 K.set_image_dim_ordering('th')
```

Далее инициализируем генератор случайных чисел на постоянное начальное значение для воспроизводимости результатов.

seed = 7

numpy.random.seed(seed)

Затем нам нужно загрузить набор данных MNIST и изменить его, чтобы он был подходящим для обучения CNN.

(X_train, y_train), (X_test, y_test) = mnist.load_data()

 # reshape to be [samples][pixels][width][height]

X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')

X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')

Как и прежде нормализуем значения пикселей в диапазоне 0 и 1.

X_train = X_train / 255

X_test = X_test / 255

y_train = np_utils.to_categorical(y_train)

y_test = np_utils.to_categorical(y_test)

num_classes = y_test.shape[1]

Затем мы определяем модель нейронной сети. Сверточные нейронные сети более сложны, чем стандартные многослойные персептроны, поэтому мы начнем с использования простой структуры

Ниже представлена архитектура сети.

1. Первый скрытый слой - это сверточный слой, Convolution2D. Этот слой имеет 32 карты функций, размер которых равен 5 × 5 и функции активации **relu**.
2. Затем мы определяем слой пулинга **maxPooling2D** с размером пула 2 × 2, который дает максимальные значения.
3. Следующий уровень - это уровень регуляризации **Dropout**. Он настроен на случайное исключение 20% нейронов в слое, чтобы уменьшить переобучение.
4. Далее - слой, который преобразует данные двумерной матрицы в вектор, называемый **Flatten**. Он позволяет обрабатывать выходные данные стандартными полносвязными слоями.
5. Затем полносвязный слой с 128 нейронами и функцией активации **relu**.

6. Наконец, выходной слой имеет 10 нейронов для 10 классов и функцию активации **softmax** для вывода вероятностных результатов распознавания для каждого класса.

```
def baseline_model():
    # create model
    model = Sequential()
    model.add(Conv2D(32, (5, 5), input_shape=(1, 28, 28), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
  return model
```

Как и в примере с моногослойным персептроном эта модель 10 эпох обучения, при каждом обновлении весов используется 200 изображений.

```
model = baseline_model()
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200, verbose=2)
scores = model.evaluate(X_test, y_test, verbose=0)
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

Точность классификации модели печатается в каждую эпоху обучения и в конце отпечатается ошибка ошибки классификации.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10  - 276s - loss: 0.2224 - acc: 0.9366 - val_loss: 0.0783 - val_acc: 0.9754
Epoch 2/10  - 279s - loss: 0.0710 - acc: 0.9789 - val_loss: 0.0454 - val_acc: 0.9846
Epoch 3/10 - 449s - loss: 0.0510 - acc: 0.9841 - val_loss: 0.0444 - val_acc: 0.9854
Epoch 4/10 - 267s - loss: 0.0389 - acc: 0.9881 - val_loss: 0.0403 - val_acc: 0.9876
```

Epoch 5/10 - 269s - loss: 0.0325 - acc: 0.9898 - val_loss: 0.0349 - val_acc: 0.9883
Epoch 6/10 - 313s - loss: 0.0267 - acc: 0.9919 - val_loss: 0.0321 - val_acc: 0.9896
Epoch 7/10 - 255s - loss: 0.0220 - acc: 0.9930 - val_loss: 0.0339 - val_acc: 0.9888
Epoch 8/10 - 271s - loss: 0.0192 - acc: 0.9939 - val_loss: 0.0329 - val_acc: 0.9896
Epoch 9/10 - 266s - loss: 0.0157 - acc: 0.9951 - val_loss: 0.0323 - val_acc: 0.9891
Epoch 10/10 - 279s - loss: 0.0145 - acc: 0.9956 - val_loss: 0.0333 - val_acc: 0.9889
CNN Error: 1.11%

Обучения сверточной нейронной сети занимает больше времени чем обучение простого персептрона, рассмотренного выше. Однако ошибка достигает 1.11%, что значительно меньше по сравнению с персепроном.

14.3. Большая сверточная нейронная сеть для MNIST

Теперь, когда мы увидели, как создать простую сверточную нейронную сеть, давайте создадим модель, которая может быть близка к новейшим научным результатам.

В начале программы импортируем классы и функции, затем загружаем и готовим данные так же, как в предыдущем примере CNN.

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
K.set_image_dim_ordering('th')
seed = 7
```

```
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
 num_classes = y_test.shape[1]
```

На этот раз мы создадим большую архитектуру сверточной нейронной сети с дополнительными сверточными слоями, слоями пуллинга и полностью связанными слоями. Сетевую топологию можно резюмировать следующим образом.

1. Сверточный слой **Conv2D** с 30 функциональными картами размером 5 × 5.
2. Слой максимального пулинга **MaxPooling2D** размером 2 * 2.
3. Сверточный слой **Conv2D** с 15 картинными картами размером 3 × 3.
4. Слой максимального пулинга **MaxPooling2D** размером 2 * 2.
5. Слой исключения **Dropout** с вероятностью 20%.
6. Слой **Flatten**.
7. Полносвязный слой **Dense** с 128 нейронами и функцией активации **relu**.
8. Полносвязный слой **Dense** с 50 нейронами и функцией активации **relu**.
9. Выходной полносвязный слой Dense с функцией активации **softmax**.

```
def larger_model():
    # create model
    model = Sequential()
    model.add(Conv2D(30, (5, 5), input_shape=(1, 28, 28), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
    model.add(Conv2D(15, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
  return model
model = larger_model()
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Large CNN Error: %.2f%%" % (100-scores[1]*100))
```

Эта модель уже достигает уровня ошибки классификации 0,89%.

15. ПРЕДСТАВЛЕНИЕ СЛОВ В ВЕКТОРНОМ ПРОСТРАНСТВЕ

15.1. Векторизация слов

Векторизация слов (**world embeding**) - это класс подходов для представления слов и документов с использованием векторного представления. Это улучшение по сравнению с традиционными схемами кодирования, где для представления каждого слова использовались большие разреженные векторы или оценка каждого слова в векторе для представления целого словарного запаса. Эти представления были скудными, потому что словари были обширными, и данное слово или документ представлялось бы большим вектором, состоящим в основном из нулевых значений.

Вместо этого в **world embeding** слова представлены плотными векторами, где вектор представляет проекцию слова в непрерывное векторное пространство.

Представление слова в векторном пространстве получается из текста и основывается на словах, которые окружают слово, когда оно используется.

Два популярных примера методов вложения слов в текст включают:

- Word2Vec.
- GloVe.

В дополнение к этим ранее разработанным методам, векторизацию слов можно изучить как часть модели глубокого обучения.

15.2. Embedding слой Keras

Keras предлагает слой **Embedding**, который можно использовать в моделях нейронных сетей для обработки текстовых данных. Он требует, чтобы входные данные были закодированы целыми числами, так что каждое слово представлено уникальным целым числом. Эта стадия подготовки данных может быть выполнена с использованием **API Tokenizer**, также предоставляемого Keras.

Слой **Embedding** инициализируется случайными весами и производит векторизацию для всех слов в наборе учебных данных.

Это гибкий слой, который можно использовать различными способами, такими как:

1. Его можно использовать отдельно, чтобы изучить векторизацию слов, которое может быть сохранено и использовано в другой модели позже.
2. Он может использоваться как часть модели глубокого обучения, в которой векторизацию изучается вместе с самой моделью.
3. Его можно использовать для загрузки предварительно подготовленной модели векторизации слов, типа передачи обучения.

Слой векторизации **Embedding** определяется как первый скрытый уровень сети. Он имеет три аргумента:

- **input_dim**: Это размер словаря текстовых данных. Например, если целые данные кодируются значениями от 0 до 10, то размер словаря будет составлять 11 слов.

- **output_dim**: Это размерность векторного пространства, в которое будут векторизироваться слова. Он определяет размер выходных векторов этого слоя для каждого слова. Например, это может быть 32 или 100 или даже больше.
- **input_length**: Это длина входных последовательностей, как вы бы определили для любого входного слоя модели Keras. Например, если все входные документы состоят из 1000 слов, это будет 1000.

  Например, ниже мы определяем слой **Embedding** со словарем в 200 слов (например, целочисленные кодированные слова от 0 до 199 включительно), векторное пространство из 32 измерений, в которое будут векторизованы слова, и входные документы, каждая из которых содержит 50 слов.

  e = Embedding(200, 32, input_length=50)

В слое **Embedding** содержатся веса, которые можно впоследствии можно анализировать. Если вы сохраните модель в файле, это будет включать в себя весы для слоя **Embedding**.

Выходом слоя ***Embedding*** является 2D-вектор с одним вектором для каждого слова во входной последовательности слов (входной документ).

Если необходимо подключить полносвязный слой непосредственно к слою **Embedding**, то вы должны сначала сгладить матрицу 2D-вывода на 1D-вектор, используя слой ***Flatten***.

Теперь давайте посмотрим, как мы можем использовать слой **Embedding** на практике.

15.3. Пример обучения векторизации

Мы создадим небольшую задачу, в которой у нас есть 10 текстовых документов, каждый из которых имеет комментарий о части работы, выполненной студентом. Каждый текстовый документ классифицируется как положительный «1» или отрицательный «0». Сначала мы определим документы и их метки классов.

from numpy import array

from keras.preprocessing.text import one_hot

from keras.preprocessing.sequence import pad_sequences

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Flatten

from keras.layers.embeddings import Embedding

docs = ['Well done!', 'Good work', 'Great effort', 'nice work', 'Excellent!',

'Weak', 'Poor effort!', 'not good', 'poor work', 'Could have done better.']

labels = array([1,1,1,1,1,0,0,0,0,0])

Затем мы можем закодировать каждый документ целыми числами. Это означает, что в качестве входных данных слой **Embedding** будет иметь последовательности целых чисел. Keras предоставляет функцию one_hot(), которая создает хэш каждого слова как эффективное целочисленное кодирование. Мы оценим размер словаря 50, что намного больше, чем необходимо для снижения вероятности столкновений совпадений от хэш-функции.

vocab_size = 50

encoded_docs = [one_hot(d, vocab_size) for d in docs]

print(encoded_docs)

Последовательности имеют разную длину, и поэтому мы будем заполнять все входные последовательности до длины 4. Это мы можем сделать это со встроенной функцией Keras pad_sequences().

max_length = 4

padded_docs=pad_sequences(encoded_docs,maxlen=max_length, padding='post')

print(padded_docs)

Теперь мы готовы определить наш слой ***Embedding*** как часть нашей модели нейронной сети. Модель представляет собой простую модель двоичной классификации. Важно отметить, что выход из слоя ***Embedding*** будет 4 вектора по 8 измерений каждый, по одному для каждого слова.

model = Sequential()

model.add(Embedding(vocab_size, 8, input_length=max_length))

model.add(Flatten())

model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])

print(model.summary())

Наконец, мы можем подгонять и оценивать классификационную модель.

model.fit(padded_docs, labels, epochs=150, verbose=0)

loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)

*print('Accuracy: %f' % (accuracy*100))*

Запуск примера сначала печатает целочисленное кодирование документов.

[[42, 38], [24, 27], [26, 11], [3, 27], [36], [42], [28, 11], [8, 24], [28, 27],
 [23, 4, 38, 31]]

Затем печатаются заполненные вектора каждого документа, заполненные нулями чтобы они были одинаковой длины.

[[42 38 0 0]
 [24 27 0 0]
 [26 11 0 0]
 [3 27 0 0]
 [36 0 0 0]
 [42 0 0 0]
 [28 11 0 0]
 [8 24 0 0]
 [28 27 0 0]
 [23 4 38 31]]

После определения сети будет напечатана информация о структуре сети. Как и ожидалось, выход слоя **Embedding** представляет собой матрицу размером 4 × 8, и эти данные сжимаются до 32-элементного вектора слоем **Flatten**.

_____

Layer (type) Output Shape Param #

```
=================================================================
embedding_1 (Embedding)      (None, 4, 8)              400
_________________________________________________________________
flatten_1 (Flatten)          (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 433
Trainable params: 433
Non-trainable params: 0
_________________________________________________________________
```

Наконец, печатается точность подготовленной модели, показывающая, что она отлично изучила набор учебных материалов.

```
Accuracy: 100.000000
```

Вы можете сохранить обученные веса слоя **Embedding** в файл для последующего использования в других моделях.

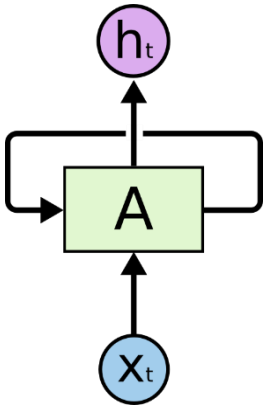
Вы также можете использовать эту модель, чтобы классифицировать другие документы, которые имеют тот же вид словаря, что и в тестовом наборе данных этого примера.

16. LSTM НЕЙРОННЫЕ СЕТИ И ПРОГНОЗИРОВАНИЕ ВРЕМЕННЫХ РЯДОВ

16.1 Рекуррентные нейронные сети

Рекуррентные нейронные сети (англ. Recurrent neural network; RNN) — вид нейронных сетей, где связи между элементами образуют направленную последовательность. Благодаря этому появляется возможность обрабатывать серии событий во времени или последовательные пространственные цепочки.

В отличие от многослойных перцептронов, рекуррентные сети могут использовать свою внутреннюю память для обработки последовательностей произвольной длины. Поэтому сети RNN применимы в таких задачах, где нечто целостное разбито на сегменты, например, распознавание рукописного текста или распознавание речи. Было предложено много различных архитектурных решений для рекуррентных сетей от простых до сложных. В последнее время наибольшее распространение получили сеть с долговременной и кратковременной памятью (LSTM) и управляемый рекуррентный блок (GRU).

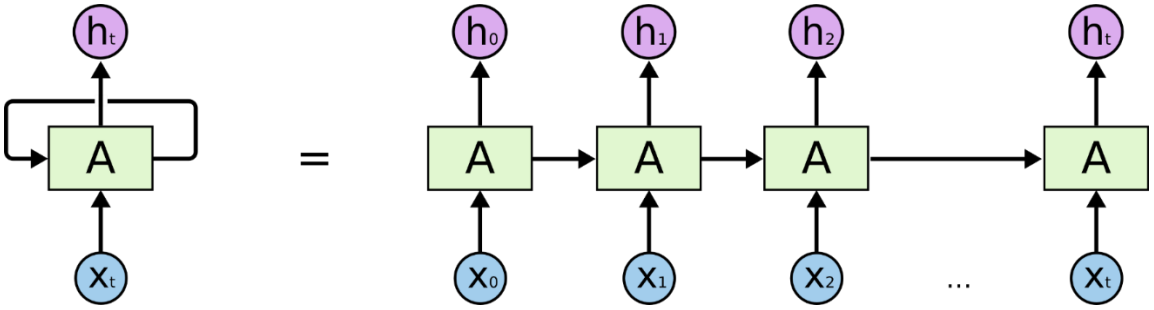


В диаграмме выше участок нейронной сети **A** получает некие данные **X** на вход и подает на выход некоторое значение **H**. Циклическая связь позволяет передавать информацию от текущего шага сети к следующему.

Существует много разновидностей, решений и конструктивных элементов рекуррентных нейронных сетей. Трудность рекуррентной сети заключается в том, что если учитывать каждый шаг времени, то становится необходимым для каждого шага времени создавать свой слой нейронов, что вызывает серьёзные вычислительные сложности. Кроме того, многослойные реализации оказываются вычислительно неустойчивыми, так как в них как правило исчезают или зашкаливают веса. Если ограничить расчёт фиксированным временным окном, то полученные модели не будут отражать долгосрочных трендов. Различные подходы пытаются усовершенствовать модель исторической памяти и механизм запоминания и забывания.

Рекуррентные нейронные сети не так уж сильно отличаются от обычных нейронных сетей. Их можно представить себе, как множество копий одной и

той же сети, причем, каждая копия передает сообщение следующей копии. Посмотрите, что получится, если мы развернем цикл:



Такая “цепная” сущность показывает, что рекуррентные нейронные сети по природе своей тесно связаны с последовательностями и списками.

16.2. Полностью рекуррентная сеть

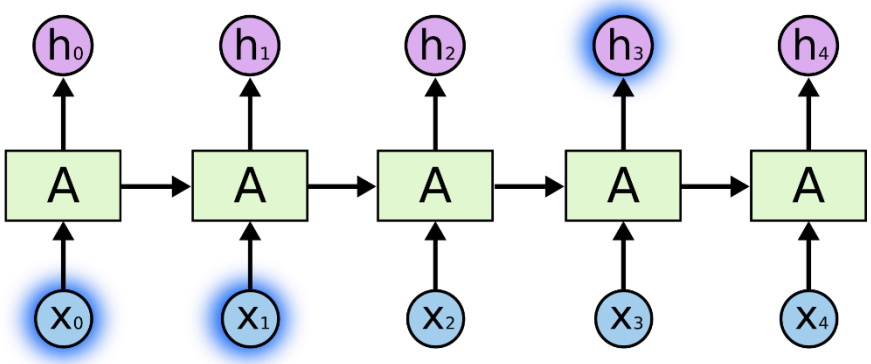
Это базовая архитектура разработана в 1980-х. Сеть строится из узлов, каждый из которых соединён со всеми другими узлами. У каждого нейрона порог активации меняется со временем и является вещественным числом. Каждое соединение имеет переменный вещественный вес. Узлы разделяются на входные, выходные и скрытые.

Для **обучения с учителем** с дискретным временем, каждый (дискретный) шаг времени на входные узлы подаются данные, а прочие узлы завершают свою активацию, и выходные сигналы готовятся для передачи нейроном следующего уровня. Если, например, сеть отвечает за распознавание речи, в результате на выходные узлы поступают уже метки (распознанные слова).

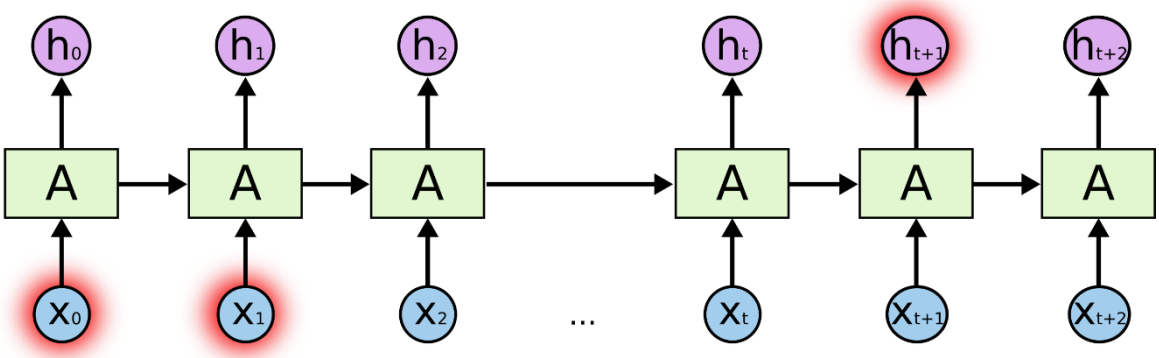
В **обучении с подкреплением (reinforcement learning)** нет учителя, обеспечивающего целевые сигналы для сети, вместо этого иногда используется функция годности[en] или функция оценки (reward function), по которой проводится оценка качества работы сети, при этом значения на выходе оказывает влияние на поведение сети на входе. В частности, если сеть реализует игру, на выходе измеряется количество пунктов выигрыша или оценки позиции. Каждая цепочка вычисляет ошибку как суммарную девиацию по выходным сигналам сети. Если имеется набор образцов обучения, ошибка вычисляется с учётом ошибок каждого отдельного образца.

16.3. Проблема долгосрочных зависимостей

Одна из идей, которая делает РНС столь притягательными, состоит в том, что они могли бы использовать полученную в прошлом информацию для текущих задач. Например, они могли бы использовать предыдущие кадры видео для понимания последующих. Иногда нам достаточно недавней информации, чтобы выполнять текущую задачу. Например, представим модель языка, которая пытается предсказать следующее слово, основываясь на предыдущих. Если мы пытаемся предсказать последнее слово в предложении “Тучи на *небе*”, нам не нужен больше никакой контекст - достаточно очевидно, что в конце предложения речь идёт о небе. В таких случаях, где невелик промежуток между необходимой информацией и местом, где она нужна, РНС могут научиться использовать информацию, полученную ранее.



Но также бывают случаи, когда нам нужен более широкий контекст. Предположим, нужно предсказать последнее слово в тексте “Я вырос во Франции… Я свободно говорю по *французски*”. Недавняя информация подсказывает, что следующее слово, вероятно, название языка, но если мы хотим уточнить, какого именно, нам нужен предыдущий контекст вплоть до информации о Франции. Совсем не редко промежуток между необходимой информацией и местом, где она нужна, становится очень большим. К сожалению, по мере роста промежутка, РНС становятся неспособны научиться соединять информацию.



Теоретически, РНС способны обрабатывать такие долговременные зависимости. Человек может тщательно подобрать их параметры, чтобы решать игрушечные проблемы такой формы. Однако, на практике, РНС не способны выучить такое.

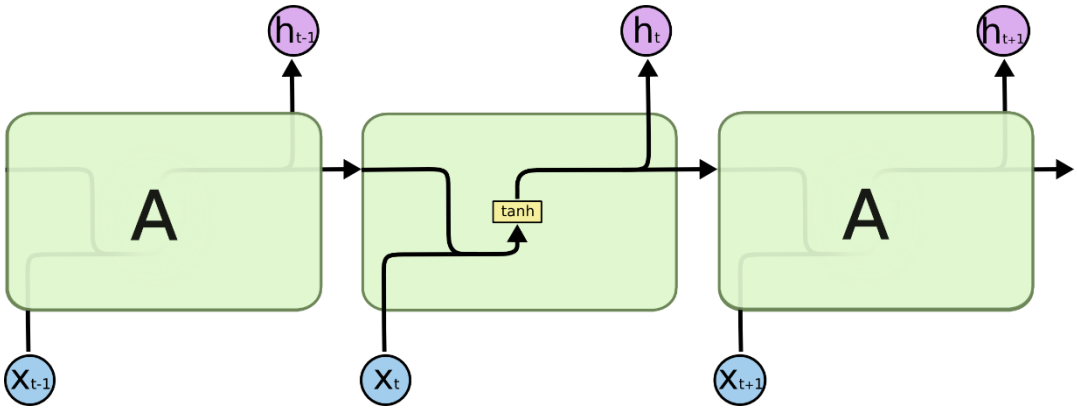
16.4. LSTM сети

Сети долго-краткосрочной памяти (Long Short Term Memory) - обычно просто называют “LSTM” - особый вид РНС, способных к обучению долгосрочным зависимостям. Они работают невероятно хорошо на большом разнообразии проблем и в данный момент широко применяются. LSTM специально спроектированы таким образом, чтобы избежать проблемы долгосрочных зависимостей. Запоминать информацию на длительный период времени - это практически их поведение по-умолчанию, а не что-то такое, что они только пытаются сделать.

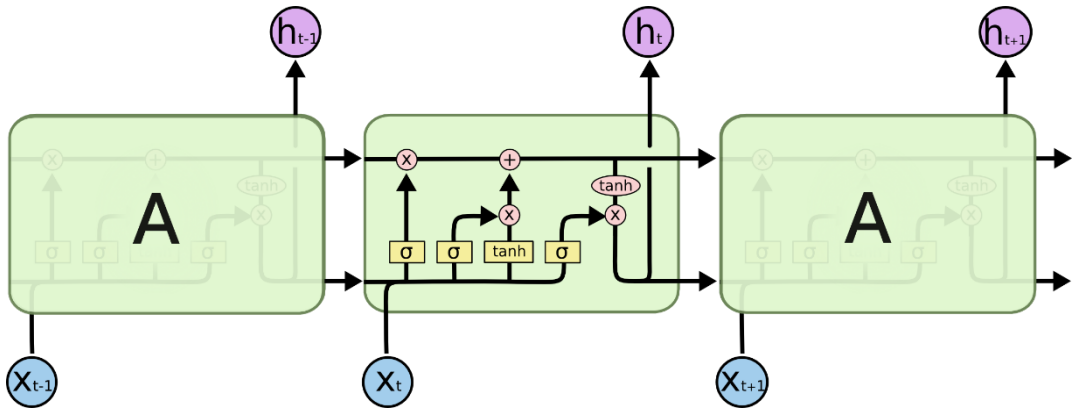
В LSTM сетях удалось обойти проблему исчезновения или зашкаливания градиентов в процессе обучения методом обратного распространения ошибки. Сеть LSTM обычно управляется с помощью рекуррентных вентилей, которые называются вентили (**gates**) «забывания». Ошибки распространяются назад по времени через потенциально неограниченное количество виртуальных слоёв. Таким образом происходит обучение в LSTM, при этом сохраняя память о тысячах и даже миллионах временных интервалов в прошлом. Топологии сетей типа LSTM могут разрабатываться в соответствии со спецификой задачи. В сети LSTM даже большие задержки между значимыми событиями могут

учитываться, и тем самым высокочастотные и низкочастотные компоненты могут смешиваться.

Все рекуррентные нейронные сети имеют форму цепи повторяющих модулей (repeating module) нейронной сети. В стандартной РНС эти повторяющие модули будут иметь очень простую структуру, например, всего один слой гиперболического тангенса (**tanhtanh**).



LSTM тоже имеют такую цепную структуру, но повторяющий модуль имеет другое строение. Вместо одного нейронного слоя их четыре, причем они взаимодействуют особым образом.



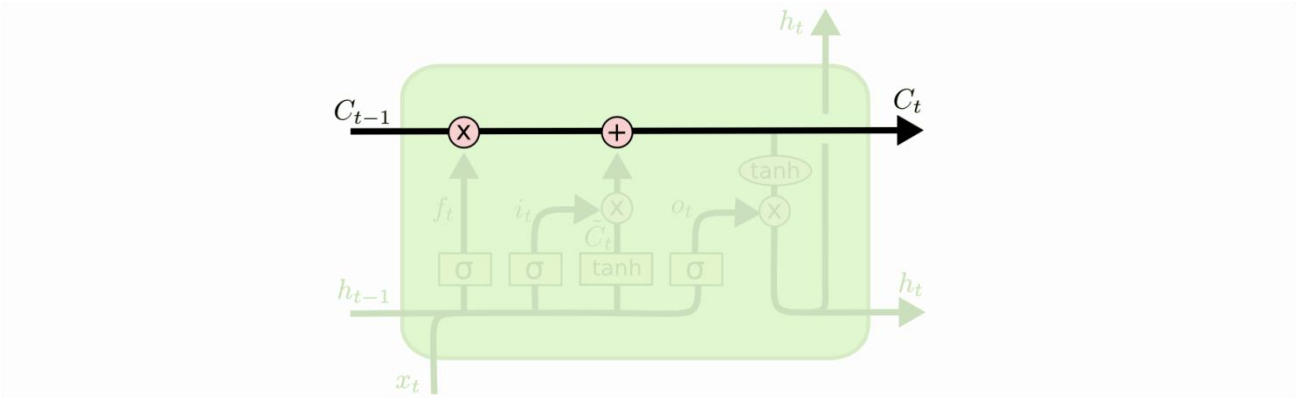
Здесь введены следующие обозначения:

В диаграмме выше каждая линия передает целый вектор от выхода одного узла к входам других. Розовые круги представляют поточечные операторы, такие как сложение векторов, в то время, как желтые

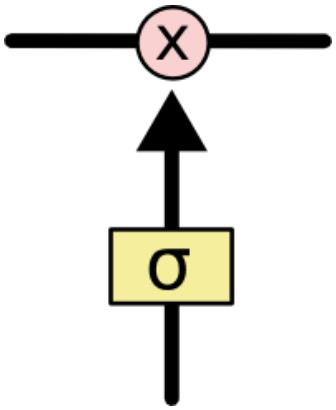
прямоугольники - это обученные слои нейронной сети. Сливающиеся линии обозначают конкатенацию, в то время как ветвящиеся линии обозначают, что их содержимое копируется, и копии отправляются в разные места.

16.5. Главная идея LSTM

Ключ к LSTM - клеточное состояние (cell state) - горизонтальная линия, проходящая сквозь верхнюю часть диаграммы. Клеточное состояние - это что-то типа ленты конвейера. Она движется прямо вдоль всей цепи только лишь с небольшими линейными взаимодействиями. Информация может просто течь по ней без изменений.



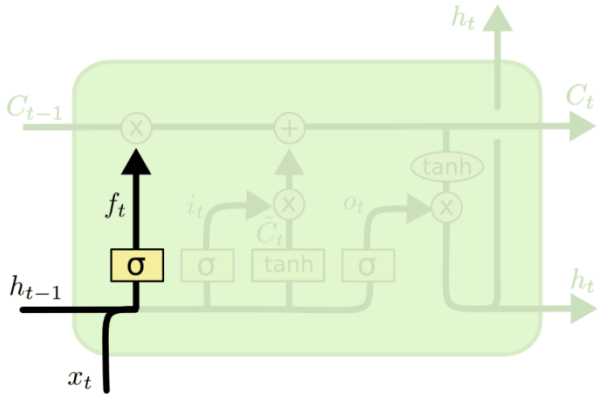
LSTM имеет способность удалять или добавлять информацию к клеточному состоянию, однако эта способность тщательно регулируется структурами, называемыми вентилями (**gates**). Вентили - это способ избирательно пропускать информацию. Они составлены из сигмоидного слоя НС и операции поточечного умножения (pointwise multiplication).



Сигмоидный слой подает на выход числа между нулем и единицей, описывая таким образом, насколько каждый компонент должен быть пропущен сквозь вентиль. Ноль - “ничего не пропускать”, один - “пропускать все”. LSTM имеет три таких вентиля, чтобы защищать и контролировать клеточное состояние.

Первым шагом в нашей LSTM будет решить какую информацию мы собираемся выбросить из клеточного состояния. Это решение принимается сигмоидным слоем, называемым “забывающим вентилем” (“**forget gate layer**”). Он получает на входе значения h_{t-1} и x_t и подает на выход число между 0 и 1. Единица означает “сохрани это полностью”, в тот время как ноль означает “избавься от этого полностью”.

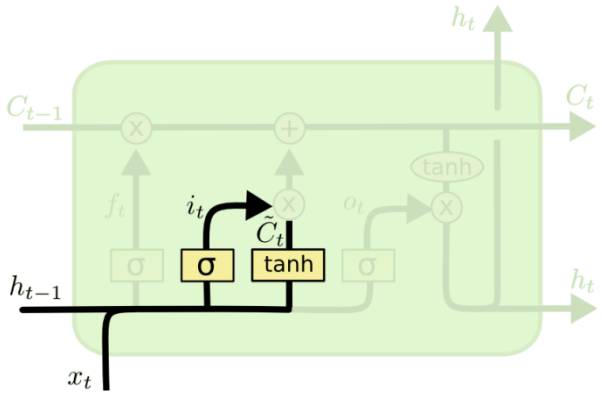
Давайте вернемся к нашему примеру языковой модели, пытающейся предсказать следующее слово, основываясь на всех предыдущих. В такой проблеме клеточное состояние может включать род подлежащего, что позволит использовать правильные формы местоимений. Когда мы видим новое подлежащее, мы забываем род предыдущего подлежащего.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

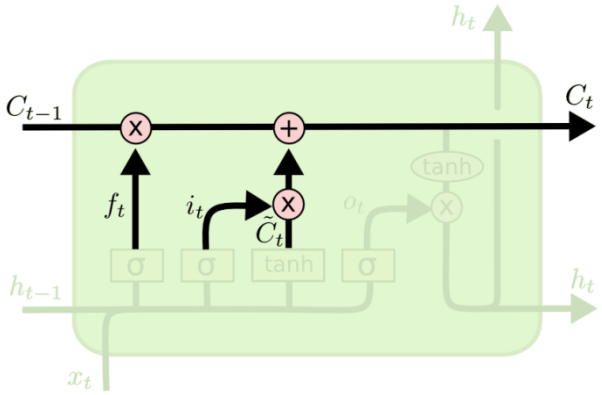
Следующим шагом будет решить, какую новую информацию мы собираемся сохранить в клеточном состоянии. Этот шаг состоит из двух частей. Во-первых, сигмоидный слой, называемый “входным вентилем” (“**input gate layer**”), решает, какие значения мы обновим. Далее, слой гирпеболического тангенса создает вектор кандидатов на новые значения σ_t, который может быть добавлен к состоянию. На следующем шаге мы соединим эти две части, чтобы создать обновление для состояния.

В примере с нашей языковой моделью мы бы хотели добавить род нового подлежащего к клеточному состоянию, чтобы заменить род старого, которое мы должны забыть.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

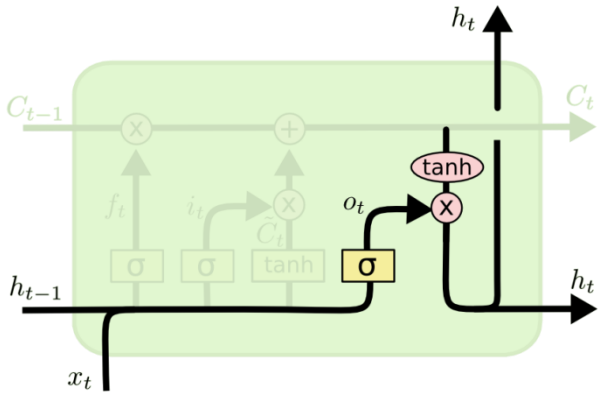
Теперь пришла пора обновить старое клеточное состояние, σ_{t-1} новым клеточным состоянием σ_t. Все решения уже приняты на предыдущих шагах, осталось только сделать это. Мы умножаем старое состояние на f_t, забывая все, что мы ранее решили забыть. В случае с языковой моделью, это как раз то место, где мы теряем информацию о роде старого подлежащего и добавляем новую информацию, как решили на предыдущих шагах.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Наконец, нам нужно решить, какой результат мы собираемся подать на выход. Этот результат будет основан на нашем клеточном состоянии, но будет его отфильтрованной версией. Сначала мы запускаем сигмоидный слой, который решает, какие части клеточного состояния мы собираемся отправить на выход. Затем мы пропускаем клеточное состояние сквозь гиперболический тангенс (**tanhtanh**) (чтобы уместить значения в промежуток от -1до 1) и умножаем его на выход сигмоидного вентиля, так что мы отправляем на выход только те части, которые мы хотим.

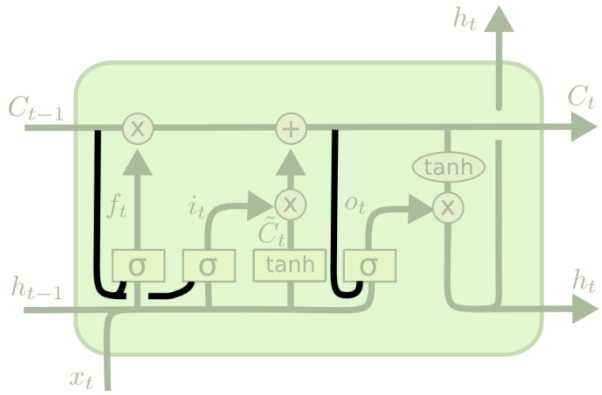
В примере с языковой моделью, если она только что видела подлежащее, она могла бы подать на выход информацию, относящуюся к глаголу (в случае, если следующее слово именно глагол). К примеру, она, возможно, подаст на выход число подлежащего (единственное или множественное). Таким образом, мы будем знать, какая форма глагола должна быть подставлена (если конечно дальше идет именно глагол).



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

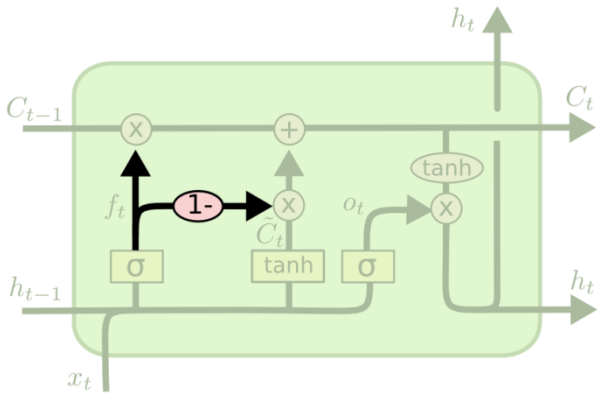
16.6. Разновидности LSTM сетей

В одном из популярных вариантов LSTM добавляются “глазковые соединения” (“peephole connections”). Это значит, что мы позволяем вентилям “подглядывать” за клеточным состоянием.



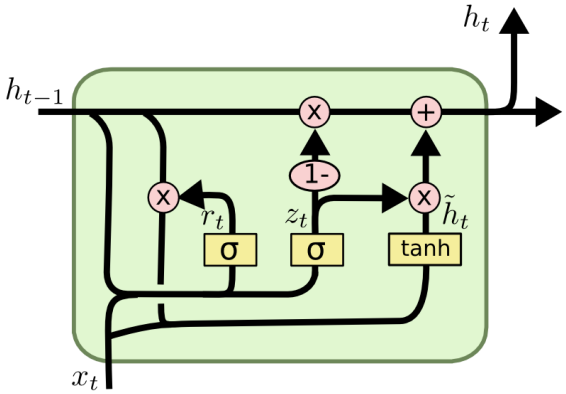
$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t]\ +\ b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t]\ +\ b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t]\ +\ b_o\right)$$

Другая вариация - использование спаренных забывающих и входных вентилей. Вместо того, чтобы независимо решать, что забыть и куда мы должны добавить новую информацию, мы принимаем эти решения одновременно. Мы забываем что-то только в том случае, когда мы получаем что-то другое на это место. Мы получаем на вход новые значения только когда забываем что-то старое.



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

Несколько более существенно отличается от LSTM вентильная рекуррентная единица (Gated Recurrent Unit) или GRU. Она совмещает забывающие и входные вентили в один “обновляющий вентиль” (“**update gate**”). Она также сливает клеточное состояние со скрытым слоем и вносит некоторые другие изменения. Модель, получающаяся в результате, проще, чем обычная модель LSTM и она набирает популярность.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Это только некоторые из наиболее заметных вариантов LSTM. Есть множество других, например, глубинно-вентильные РНС (Depth Gated RNNs). Существует и совершенно другой подход к изучению долговременных зависимостей, например, часовые РНС (Clockwork RNNs)

16.7. Прогнозирование временных рядов

Задачи прогнозирования временных рядов - сложный тип проблемы прогнозирующего моделирования. В отличие от регрессионного предсказательного моделирования временные ряды также добавляют сложность зависимости последовательности от входных переменных.

Мощный тип нейронной сети, предназначенный для обработки последовательностей называется рекуррентными нейронными сетями. Сеть с

длинной короткой памятью или сеть LSTM - это тип рекуррентной нейронной сети, используемой в глубоком обучении, потому что можно успешно обучать очень большие архитектуры.

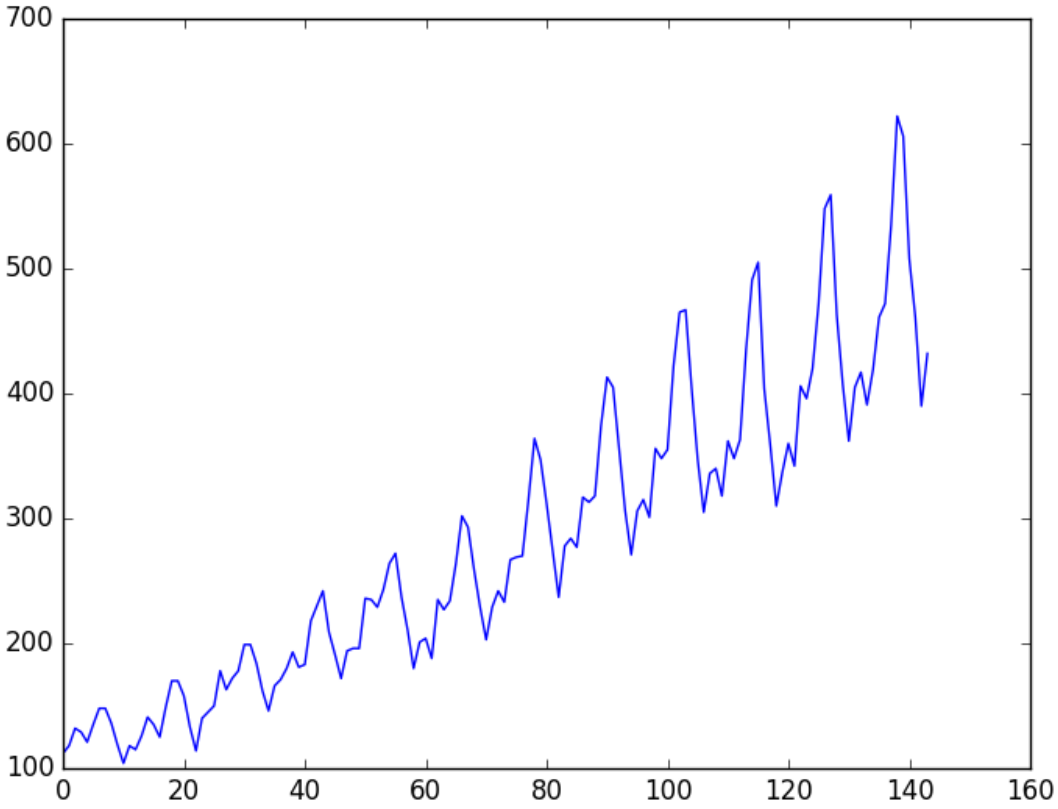
В этом разделе мы разработаем ряд LSTM для стандартной задачи прогнозирования временных рядов. Эти примеры помогут вам разработать свои собственные структурированные LSTM-сети для задач прогнозирования временных рядов.

Задача, которую мы рассмотрим это - проблема прогнозирования пассажирских авиаперевозок. Задача состоит в том зная год и месяц предсказать количество пассажиров международных авиакомпаний.

Набор данных доступен бесплатно можно скачать с адреса https://datamarket.com/data/set/22u3/international-airline-passengers-monthly-totals-in-thousands-jan-49-dec-60#!ds=22u3&display=line с именем файла « *international-airlines-passengers.csv* ». Данные варьируются от января 1949 года до декабря 1960 года или 12 лет с 144 наблюдениями. Мы можем загрузить этот набор данных с помощью библиотеки Pandas. Нам не интересна дата, учитывая, что каждое наблюдение разделяется одним и тем же интервалом в один месяц. Поэтому, когда мы загружаем набор данных, мы можем исключить первый столбец. После загрузки мы можем легко построить весь набор данных. Код для загрузки и построения набора данных приведен ниже.

```
import pandas
import matplotlib.pyplot as plt
dataset = pandas.read_csv('international-airline-passengers.csv', usecols=[1], engine='python', skipfooter=3)
plt.plot(dataset)
plt.show()
```

С течением времени можно увидеть восходящий тренд в наборе данных и некоторую периодичность для набора данных, который, вероятно, соответствует периоду отпуска в северном полушарии.



Мы можем сформулировать эту задачу как задачу регрессии. То есть, учитывая количество пассажиров (в тысячах единиц) в этом месяце прогнозировать количество пассажиров в следующем месяце. Мы можем написать простую функцию, чтобы преобразовать наш единственный столбец данных в двухстолбцовый набор данных: первая колонка, содержащая количество пассажиров и второй столбец, который будет содержать количество пассажиров в следующем месяце.

Прежде чем мы начнем, давайте сначала импортируем все функции и классы, которые мы намерены использовать.

```
import numpy
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
```

from keras.layers import LSTM

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean_squared_error

Прежде чем мы что-либо сделать инициализируем генератор случайных чисел, чтобы гарантировать, что наши результаты будут воспроизводимыми.

numpy.random.seed(7)

Используем код из предыдущего раздела для загрузки набора данных в виде данных Pandas. Затем мы можем извлечь массив NumPy из фрейма данных и преобразовать целочисленные значения в значения с плавающей запятой, которые более подходят для работы с нейронной сетью.

dataframe = read_csv('international-airline-passengers.csv',usecols=[1], engine='python', skipfooter=3)

dataset = dataframe.values

dataset = dataset.astype('float32')

LSTM чувствительны к шкале входных данных, особенно когда используются сигмоидные (по умолчанию) или функции активации **tanh**. Поэтому необходимо произвести масштабирование данных до диапазона от 0 до 1, также называемого нормализацией. Мы можем легко нормализовать набор данных, используя **класс** предварительной обработки **MinMaxScaler** из библиотеки **scikit-learn**.

scaler = MinMaxScaler(feature_range=(0, 1))

dataset = scaler.fit_transform(dataset)

После того, как мы моделируем наши данные и оценим качество нашей модели на учебном наборе данных, нам нужно узнать насколько точен прогноз на данных которые сеть не видела. Для обычной задачи классификации или регрессии мы будем делать это с использованием перекрестной проверки. При использовании временных рядов важна последовательность значений. Простым методом, который мы можем использовать, является разделение упорядоченного набора данных на учебный и тестовые наборы данных. Приведенный ниже код разделяет данные на учебные наборы данных с

67% наблюдений, которые мы можем использовать для обучения нашей модели, оставляя 33% для тестирования модели.

```
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
```

Теперь определим функцию для создания нового набора данных, как описано выше. Функция принимает два аргумента: **набор данных**, который представляет собой массив **NumPy**, который мы хотим преобразовать в набор данных, и **look_back**, который представляет собой число предыдущих шагов времени для использования в качестве входных переменных для прогнозирования следующего периода времени - в этом случае по умолчанию - 1. Это значение по умолчанию создаст набор данных, где X - количество пассажиров в заданное время (t), а Y - количество пассажиров в следующий момент времени (t + 1).

```
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return numpy.array(dataX), numpy.array(dataY)
```

Далее используем эту функцию для подготовки наборов данных для обучения и для тестирования нейронной сети и преобразуем данные в структуру, соответствующую входу нейронной сети.

```
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

Теперь мы разработаем и настроим нашу сеть LSTM для решения этой задачи. В слой с 4 блоками LSTM или нейронами и выходной уровень, который на выходе дает одно значение. Для LSTM нейронов используется сигмоидальная функция активации по умолчанию и сеть обучается в течение 100 эпох.

```
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
```

После того, как модель прошла обучение мы можем оценить качество модели учебном и на тестовом наборах данных. Обратите внимание, что мы инвертируем предсказания перед вычислением ошибок, чтобы гарантировать, что результат выводится в тех же единицах, что и исходные данные (тысячи пассажиров в месяц).

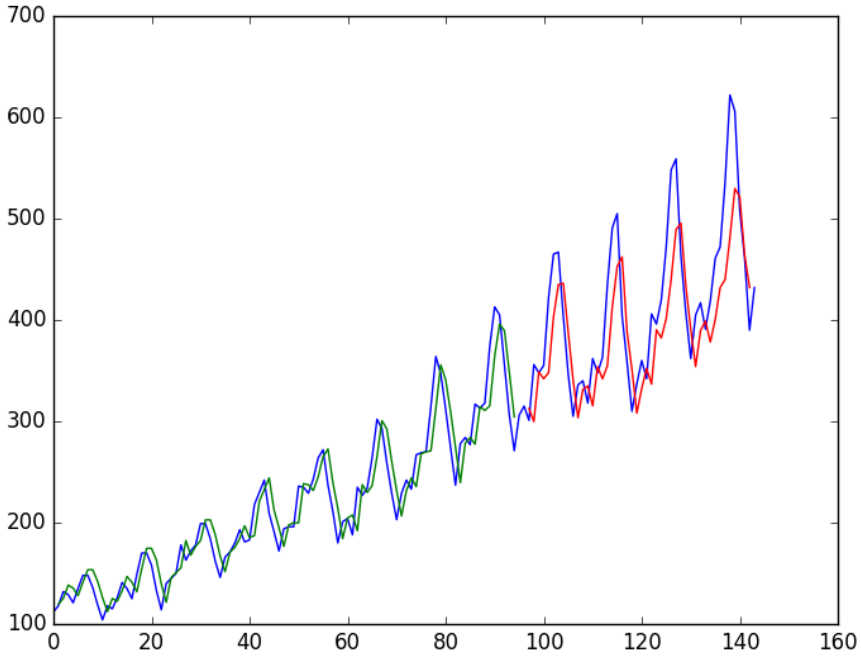
```
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
```

Наконец, мы можем генерировать предсказания, используя учебные и тестовые данные, для того чтобы получить визуальное представление о качестве модели.

Из-за того, как был подготовлен набор данных, мы должны сдвинуть предсказания так, чтобы они выровнялись по оси **x** с исходным набором данных.

```
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Исходный набор данных отображен синим цветом, прогнозы для набора учебных данных зеленым цветом и прогнозы по тестовому набору данных красным цветом. Мы видим, что модель отлично справилась с прогнозом как учебном, так и на тестовом наборе данных.



В консоли программа выдала следующую информацию об ошибках:

Train Score: 22.93 RMSE

Test Score: 47.53 RMSE

17. КЛАССИФИКАЦИЯ ПОСЛЕДОВАТЕЛЬНОСТЕЙ С ИСПОЛЬЗОВАНИЕМ LSTM НЕЙРОННЫХ СЕТЕЙ

Классификация последовательностей - это проблема прогнозирующего моделирования, в которой у вас есть некоторая последовательность входов по пространству или времени, и задача заключается в прогнозировании категории для последовательности. Сложность этой проблемы заключается в том, что последовательности могут варьироваться по длине, состоять из очень большого словарного запаса входных символов и могут потребовать от модели изучения долгосрочного контекста или зависимостей между символами во входной последовательности. В этом разделе мы разработаем LSTM рекуррентную модели нейронной сети для задач классификации последовательностей.

Задача, которую мы будем решать - это задача классификации тональности отзыва фильма IMDB. Каждый отзыв представляет собой переменную последовательность слов, и тональность каждого отзыва фильма должна быть классифицирована. Набор данных содержит 25 000 высокополярных отзывов фильмов (хорошие или плохие) для обучения и того же количества для тестирования. Keras содержит функция **imdb.load_data (), которая** позволяет загружать набор данных в формате, который готов для использования в нейронной сети. Слова были заменены целыми числами, которые указывают упорядоченную частоту каждого слова в наборе данных. Поэтому предложения в каждом обзоре состоят из последовательности целых чисел.

Будем отображать каждое слово на 32-значный вещественный вектор. Мы также ограничим общее количество слов, которые нас интересуют в моделировании, до 5000 наиболее часто встречающихся слов. И так как длина последовательности (количество слов) в каждом обзоре меняется мы будем

ограничивать каждый обзор 500 словами, усекая длинные обзоры и заполняя более короткие обзоры нулевыми значениями.

Начнем как обычно с импорта классов и функций, необходимых для этой модели, и инициализации генератора случайных чисел, чтобы мы могли легко воспроизвести результаты.

```
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
numpy.random.seed(7)
```

Далее нам нужно загрузить набор данных IMDB. Мы ограничиваем набор данных до 5000 слов. Мы также разделили набор данных на поезд (50%) и тест (50%).

```
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
```

Затем нам нужно усечь или дополнить входные последовательности так, чтобы они были одинаковой длины для обучения нейронной сети.

```
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
```

Теперь мы можем создать, скомпилировать и обучить нашу модель LSTM.

Первый слой - это **Embedded** слой, который использует 32 вектора для представления каждого слова. Следующий слой - это слой LSTM содержащий 100 нейронов. Наконец, поскольку это проблема классификации, мы используем полносвязный **Dense** выходной слой с одним нейроном и функцией активации сигмоида, чтобы получить на выходе 0 или 1 для предсказания двух классов (хороших и плохих).

Поскольку это проблема двоичной классификации, используется логарифмическая потеря как функция ошибок (**binary_crossentropy**). Используется эффективный алгоритм оптимизации **ADAM**. Модель подходит обучение за 3 эпохи, потому что она быстро переобучается.

```
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=
max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, epochs=3, batch_size=64)
```

После обучения сети мы оцениваем качество модели по тестовым данным.

```
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Выполнение этого примера приводит к следующему результату.

```
Epoch 1/3 16750/16750 [==========] - 107s - loss: 0.5570 - acc: 0.7149
Epoch 2/3 16750/16750 [==========] - 107s - loss: 0.3530 - acc: 0.8577
Epoch 3/3 16750/16750 [==========] - 107s - loss: 0.2559 - acc: 0.9019
Accuracy: 86.79%
```

18. НЕЙРОННЫЕ СЕТИ НА ОСНОВЕ БИБЛИОТЕКИ TENSORFLOW

18.1. Начало работы с TensorFlow

TensorFlow - это библиотека программного обеспечения с открытым исходным кодом, созданная Google, которая используется для внедрения систем машинного обучения и глубокого обучения. Эти два имени содержат ряд мощных алгоритмов, которые разделяют общую задачу - позволить компьютеру узнать, как автоматически определять сложные шаблоны и / или

принимать наилучшие возможные решения. TensorFlow, в основе своей, является библиотекой для программирования потока данных. Он использует различные методы оптимизации, чтобы сделать вычисления математических выражений проще и эффективнее.

Некоторые из ключевых особенностей TensorFlow:

- Эффективно работает с математическими выражениями, включающими многомерные массивы
- Хорошая поддержка глубоких нейронных сетей и концепций машинного обучения
- Использование GPU / CPU, где один и тот же код может быть выполнен на обеих архитектурах
- Высокая масштабируемость вычислений на машинах и огромные массивы данных

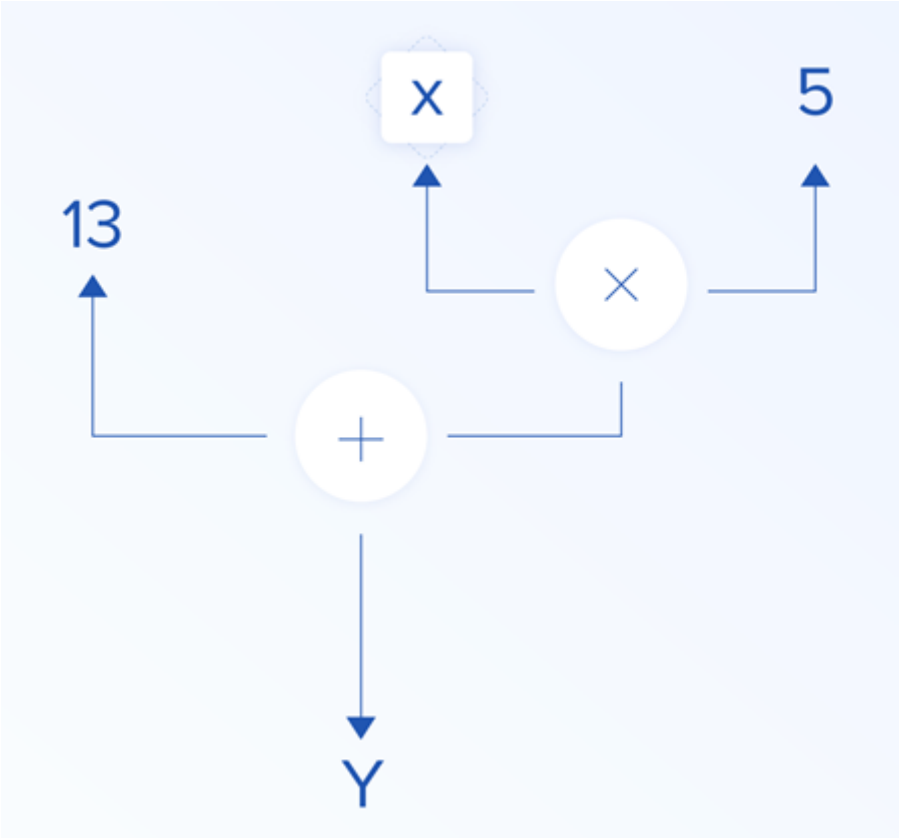
Установить TensorFlow можно выполнив команду:

```
pip install tensorflow
```

Загрузка и сборка TensorFlow может занять несколько минут.

18.2. Основы работы в TensorFlow

В TensorFlow вычисление описывается с использованием графов потоков данных. Каждый узел графа представляет собой экземпляр математической операции (например, сложение, деление или умножение), и каждое ребро представляет собой многомерный набор данных (тензор), на котором выполняются операции.



Прежде чем перейти к обсуждению элементов TensorFlow, мы сначала создадим сеанс работы с TensorFlow, чтобы понять, как выглядит программа TensorFlow.

В TensorFlow **константы** создаются с использованием функции:

constant(value, dtype=None, shape=None, name='Const', verify_shape=False),

где **value** постоянное значение, которое будет использоваться при дальнейших вычислениях, **dtype** является параметром, указывающим тип данных (например, float32/64, int8/16), shape является необязательным параметром, указывающим размер массива данных, **name** является необязательным задающим имя для тензора. Если вам нужны константы с определенными значениями внутри вашей обучающей модели, тогда объект типа **constant** может использоваться как в следующем примере:

z = tf.constant(5.2, name="x", dtype=tf.float32)

Переменные в TensorFlow являются буферами в памяти, содержащими тензоры, которые должны быть явно инициализированы. Просто вызывая конструктор, мы добавляем переменную в вычислительный граф. Переменные

особенно полезны, как только вы начинаете с моделей обучения, и они используются для хранения и обновления параметров. Начальное значение, переданное в качестве аргумента конструктора, представляет собой тензор или объект, который может быть преобразован или возвращен как тензор. Это означает, что если мы хотим заполнить переменную некоторыми предопределенными или случайными значениями, которые будут использоваться впоследствии в процессе обучения и обновлены при итерациях, мы можем определить ее следующим образом:

k = tf.Variable(tf.zeros([1]), name="k")

Другой способ использования переменных в TensorFlow - это вычисления, когда эта переменная не является обучаемой и может быть определена следующим образом:

k = tf.Variable(tf.add(a, b), trainable= False)

Сеанс TensorFlow инкапсулирует управление и состояние среды выполнения TensorFlow. Сеанс без параметров будет использовать граф по умолчанию, созданный в текущем сеансе, иначе класс сеанса принимает параметр графа, который используется в этом сеансе для выполнения. Ниже приведен краткий фрагмент кода, в котором показано, как термины, определенные выше, могут использоваться в TensorFlow для вычисления простой линейной функции **y=a*x+b**:

```
import tensorflow as tf
x = tf.constant(-2.0, name="x", dtype=tf.float32)
a = tf.constant(5.0, name="a", dtype=tf.float32)
b = tf.constant(13.0, name="b", dtype=tf.float32)
y = tf.Variable(tf.add(tf.multiply(a, x), b))
init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    print(session.run(y))
```

18.3. Определение вычислительных графов в TensorFlow

Преимущество при работе с графами потоков данных заключается в том, что модель отделена от ее исполнения, т.е. неважно на каком вычислительном устройстве выполняется программный код, на процессоре, графическом процессоре или некоторой комбинации. Программа в среде TensorFlow может выполняться на процессоре или графическом процессоре, а вся сложность, связанная с кодом исполнения скрыта от пользователя. Граф вычисления - это встроенный процесс, который использует библиотеку, не требуя прямого вызова объекта графа. Граф вычислений может быть построен в процессе использования библиотеки TensorFlow без необходимости явно создавать объекты **Graph**.

Объект **Graph** в TensorFlow может быть создан в результате простой строки кода **c = tf.add(a, b)**. Это код создаст операционный узел, который принимает два тензора **a** и **b** которые вычисляют их сумму **c** в качестве результата.

Плейсхолдер - это способ, позволяющий разработчикам вводить данные в график вычислений через заполнители, которые связаны внутри некоторых выражений. Сигнатура плейсхолдера:

```
placeholder(dtype, shape= None , name= None )
```

где **dtype** - тип элементов в тензорах.

Преимущество плейсхолдеров заключается в том, что они позволяют разработчикам создавать операции в вычислительном графе вообще, без необходимости заранее предоставлять данные, и данные могут быть добавлены во время выполнения из внешних источников.

Рассмотрим простую задачу умножения двух целых чисел **x** и **y** в TensorFlow и использованием плейсхолдера:

```
import tensorflow as tf
x = tf.placeholder(tf.float32, name="x")
y = tf.placeholder(tf.float32, name="y")
```

```
z = tf.multiply(x, y, name="z")
with tf.Session() as session:
    print(session.run(z, feed_dict={x: 2.1, y: 3.0}))
```

18.4. Визуализация вычислительного графа с помощью TensorBoard

TensorBoard - это инструмент визуализации для анализа графиков потока данных. Он может быть полезно для лучшего понимания моделей машинного обучения. С TensorBoard вы можете получить представление о различных типах статистики о параметрах и подробностях о частях вычислительного графа. Глубокая нейронная сеть имеет большое количество узлов. TensorBoard позволяет разработчикам получить представление о каждом узле и о том, как вычисление выполняется во время выполнения TensorFlow.

Теперь давайте вернемся к нашему примеру где мы определили линейную функцию **y = a*x + b.**

Чтобы регистрировать события с сеанса, которые позже могут использоваться в TensorBoard, TensorFlow предоставляет класс **FileWriter**. Его можно использовать для создания файла событий для хранения сводок и событий. Конструктор **FileWriter** принимает шесть параметров и выглядит так:

__init__(logdir, graph=None, max_queue=10, flush_secs=120, graph_def=None, filename_suffix=None) - где требуется указать обязательный параметр **logdir**, а другие - значения по умолчанию. Параметр графа будет передан из объекта сеанса, созданного в программе. Полный код примера выглядит так:

```
import tensorflow as tf
x = tf.constant(-2.0, name="x", dtype=tf.float32)
a = tf.constant(5.0, name="a", dtype=tf.float32)
b = tf.constant(13.0, name="b", dtype=tf.float32)
y = tf.Variable(tf.add(tf.multiply(a, x), b))
init = tf.global_variables_initializer()
```

with tf.Session() as session:

merged = tf.summary.merge_all()

writer = tf.summary.FileWriter("logs", session.graph)

session.run(init)

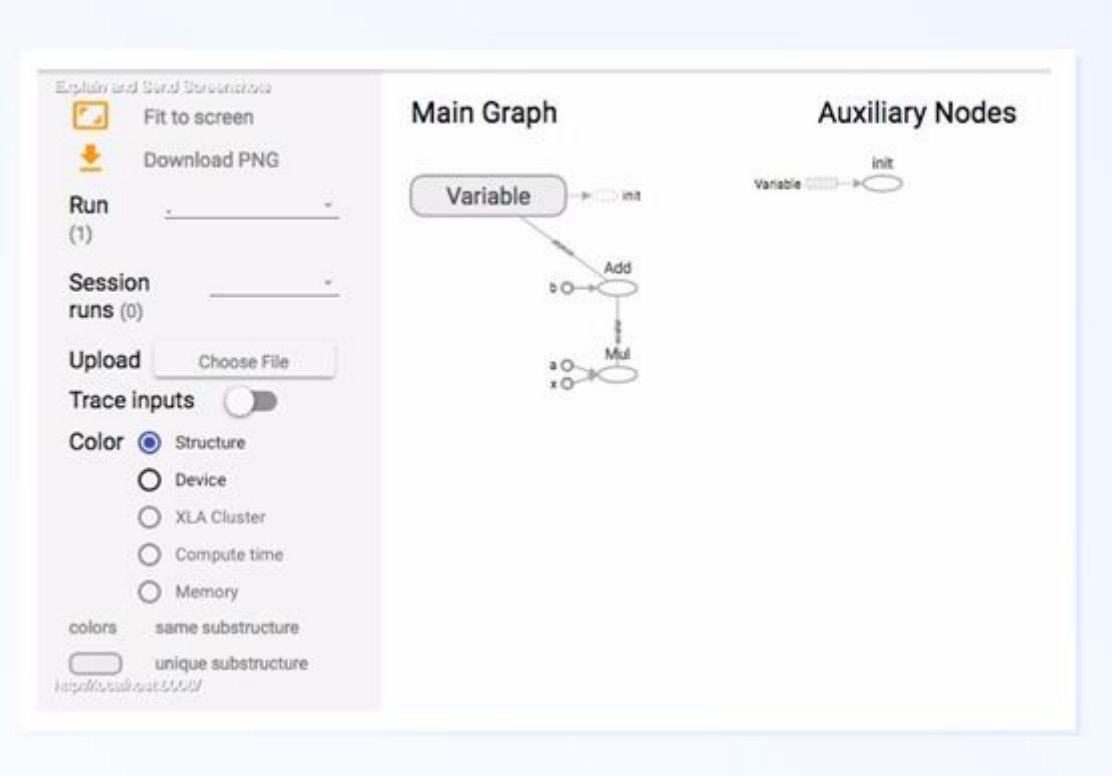
print(session.run(y))

Мы добавили две новые строки в которых создается и используется объект **FileWriter** для вывода событий в файл, как описано выше.

После запуска программы у нас появляется файл в журналах каталогов, и для того чтобы посмотреть содержимое этого файла необходимо запустить tensorboard:

tensorboard --logdir logs/

После открытия http://localhost:6006 и нажатия на пункт меню «Графики» (расположенный в верхней части страницы) вы сможете увидеть график, как на картинке ниже:



TensorBoard маркирует константы и сводные узлы конкретными символами, которые показаны ниже.

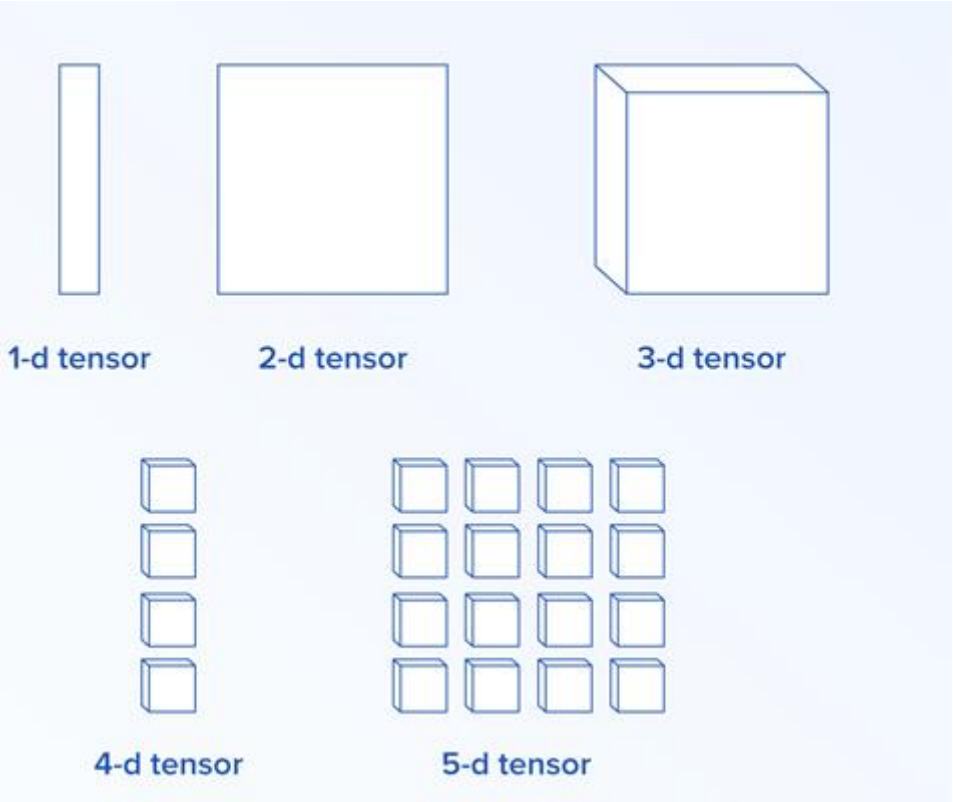
18.5. Математика с TensorFlow

Тензоры - это основные структуры данных в TensorFlow, и они представляют собой соединительные грани в графе потока данных.

Тензор просто идентифицирует многомерный массив или список. Тензорную структуру можно идентифицировать тремя параметрами: рангом, размером и типом.

- Ранг: Определяет количество измерений тензора. Ранг известен как порядок или n-размерности тензора, где, например, тензор ранга 1 является тензором вектора или ранга 2, является матрицей.
- Размер: Размер тензора - это количество строк и столбцов.
- Тип: Тип данных элементов тензора.

Чтобы построить тензор в TensorFlow, мы можем построить n-мерный массив. Это можно сделать легко, используя библиотеку **NumPy**, или путем преобразования n-мерного массива **Python** в тензор TensorFlow.



Чтобы построить 1-мерный тензор, мы будем использовать массив **NumPy**:

import numpy as np

tensor_1d = np.array([1.45, -1, 0.2, 102.1])

Работа с подобным массивом аналогична работе со встроенным списком **Python**. Основное отличие состоит в том, что массив **NumPy** также содержит некоторые дополнительные свойства, такие как размер, форма и тип.

>> print tensor_1d

[1.45 -1. 0.2 102.1]

>> print tensor_1d[0]

1.45

>> print tensor_1d[2]

0.2

>> print tensor_1d.ndim

1

>> print tensor_1d.shape

(4,)

>> print tensor_1d.dtype

float64

Массив NumPy можно легко преобразовать в тензор TensorFlow используя вспомогательную функцию **convert_to_tensor**, что помогает разработчикам преобразовывать объекты Python в объекты тензора. Эта функция принимает тензорные объекты, массивы **NumPy** и списки Python.

tensor = tf.convert_to_tensor(tensor_1d, dtype=tf.float64)

Теперь, если мы привяжем наш тензор к сеансу TensorFlow, мы сможем увидеть результаты нашего преобразования.

i*mport numpy as np*

import tensorflow as tf

tensor_1d = np.array([1.45, -1, 0.2, 102.1])

tensor = tf.convert_to_tensor(tensor_1d, dtype=tf.float64)

with tf.Session() as session:

print(session.run(tensor))

print(session.run(tensor[0]))

print(session.run(tensor[1]))

Вывод:

[1.45 -1. 0.2 102.1]

1.45

-1.0

Мы можем создать 2-й тензор или матрицу аналогичным образом:

```
tensor_2d = np.array(np.random.rand(4, 4), dtype='float32')
tensor_2d_1 = np.array(np.random.rand(4, 4), dtype='float32')
tensor_2d_2 = np.array(np.random.rand(4, 4), dtype='float32')

m1 = tf.convert_to_tensor(tensor_2d)
m2 = tf.convert_to_tensor(tensor_2d_1)
m3 = tf.convert_to_tensor(tensor_2d_2)
mat_product = tf.matmul(m1, m2)
mat_sum = tf.add(m2, m3)
mat_det = tf.matrix_determinant(m3)

with tf.Session() as session:
    print session.run(mat_product)
    print session.run(mat_sum)
    print session.run(mat_det)
```

18.6. Тензорные операции

В приведенном выше примере мы вводим несколько операций TensorFlow для векторов и матриц. Операции выполняют определенные вычисления на тензорах. Функции **TensorFlow** приведены в таблице ниже.

Оператор TensorFlow	Описание
tf.add	x + y
tf.subtract	xy
tf.multiply	x * y

Оператор TensorFlow	Описание
tf.div	x / y
tf.mod	x% y
tf.abs	| X |
tf.negative	-X
tf.sign	sign (x)
tf.square	x * x
tf.round	round(x)
tf.sqrt	SQRT (x)
tf.pow	x ^ y
tf.exp	e ^ x
tf.log	log (x)
tf.maximum	max (x, y)
tf.minimum	min (x, y)
tf.cos	cos (x)
tf.sin	sin (x)

Операции **TensorFlow**, перечисленные в таблице выше, работают с тензорными объектами и выполняются поэлементно. Поэтому, если вы хотите вычислить косинус для вектора **x**, операция TensorFlow будет выполнять вычисления для каждого элемента в переданном тензоре:

tensor_1d = np.array([0, 0, 0])

```
tensor = tf.convert_to_tensor(tensor_1d, dtype=tf.float64)

with tf.Session() as session:

    print session.run(tf.cos(tensor))
```

Вывод:

```
[ 1.  1.  1.]
```

18.7. Матричные операции

Матричные операции очень важны для моделей машинного обучения, таких как линейная регрессия, поскольку они часто используются в них. TensorFlow поддерживает все наиболее распространенные операции с матрицами, такие как умножение, инверсия, вычисление определителя, решение линейных уравнений и многое другое. Давайте напишем некоторый код, который будет выполнять базовые матричные операции, такие как умножение, транспонирование, вычислении определителя, умножения и другие.

Ниже приведены основные примеры вызова этих операций.

```
import tensorflow as tf
import numpy as np


def convert(v, t=tf.float32):
        return tf.convert_to_tensor(v, dtype=t)
m1 = convert(np.array(np.random.rand(4, 4), dtype='float32'))
m2 = convert(np.array(np.random.rand(4, 4), dtype='float32'))
m3 = convert(np.array(np.random.rand(4, 4), dtype='float32'))
m4 = convert(np.array(np.random.rand(4, 4), dtype='float32'))
m5 = convert(np.array(np.random.rand(4, 4), dtype='float32'))
m_tranpose = tf.transpose(m1)
```

```
m_mul = tf.matmul(m1, m2)
m_det = tf.matrix_determinant(m3)
m_inv = tf.matrix_inverse(m4)
m_solve = tf.matrix_solve(m5, [[1], [1], [1], [1]])

with tf.Session() as session:
    print(session.run(m_tranpose))
    print(session.run(m_mul))
    print(session.run(m_inv))
    print(session.run(m_det))
    print(session.run(m_solve))
```

TensorFlow поддерживает различные виды редукции. **Редукция** - это операция, которая удаляет один или несколько измерений из тензора, выполняя определенные операции по этим измерениям. Мы представим несколько из них в приведенном ниже примере.

```
import tensorflow as tf

import numpy as np

def convert(v, t=tf.float32):

   return tf.convert_to_tensor(v, dtype=t)

x = convert(

   np.array(   [ (1, 2, 3),  (4, 5, 6),  (7, 8, 9)   ]), tf.int32)

bool_tensor = convert([(True, False, True), (False, False, True), (True, False, False)], tf.bool)

red_sum_0 = tf.reduce_sum(x)
```

```
red_sum = tf.reduce_sum(x, axis=1)

red_prod_0 = tf.reduce_prod(x)

red_prod = tf.reduce_prod(x, axis=1)

red_min_0 = tf.reduce_min(x)

red_min = tf.reduce_min(x, axis=1)

red_max_0 = tf.reduce_max(x)

red_max = tf.reduce_max(x, axis=1)

red_mean_0 = tf.reduce_mean(x)

red_mean = tf.reduce_mean(x, axis=1)

red_bool_all_0 = tf.reduce_all(bool_tensor)

red_bool_all = tf.reduce_all(bool_tensor, axis=1)

red_bool_any_0 = tf.reduce_any(bool_tensor)

red_bool_any = tf.reduce_any(bool_tensor, axis=1)

with tf.Session() as session:

    print("Reduce sum without passed axis parameter: ", session.run(red_sum_0))

    print("Reduce sum with passed axis=1: ", session.run(red_sum))

    print("Reduce product without passed axis parameter: ", session.run(red_prod_0))

    print("Reduce product with passed axis=1: ", session.run(red_prod))

    print("Reduce min without passed axis parameter: ", session.run(red_min_0))

    print("Reduce min with passed axis=1: ", session.run(red_min))
```

print("Reduce max without passed axis parameter: ", session.run(red_max_0))

print("Reduce max with passed axis=1: ", session.run(red_max))

print("Reduce mean without passed axis parameter: ", session.run(red_mean_0))

print("Reduce mean with passed axis=1: ", session.run(red_mean))

print("Reduce bool all without passed axis parameter: ", session.run(red_bool_all_0))

print("Reduce bool all with passed axis=1: ", session.run(red_bool_all))

print("Reduce bool any without passed axis parameter: ", session.run(red_bool_any_0))

print("Reduce bool any with passed axis=1: ", session.run(red_bool_any))

Вывод программы:

Reduce sum without passed axis parameter: 45

Reduce sum with passed axis=1: [6 15 24]

Reduce product without passed axis parameter: 362880

Reduce product with passed axis=1: [6 120 504]

Reduce min without passed axis parameter: 1

Reduce min with passed axis=1: [1 4 7]

Reduce max without passed axis parameter: 9

Reduce max with passed axis=1: [3 6 9]

Reduce mean without passed axis parameter: 5

Reduce mean with passed axis=1: [2 5 8]

Reduce bool all without passed axis parameter: False

Reduce bool all with passed axis=1: [False False False]

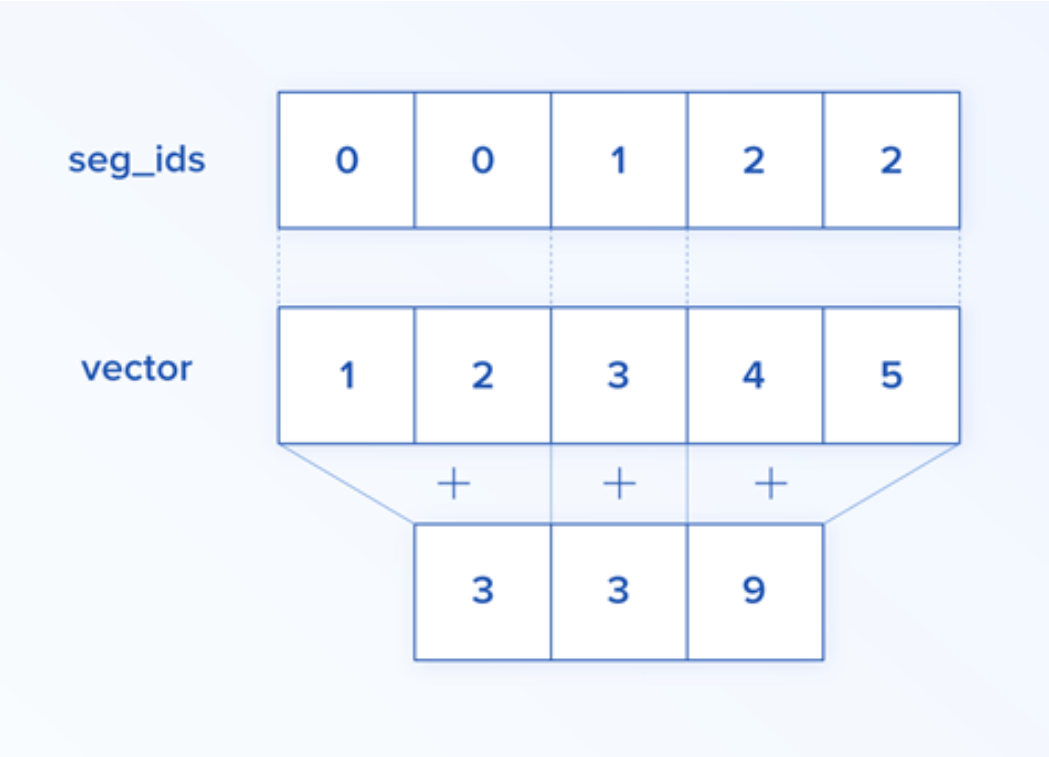
Reduce bool any without passed axis parameter: True

Reduce bool any with passed axis=1: [True True True]

Первым параметром операторов редукции является тензор, который мы хотим уменьшить. Второй параметр - это индексы размерностей, по которым мы хотим выполнить редукцию. Этот параметр является необязательным, и, если его не передать, редукция будет выполняться по всем измерениям.

Сегментация - это процесс, в котором одним из измерений является процесс отображения размеров на предоставленные сегментные индексы, а результирующие элементы определяются строкой индекса.

Сегментация группирует элементы под повторными индексами, поэтому, например, в нашем случае мы имеем сегментированные ids, [0, 0, 1, 2, 2] применяемые к тензору tens1, что означает, что первый и второй массивы будут преобразованы после операции сегментации (в нашем случае суммирования) и получим новый массив, который выглядит (2, 8, 1, 0) = (2+0, 5+3, 3-2, -5+5). Третий элемент в тензоре tens1 нетронутый, потому что он не сгруппирован ни в один повторный индекс, а последние два массива суммируются так же, как и в случае первой группы.




```
import tensorflow as tf

import numpy as np

def convert(v, t=tf.float32):

    return tf.convert_to_tensor(v, dtype=t)

seg_ids = tf.constant([0, 0, 1, 2, 2])

tens1 = convert(np.array([(2, 5, 3, -5), (0, 3, -2, 5), (4, 3, 5, 3), (6, 1, 4, 0), (6, 1, 4, 0)]), tf.int32)

tens2 = convert(np.array([1, 2, 3, 4, 5]), tf.int32)

seg_sum = tf.segment_sum(tens1, seg_ids)

seg_sum_1 = tf.segment_sum(tens2, seg_ids)

with tf.Session() as session:

    print("Segmentation sum tens1: ", session.run(seg_sum))
```

print("Segmentation sum tens2: ", session.run(seg_sum_1))

Вывод:

Segmentation sum tens1: [[2 8 1 0]

[4 3 5 3]

[12 2 8 0]]

Segmentation sum tens2: [3 3 9]

TensorFlow содержит также такие методы, как:

- **argmin,** которая возвращает индекс с минимальным значением по осям входного тензора,
- **argmax**, которая возвращает индекс с максимальным значением по осям входного тензора,
- **setdiff,** который вычисляет разницу между двумя списками чисел или строк.

Ниже мы приводим несколько примеров использования этих функций:

```
import numpy as np

import tensorflow as tf

def convert(v, t=tf.float32):

    return tf.convert_to_tensor(v, dtype=t)

x = convert(np.array([

    [2, 2, 1, 3],

    [4, 5, 6, -1],

    [0, 1, 1, -2],

    [6, 2, 3, 0]
```

```
]))

y = convert(np.array([1, 2, 5, 3, 7]))

z = convert(np.array([1, 0, 4, 6, 2]))

arg_min = tf.argmin(x, 1)

arg_max = tf.argmax(x, 1)

unique = tf.unique(y)

diff = tf.setdiff1d(y, z)

with tf.Session() as session:

    print("Argmin = ", session.run(arg_min))

    print("Argmax = ", session.run(arg_max))

    print("Unique_values = ", session.run(unique)[0])

    print("Unique_idx = ", session.run(unique)[1])

    print("Setdiff_values = ", session.run(diff)[0])

    print("Setdiff_idx = ", session.run(diff)[1])

    print(session.run(diff)[1])
```

Вывод:

```
Argmin =  [2 3 3 3]

Argmax =  [3 2 1 0]

Unique_values =  [1. 2. 5. 3. 7.]

Unique_idx =  [0 1 2 3 4]
```

Setdiff_values = [5. 3. 7.]

Setdiff_idx = [2 3 4]

[2 3 4]

18.8. Пример нейронной сети в TensorFlow

Рассмотрим пример применения **TensorFlow** для создания простой трехслойной нейронной сети. В этом примере мы будем использовать набор данных **MNIST** (и связанный с ним загрузчик), который предоставляет пакет TensorFlow. Этот набор данных **MNIST** представляет собой набор изображений в оттенках серого размером 28 × 28 пикселей, которые представляют собой рукописные цифры. Он имеет 55 000 учебных рядов, 10 000 строк тестирования и 5000 строк проверки.

Мы можем загрузить данные, запустив:

```
import tensorflow as tf

import numpy as np

from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

One_hot=True - аргумент указывает, что вместо меток, связанных с каждым проецирование изображения, сама цифра, то есть «4», то есть вектор с «один горячий» узел и все остальные узлы равно нулю, то есть [0, 0, 0, 0 , 1, 0, 0, 0, 0, 0]. Это позволяет нам легко подавать его в выходной слой нашей нейронной сети.

Затем мы можем настроить переменные-плейсхолдеры для данных обучения (и некоторые параметры обучения):

```
# Python optimisation variables
```

learning_rate = 0.5

epochs = 10

batch_size = 100

 # declare the training data placeholders

 # input x - for 28 x 28 pixels = 784

x = tf.placeholder(tf.float32, [None, 784])

 # now declare the output data placeholder - 10 digits

y = tf.placeholder(tf.float32, [None, 10])

Обратите внимание, что входной уровень **X** представляет содержит 784 узла, соответствующих 28 x 28 (= 784) пикселам, а выходной уровень **Y**- 10 узлов, соответствующих 10 возможных разрядов. Опять же, размер ***X*** равен (? X 784), где ? обозначает еще не заданное количество выборок, которые нужно ввести - это функция переменной-плейсхолдера.

Теперь нам нужно настроить переменные веса и смещения для трехслойной нейронной сети. Всегда должно быть *L-1* количество тензоров веса / смещения, где *L* - количество слоев. Поэтому в этом случае нам нужно настроить два тензора для каждого:

 # now declare the weights connecting the input to the hidden layer

W1 = tf.Variable(tf.random_normal([784, 300], stddev=0.03), name='W1')

b1 = tf.Variable(tf.random_normal([300]), name='b1')

 # and the weights connecting the hidden layer to the output layer

W2 = tf.Variable(tf.random_normal([300, 10], stddev=0.03), name='W2')

b2 = tf.Variable(tf.random_normal([10]), name='b2')

В этом коде мы объявляем некоторые переменные для **W1** и **b1**, веса и смещения для связей между входным и скрытым слоями. Эта нейронная сеть будет иметь 300 узлов в скрытом слое, поэтому размер весового тензора **W1** равен [784, 300]. Мы инициализируем значения весов, используя случайное нормальное распределение со средним значением равным нулю и стандартным отклонением 0,03. TensorFlow имеет реплицированную версию случайной нормальной функции **numpy**, которая позволяет вам создать матрицу заданного размера, заполненную случайными выборками, полученными из данного распределения. Аналогично, мы создаем переменные **W2** и **b2** для подключения скрытого слоя к выходному уровню нейронной сети.

Затем мы должны настроить входы узлов и функции активации узлов скрытого слоя:

 # calculate the output of the hidden layer

hidden_out = tf.add(tf.matmul(x, W1), b1)

hidden_out = tf.nn.relu(hidden_out)

В первой строке мы выполняем стандартное матричное умножение весов ***W1*** на входной вектор ***X*** и добавляем смещение ***b1***. Матричное умножение выполняется с использованием операции ***tf.matmul***. Затем мы завершаем операцию ***hidden_out***, применяя функцию активации **relu** к произведению матрицы весов **W1** и входа **X** плюс смещение.

Теперь давайте настроим выходной уровень, *y_*:

 # now calculate the hidden layer output - in this case, let's use a softmax activated

 # output layer

y_ = tf.nn.softmax(tf.add(tf.matmul(hidden_out, W2), b2))

Снова мы выполняем умножение веса с выходом из скрытого слоя (***hidden_out***) и добавляем смещение ***b2*** . В этом случае мы будем использовать активацию **softmax** для выходного уровня.

Мы также должны включить функцию затрат или. Здесь мы будем использовать функцию кросс-энтропии. Мы можем реализовать эту функцию кросс-энтропии в TensorFlow со следующим кодом:

 # now let's define the cost function which we are going to train the model on

y_clipped = tf.clip_by_value(y_, 1e-10, 0.9999999)

*cross_entropy = -tf.reduce_mean(tf.reduce_sum(y * tf.log(y_clipped) + (1 - y) * tf.log(1 - y_clipped), axis=1))*

Первая строка - операция, преобразующая выход **y_clipped** версию, ограниченную между 1e-10 и 0.999999. Вторая строка - расчет кросс-энтропии.

Чтобы выполнить этот расчет, сначала мы используем функцию ***tf.reduce_sum***, *которая* берет сумму заданной оси тензора.

Давайте настроим оптимизатор в TensorFlow:

 # add an optimiser

optimizer= tf.train.GradientDescentOptimizer(
learning_rate=learning_rate).minimize(cross_entropy)

Здесь мы просто используем оптимизатор градиентного спуска, предоставляемый с **TensorFlow**. Мы инициализируем его с помощью скорости обучения, а затем указываем, что мы хотим сделать, то есть свести к минимуму транзакционную операцию кросс-энтропии, которую мы создали. Затем эта функция выполнит градиентный спуск.

Далее настроим операцию инициализации переменных и операцию для измерения точности наших прогнозов:

 # finally setup the initialisation operator

init_op = tf.global_variables_initializer()

 # define an accuracy assessment operation

correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

 # add a summary to store the accuracy

tf.summary.scalar('accuracy', accuracy)

Операция предсказания ***correct_prediction*** использует функцию ***tf.equal*** *TensorFlow*, которая возвращает *True* или *False в* зависимости от того, равны ли его аргументы. Функция ***tf.argmax*** совпадает *с* функцией **numpy argmax**, которая возвращает индекс максимального значения в векторе или тензоре. Поэтому операция ***correct_prediction*** возвращает тензор размера *(*m *x* 1*)* True и False *значения*, определяющие, правильно ли предсказала цифру нейронная сеть. Затем мы хотим вычислить среднюю точность из этого тензора - сначала мы должны отличить тип операции ***correct_prediction*** от булева до плавающего *TensorFlow*, чтобы выполнить операцию ***reduce_mean***. Как только мы это сделаем, теперь у нас есть функция точности, которая готова оценить производительность нашей нейронной сети.

Теперь у нас есть все необходимое для настройки процесса обучения нашей нейронной сети. Запускаем процесс обучения нейронной сети.

merged = tf.summary.merge_all()

writer = tf.summary.FileWriter('C:\\D')

 # start the session

```
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
        print("Epoch:", (epoch + 1), "cost =", "{:.3f}".format(avg_cost))
        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}))
```

Запуск этой программы дает следующий результат:

```
Epoch: 1 cost = 0.586
Epoch: 2 cost = 0.213
```

Epoch: 3 cost = 0.150

Epoch: 4 cost = 0.113

Epoch: 5 cost = 0.094

Epoch: 6 cost = 0.073

Epoch: 7 cost = 0.058

Epoch: 8 cost = 0.045

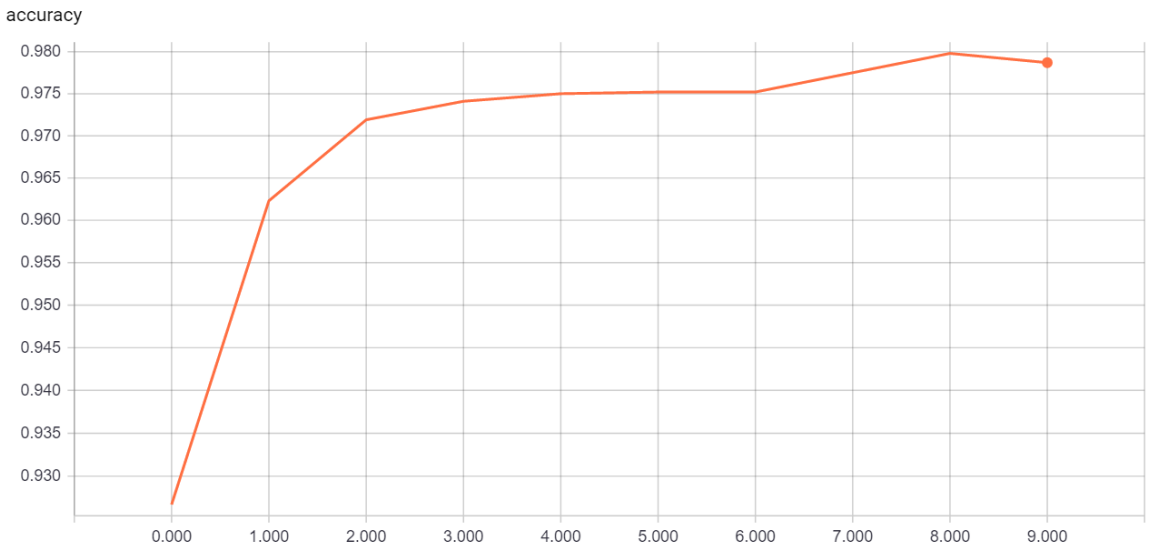
Epoch: 9 cost = 0.036

Epoch: 10 cost = 0.027

Training complete!

0.9787

Мы получаем примерно 98% точности на тестовом наборе что довольно неплохо для данной задачи. Мы могли бы сделать несколько вещей, чтобы улучшить модель, например, регуляризацию, но здесь нас просто интересует исследование основ TensorFlow. Вы также можете использовать визуализацию TensorBoard, чтобы посмотреть на повышение точности работы нейронной сети при обучении



Список литературы

1. Барский, А.Б. Логические нейронные сети: Учебное пособие / А.Б. Барский. - М.: Бином, 2013. - 352 с.
2. Галушкин, А.И. Нейронные сети: основы теории. / А.И. Галушкин. - М.: РиС, 2014. - 496 с.
3. Галушкин, А.И. Нейронные сети: история развития теории: Учебное пособие для вузов. / А.И. Галушкин, Я.З. Цыпкин. - М.: Альянс, 2015. - 840 с.
4. Редько, В.Г. Эволюция, нейронные сети, интеллект: Модели и концепции эволюционной кибернетики / В.Г. Редько. - М.: Ленанд, 2015. - 224 с.
5. Рутковская, Д. Нейронные сети, генетические алгоритмы и нечеткие системы / Д. Рутковская, М. Пилиньский, Л. Рутковский. - М.: РиС, 2013. - 384 с.
6. Ширяев, В.И. Финансовые рынки: Нейронные сети, хаос и нелинейная динамика / В.И. Ширяев. - М.: КД Либроком, 2016. - 232 с.
7. Яхъяева, Г.Э. Нечеткие множества и нейронные сети: Учебное пособие / Г.Э. Яхъяева. - М.: БИНОМ. ЛЗ, ИНТУИТ.РУ, 2012. - 316 с.
8. Суровцев И.С., Клюкин В.И., Пивоварова Р.П. Нейронные сети. — Воронеж: ВГУ, 1994. — 224 с.

9. Уоссермен Ф. Нейрокомпьютерная техника: теория и практика. — М.: Мир, 1992.
10. http://www.asimovinstitute.org/neural-network-zoo/
11. http://online.cambridgecoding.com/notebooks/cca_admin/convolutional-neural-networks-with-keras
12. https://habr.com/company/ods/blog/325432/