*«Реализация и применение нейронных сетей в программирование»*

*Выполнил ученик 380818-1 группы:*

*Кочанков Илья Дмитриевич*

*Научный руководитель:*

*Карпенко Сергей Николаевич*

Нижний Новгород 2019 год.

Содержание

[Введение. 4](#_Toc9278695)

[«Немного истории» 5](#_Toc9278696)

[«Математическая модель нейронной сети» 7](#_Toc9278697)

[«Преимущества нейронных сетей» 9](#_Toc9278698)

[«Недостатки нейронных сетей» 11](#_Toc9278699)

[«Принцип работы искусственного нейрона» 12](#_Toc9278700)

[Что такое нейрон? 12](#_Toc9278701)

[Что такое синапс? 13](#_Toc9278702)

[Что такое искусственный нейрон? 14](#_Toc9278703)

[«Самые известные и наиболее используемые функции активации» 16](#_Toc9278704)

[«Виды нейронных сетей» 19](#_Toc9278705)

[I. Сети прямого распространения 19](#_Toc9278706)

[II. Сети с обратными связями 21](#_Toc9278707)

[«Обучение нейронной сети» 23](#_Toc9278708)

[«Перцептрон» 26](#_Toc9278709)

[Алгоритм обратного распространения ошибки 34](#_Toc9278710)

[«Сверточная нейронная сеть» 44](#_Toc9278711)

[«Обучение сверточной нейросети» 54](#_Toc9278712)

[«Библиотека классов для работы с нейросетями» 60](#_Toc9278713)

[«Однослойный перцептрон» 63](#_Toc9278714)

[Постановка задачи 63](#_Toc9278715)

[Цифры в матричном формате 64](#_Toc9278716)

[Алгоритм обучения 64](#_Toc9278717)

[Программа 1 65](#_Toc9278718)

[«Однослойный перцептрон (усовершенствование)» 68](#_Toc9278719)

[Постановка задачи 68](#_Toc9278720)

[Алгоритм обучения 68](#_Toc9278721)

[Программа 2 69](#_Toc9278722)

[«Многослойный перцептрон» 73](#_Toc9278723)

[Постановка задачи 73](#_Toc9278724)

[Алгоритм обучения 73](#_Toc9278725)

[Программа 3 73](#_Toc9278726)

[«Рукописный ввод цифр на базе многослойного перцептрона» 79](#_Toc9278727)

[Постановка задачи 79](#_Toc9278728)

[«Сверточная нейросеть» 82](#_Toc9278729)

[Постановка задачи 82](#_Toc9278730)

[Программа 4 84](#_Toc9278731)

[Рукописный ввод цифр на базе сверточной нейросети» 94](#_Toc9278732)

[Постановка задачи 94](#_Toc9278733)

[«Заключение» 97](#_Toc9278734)

[«Исходный код библиотеки для работы с нейросетями» 98](#_Toc9278735)

[Сверточная нейронная сеть 98](#_Toc9278736)

[Base\_Cnn.h 98](#_Toc9278737)

[NeyronCnn.h 99](#_Toc9278738)

[CNNLearning.h 100](#_Toc9278739)

[CNNLearns.h 103](#_Toc9278740)

[CNNs.h 103](#_Toc9278741)

[Перцептрон 103](#_Toc9278742)

[Base\_Perceptron.h 103](#_Toc9278743)

[NeyronPerceptron.h 104](#_Toc9278744)

[PerceptronLearning.h 105](#_Toc9278745)

[PLearns.h 108](#_Toc9278746)

[Perceptrons.h 108](#_Toc9278747)

[Вспомогательные классы 108](#_Toc9278748)

[Matrix.h 108](#_Toc9278749)

[Filter.h 114](#_Toc9278750)

[Weights.h 116](#_Toc9278751)

[Func.h 119](#_Toc9278752)

[Functors.h 119](#_Toc9278753)

[«Список используемых источников и литературы» 120](#_Toc9278754)

Введение.

Нейронная сеть (биологическая нейронная сеть) — совокупность нейронов головного и спинного мозга центральной нервной системы (ЦНС) и ганглия периферической нервной системы (ПНС), которые связаны или функционально объединены в нервной системе, выполняют специфические физиологические функции.

Первые представления о нейронных сетях отмечены у Бэйна (1873) в своих работах он рассматривал мыслительную деятельность как результат взаимодействия между нейронами в головном мозге.

В наши дни мы часто можем услышать по телевизору словосочетание «нейронные сети», плохо представляя, что же это такое.

Нейросеть – это обучаемая система. Она действует не только в соответствии с заданным алгоритмом и формулами, но и на основании прошлого опыта. Нейросеть применяется там, где алгоритм решения задачи и правила решения неизвестны. Например, как нам распознать лицо на фото? Какие математические формулы использовать? Тут уравнения не помогут. Тут нужен опыт. А опыт получается при обучении. Возможность обучения — одно из главных преимуществ нейронных сетей перед традиционными алгоритмами. Это дает им невероятный потенциал в деле разработки искусственного интеллекта. Посудите сами: запрограммировать все возможные ситуации в память робота невозможно даже в теории. Но если искусственный разум сможет сам получать опыт, он со временем разовьется до той степени, когда окажется способен справиться даже с той ситуацией, к которой его не готовили.

Сегодня нейросети применяются не только в компьютерных технологиях, но и в медицине, в педагогике, на финансовом рынке, в сложных маркетинговых исследованиях, в области телекоммуникаций и информационных систем — цепочку достижений можно перечислять, наверное, до бесконечности. К примеру, с помощью нейросимуляторов можно определить цвет глаз и внешний вид будущего ребенка, диагностировать слух у грудных детей, и даже распознать вид грибов.

Специалисты пророчат еще более ошеломительные достижения в развитии искусственных нейронных сетей во всех областях человеческой деятельности и самое главное – в интеллектуальной.

«Немного истории»

1943 — У. Маккалок и У. Питтс формализуют понятие нейронной сети в фундаментальной статье о логическом исчислении идей и нервной активности. В начале своего сотрудничества с Питтсом Н. Винер предлагает ему вакуумные лампы в качестве идеального на тот момент средства для реализации эквивалентов нейронных сетей.

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 то время казалось, что уже не за горами создание полноценного искусственного интеллекта.

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.

В 1963 году в Институте проблем передачи информации АН СССР. А. П. Петровым проводится подробное исследование задач «трудных» для перцептрона. Эта пионерская работа в области моделирования ИНС в СССР послужила отправной точкой для комплекса идей М. М. Бонгарда — как «сравнительно небольшой переделкой алгоритма (перцептрона) исправить его недостатки».

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. Изобретение не привлекло особого внимания.

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 независимо и одновременно С. И. Барцевым и В. А. Охониным (Красноярская группа) переоткрыт и существенно развит метод обратного распространения ошибки. Начался взрыв интереса к обучаемым нейронным сетям.

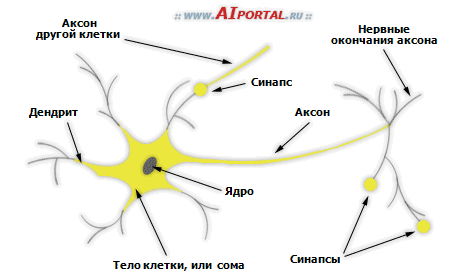
2007 Джеффри Хинтоном в университете Торонто созданы алгоритмы глубокого обучения многослойных нейронных сетей. Успех обусловлен тем, что Хинтон при обучении нижних слоев сети использовал ограниченную машину Больцмана (RBM — Restricted Boltzmann Machine). Глубокое обучение по Хинтону — это очень медленный процесс. Необходимо использовать много примеров распознаваемых образов (например, множество лиц людей на разных фонах). После обучения получается готовое быстро работающее приложение, способное решать конкретную задачу (например, осуществлять поиск лиц на изображении). Функция поиска лиц людей на сегодняшний день стала стандартной и встроена во все современные цифровые фотоаппараты. Технология глубокого обучения активно используется интернет-поисковиками при классификации картинок по содержащимся в них образам. Применяемые при распознавании искусственные нейронные сети могут иметь до 9 слоев нейронов, их обучение ведётся на миллионах изображений с отыскиваемым образом.

«Математическая модель нейронной сети»

Нейронная сеть или нервная система человека – это сложная сеть структур человека, обеспечивающая взаимосвязанное поведение всех систем организма.

Биологический нейрон – это специальная клетка, которая структурно состоит из ядра, тела клетки и отростков. Одной из ключевых задач нейрона является передача электрохимического импульса по всей нейронной сети через доступные связи с другими нейронами. Притом, каждая связь характеризуется некоторой величиной, называемой силой синаптической связи. Эта величина определяет, что произойдет с электрохимическим импульсом при передаче его другому нейрону: либо он усилится, либо он ослабится, либо останется неизменным.

Биологическая нейронная сеть обладает высокой степенью связности: на один нейрон может приходиться несколько тысяч связей с другими нейронами. Но, это приблизительное значение и в каждом конкретном случае оно разное. Передача импульсов от одного нейрона к другому порождает определенное возбуждение всей нейронной сети. Величина этого возбуждения определяет реакцию нейронной сети на какие-то входные сигналы. Например, встреча человека со старым знакомым может привести к сильному возбуждению нейронной сети, если с этим знакомым связаны какие-то яркие и приятные жизненные воспоминания. В свою очередь сильное возбуждение нейронной сети может привести к учащению сердцебиения, более частому морганию глаз и к другим реакциям. Встреча же с незнакомым человеком для нейронной сети пройдет практически незаметной, а значит и не вызовет каких-либо сильных реакций.



Сильно упрощенная модель биологической нейронной сети

Иску́сственная нейро́нная се́ть (ИНС) — математическая модель, а также её программное или аппаратное воплощение, построенная по принципу организации и функционирования биологических нейронных сетей — сетей нервных клеток живого организма. Это понятие возникло при изучении процессов, протекающих в мозге, и при попытке смоделировать эти процессы. После разработки алгоритмов обучения получаемые модели стали использовать в практических целях: в задачах прогнозирования, для распознавания образов, в задачах управления и др.

ИНС представляют собой систему соединённых и взаимодействующих между собой простых процессоров (искусственных нейронов). Такие процессоры обычно довольно просты (особенно в сравнении с процессорами, используемыми в персональных компьютерах). Каждый процессор подобной сети имеет дело только с сигналами, которые он периодически получает, и сигналами, которые он периодически посылает другим процессорам. И, тем не менее, будучи соединёнными в достаточно большую сеть с управляемым взаимодействием, такие по отдельности простые процессоры вместе способны выполнять довольно сложные задачи.



Схема строения нейронной сети

«Преимущества нейронных сетей»

* Устойчивость к шумам входных данных

Представьте себе людей на пешеходном переходе. Вы без труда окидываете всех их взглядом и легко различаете лица. Однако рассмотрим эту картину подробнее. Помимо непосредственно лиц на изображении есть еще и асфальт, одежда людей, машины, светофор, сумки. И вся это ненужная (шумовая) информация тоже подается нам в глаза!

Но мы абсолютно не обращаем на нее внимания, мастерски различая лица. Как я уже говорил выше, мы просто натренировались их различать. Важная часть тренировки — игнорирование шумовых сигналов.

Это качество есть и у искусственных нейронных сетей. После тренировки они способны не обращать внимание на входы, на которые подаются шумовые данные.

* Адаптация к изменениям

Пусть у вас есть нейронная сеть, которая прогнозирует рост/падение цен на бирже. Однако постепенно, день за днем, ситуация на рынке меняется. Если бы ваша сеть не адаптировалась к этим изменениям, то она перестала бы давать правильные ответы уже через неделю. Но искусственные нейронные сети, обучаясь на данных, каждый раз подстраиваются под среду.

* Отказоустойчивость

Иногда случается, так, что в результате наследственных заболеваний или других проблемах человеку приходится удалять половину головного мозга. Такие случаи действительно бывают. Поразительно то, что за определенное время оставшееся полушарие берет на себя функции исчезнувшего. Может быть не в полной мере, однако система (человек) продолжает функционировать.

Это свойство проявляется и у искусственных нейронных сетей. Они могут выдавать корректные результаты даже при значительном повреждении составляющих их компонентов.

* Сверхвысокое быстродействие

Компьютер выполняет команды последовательно. Однако в голове человека каждый нейрон является маленьким процессором (который принимает сигнал, преобразует его, и подает на выход). И таких процессоров у нас в голове миллиарды. Получаем гигантскую сеть распределенных вычислений. Сигнал обрабатывается нейронами одновременно.

Это свойство потенциально проявляется и в искусственных нейронных сетях. Если у вас многоядерный компьютер, то это свойство будет выполняться. Для одноядерных компьютеров никакой разницы заметно не будет.

«Недостатки нейронных сетей»

* Недостатки нейронных сетей

Нейронные сети не способны давать точные и однозначные ответы.

* Многошаговое принятие решений

Нейроны искусственной нейросети, в общем случае, не зависят друг от друга. Они просто получают сигнал, преобразуют его и отдают дальше. Они не смотрят друг на друга и, в зависимости от нейрона-соседа, меняют свои синапсы. Отсюда следует, что нейронная сеть может решать задачу только в один заход, залпом.

Поэтому совершенно бесполезно просить нейросеть доказать теорему. Там требуется цепочка последовательных шагов.

* Вычислительные задачи

Этот недостаток искусственных нейронных сетей в какой-то степени является следствием двух предыдущих недостатков.

Первая проблема – очередность. Надо каким-то образом, используя только входы сети, указать ей какая часть выражения находится под корнем, а какая часть находится слева от знака равенства. Да и как передать сам знак равенства?

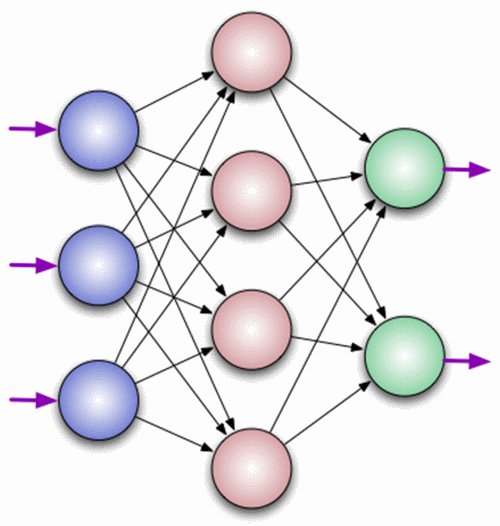
Предположим, что вы каким-то образом смогли доставить эти данные в сеть

Вторая проблема – последовательные шаги. Уже описанный выше недостаток.

И главный камень преткновения – невозможность выдачи точных результатов.

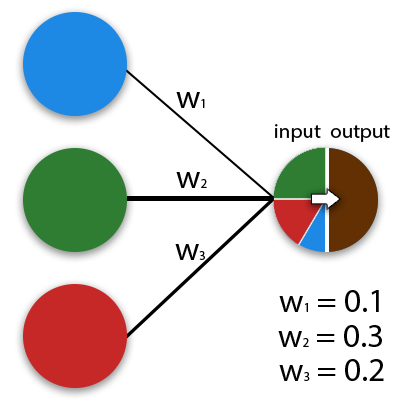
«Принцип работы искусственного нейрона»

Что такое нейрон?



Нейрон — это вычислительная единица, которая получает информацию, производит над ней простые вычисления и передает ее дальше. Они делятся на три основных типа: входной (синий), скрытый (красный) и выходной (зеленый). В том случае, когда нейросеть состоит из большого количества нейронов, вводят термин слоя. Соответственно, есть входной слой, который получает информацию, n скрытых слоев (обычно их не больше 3), которые ее обрабатывают и выходной слой, который выводит результат. У каждого из нейронов есть 2 основных параметра: входные данные (input data) и выходные данные (output data). В случае входного нейрона: input=output. В остальных, в поле input попадает суммарная информация всех нейронов с предыдущего слоя, после чего, она нормализуется, с помощью функции активации и попадает в поле output.

Что такое синапс?



Синапс — это связь между двумя нейронами. У синапсов есть 1 параметр — вес. Благодаря ему, входная информация изменяется, когда передается от одного нейрона к другому. Допустим, есть 3 нейрона, которые передают информацию следующему. Тогда у нас есть 3 веса, соответствующие каждому из этих нейронов. У того нейрона, у которого вес будет больше, та информация и будет доминирующей в следующем нейроне (пример — смешение цветов). На самом деле, совокупность весов нейронной сети или матрица весов — это своеобразный мозг всей системы. Именно благодаря этим весам, входная информация обрабатывается и превращается в результат.

Что такое искусственный нейрон?



Модель искусственного нейрона

У каждого нейрона, в том числе и у искусственного, должны быть какие-то входы, через которые он принимает сигнал. Поступившие на входы сигналы умножаются на свои веса. Сигнал первого входа x\_1 умножается на соответствующий этому входу вес w\_1 . В итоге получаем x\_1 w\_1 . И так до n -ого входа. В итоге на последнем входе получаем x\_n w\_n.

Теперь все произведения передаются в сумматор. Уже исходя из его названия можно понять, что он делает. Он просто суммирует все входные сигналы, умноженные на соответствующие веса:

x_1 w_1 + x_2 w_2 + \cdots + x_n w_n = \sum\limits_{i=1}^n x_i w_i 

Результатом работы сумматора является число, называемое взвешенной суммой.

Роль сумматора очевидна – он агрегирует все входные сигналы (которых может быть много) в какое-то одно число – взвешенную сумму, которая характеризует поступивший на нейрон сигнал в целом. Еще взвешенную сумму можно представить, как степень общего возбуждения нейрона.

Просто так подавать взвешенную сумму на выход достаточно бессмысленно. Нейрон должен как-то обработать ее и сформировать адекватный выходной сигнал. Именно для этих целей и используют функцию активации.

Она преобразует взвешенную сумму в какое-то число, которое и является выходом нейрона (выход нейрона обозначим переменной out ).

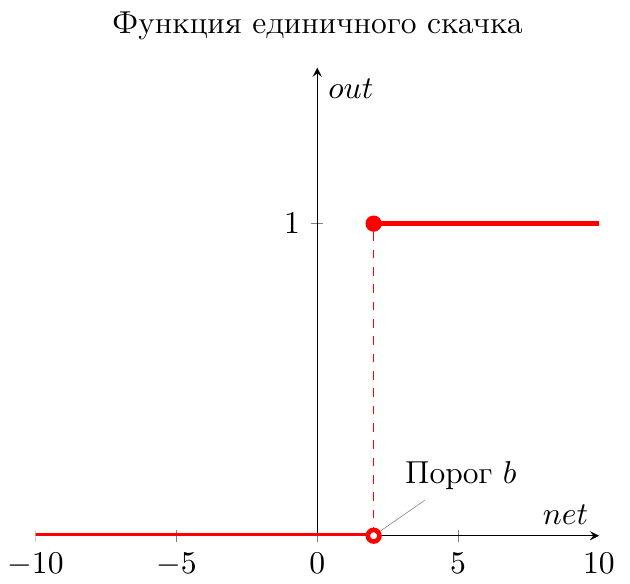
Для разных типов искусственных нейронов используют самые разные функции активации. В общем случае их обозначают символом f(net) . Указание взвешенного сигнала в скобках означает, что функция активации принимает взвешенную сумму как параметр.

«Самые известные и наиболее используемые функции активации»

* Функция единичного скачка

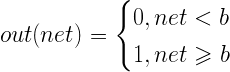
Самый простой вид функции активации. Выход нейрона может быть равен только 0 или 1. Если взвешенная сумма больше определенного порога b, то выход нейрона равен 1. Если ниже, то 0.

Графически эту функцию активации можно изобразить следующим образом:



На горизонтальной оси расположены величины взвешенной суммы. На вертикальной оси — значения выходного сигнала. Как легко видеть, возможны только два значения выходного сигнала: 0 или 1. Причем 0 будет выдаваться всегда от минус бесконечности и вплоть до некоторого значения взвешенной суммы, называемого порогом. Если взвешенная сумма равна порогу или больше него, то функция выдает 1.

Теперь запишем эту функцию активации математически:



* Сигмоидальная функция (Сигмоида)

На самом деле существует целое семейство сигмоидальных функций, некоторые из которых применяют в качестве функции активации в искусственных нейронах.

Все эти функции обладают некоторыми очень полезными свойствами, ради которых их и применяют в нейронных сетях. Мы рассмотрим самую простую и часто используемую логистическую сигмоиду.

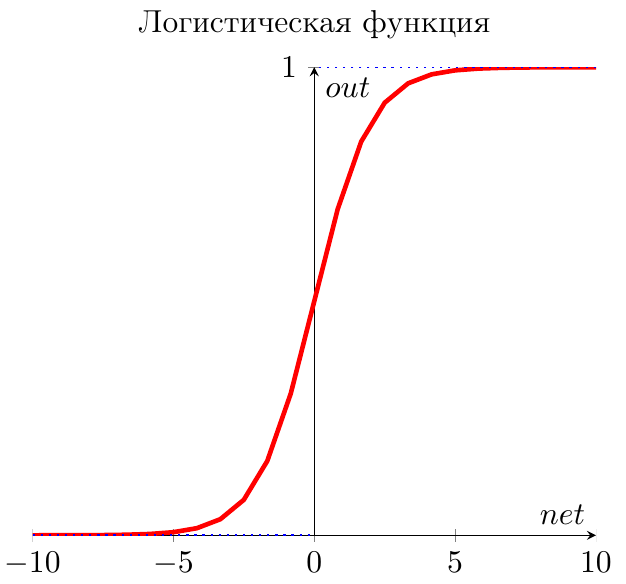
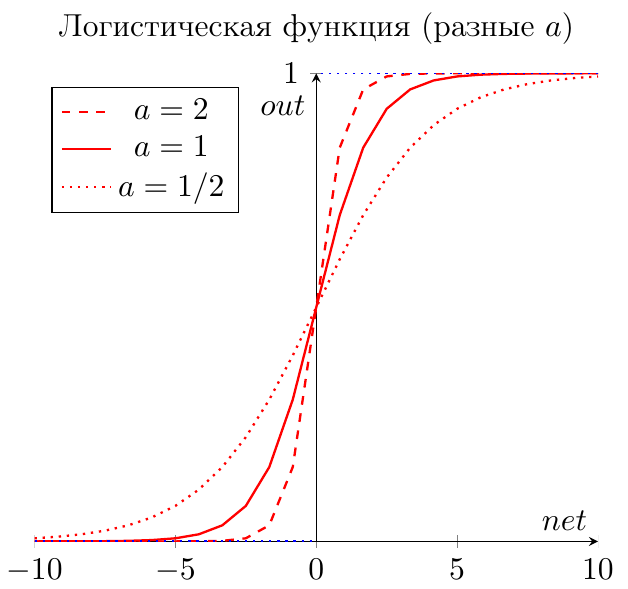


График этой функции выглядит достаточно просто. Если присмотреться, то можно увидеть некоторое подобие английской буквы S , откуда и пошло название семейства этих функций.

А вот так она записывается аналитически:

out(net) = \frac{1}{1+\exp(-a \cdot net)} 

Что за параметр a? Это какое-то число, которое характеризует степень крутизны функции. Ниже представлены логистические функции с разным параметром a.



Использование логистической функции в качестве функции активации приведет к тому, что вы будете получать цифру между 0 и 1. Причем чем больше взвешенная сумма, тем ближе выход будет к 1 (но никогда не будет точно ей равен). И наоборот, чем меньше взвешенная сумма, тем ближе выход нейрона будет к 0.

Свойства логической функции:

* Является «сжимающей» функцией, то есть вне зависимости от аргумента (взвешенной суммы), выходной сигнал всегда будет в пределах от 0 до 1
* Она более гибкая, чем функция единичного скачка – ее результатом может быть не только 0 и 1, но и любое число между ними
* Во всех точках она имеет производную, и эта производная может быть выражена через эту же функцию

«Виды нейронных сетей»

Мы разобрались со структурой искусственного нейрона. Искусственные нейронные сети состоят из совокупности искусственных нейронов. Возникает логичный вопрос – а как располагать/соединять друг с другом эти самые искусственные нейроны?

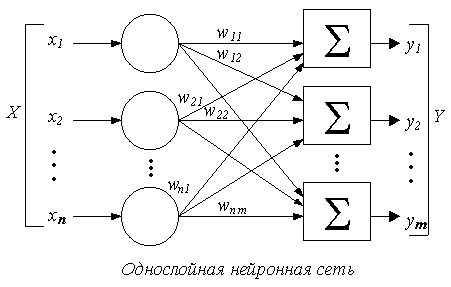
1. Сети прямого распространения

Сети прямого распространения – искусственные нейронные сети, в которых сигнал распространяется строго от входного слоя к выходному. В обратном направлении сигнал не распространяется.

1. Однослойные нейронные сети

Однослойная нейронная сеть – сеть, в которой сигналы от входного слоя сразу подаются на выходной слой, который и преобразует сигнал и сразу же выдает ответ.

Выглядит однослойная нейронная сеть следующим образом:

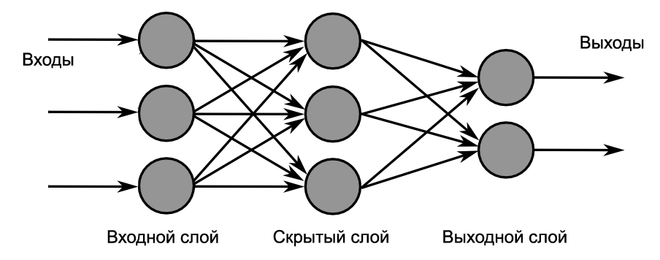


На этой картинке входной слой обозначен кружками (он не считается за слой нейронной сети), а справа расположен слой обычных нейронов.

Нейроны соединены друг с другом стрелками. Над стрелками расположены веса соответствующих связей (весовые коэффициенты).

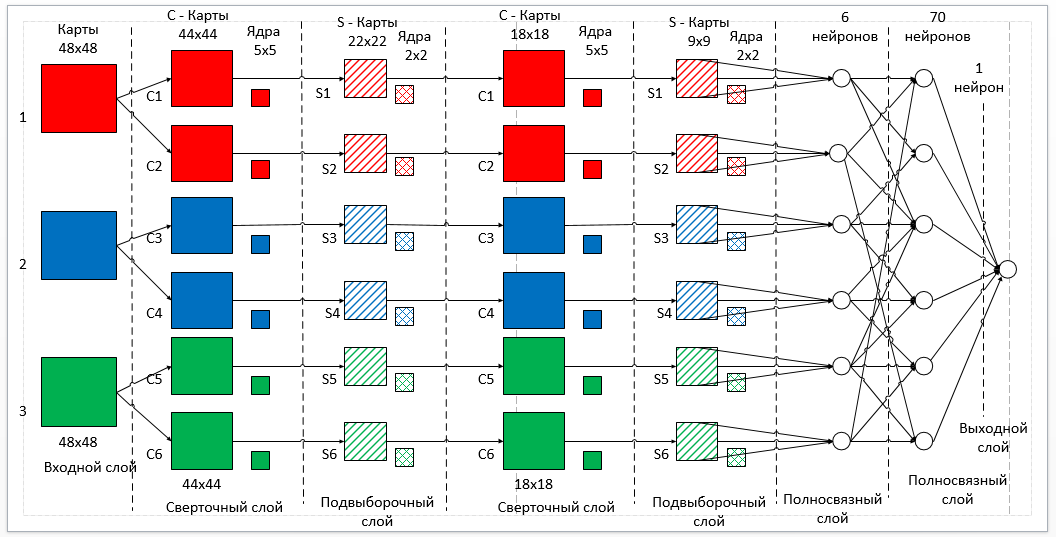
1. Многослойные нейронные сети

Многослойная нейронная сеть – нейронная сеть, состоящая из входного, выходного и расположенного(ых) между ними одного (нескольких) скрытых слоев нейронов.



Такая структура нейронных сетей копирует многослойную структуру определенных отделов мозга.

Многослойные нейронные сети обладают гораздо большими возможностями, чем однослойные.

1. Сверточные нейронные сети

Наилучшие результаты в области распознавания лиц показала Convolutional Neural Network или сверточная нейронная сеть, которая является логическим развитием идей таких архитектур нейронных сетей как когнитрона и неокогнитрона. Успех обусловлен возможностью учета двумерной топологии изображения, в отличие от многослойного персептрона.

Сверточные нейронные сети обеспечивают частичную устойчивость к изменениям масштаба, смещениям, поворотам, смене ракурса и прочим искажениям. Сверточные нейронные сети объединяют три архитектурных идеи, для обеспечения инвариантности к изменению масштаба, повороту сдвигу и пространственным искажениям.

На данный момент сверточная нейронная сеть и ее модификации считаются лучшими по точности и скорости алгоритмами нахождения объектов на сцене. Начиная с 2012 года, нейросети занимают первые места на известном международном конкурсе по распознаванию образов ImageNet.

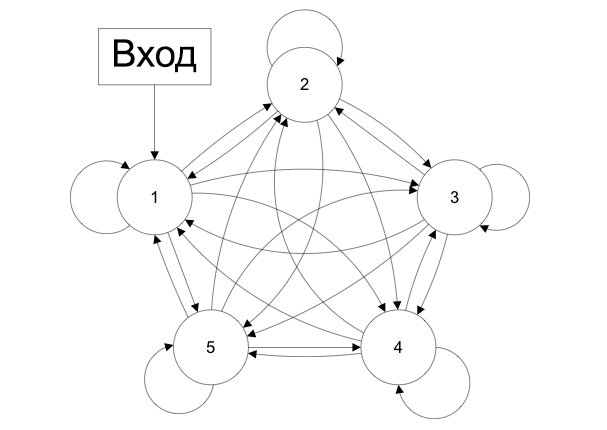
1. Сети с обратными связями

Сети с обратными связями – искусственные нейронные сети, в которых выход нейрона может вновь подаваться на его вход. В более общем случае это означает возможность распространения сигнала от выходов к входам.

В сетях такого типа сигнал может идти и в обратную сторону. В чем преимущество?

Дело в том, что в сетях прямого распространения выход сети определяется входным сигналом и весовыми коэффициентами при искусственных нейронах.

А в сетях с обратными связями выходы нейронов могут возвращаться на входы. Это означает, что выход какого-нибудь нейрона определяется не только его весами и входным сигналом, но еще и предыдущими выходами (так как они снова вернулись на входы).



Возможность сигналов циркулировать в сети открывает новые, удивительные возможности нейронных сетей. С помощью таких сетей можно создавать нейросети, восстанавливающие или дополняющие сигналы. Другими словами, такие нейросети имеют свойства кратковременной памяти (как у человека).

«Обучение нейронной сети»

Искусственная нейронная сеть – это совокупность искусственных нейронов. Теперь давайте возьмем, например, 100 нейронов и соединим их друг с другом. Ясно, что при подаче сигнала на вход, мы получим что-то бессмысленное на выходе.

Изменять общее количество искусственных нейронов бессмысленно по двум причинам. Во-первых, увеличение количества вычислительных элементов в целом лишь делает систему тяжеловеснее и избыточнее. Во-вторых, если вы соберете 1000 дураков вместо 100, то они все-равно не смогут правильно ответить на вопрос.

Сумматор изменить не получится, так как он выполняет одну жестко заданную функцию – складывать. Если мы его заменим на что-то или вообще уберем, то это вообще уже не будет искусственным нейроном.

Если менять у каждого нейрона функцию активации, то мы получим слишком разношерстную и неконтролируемую нейронную сеть. К тому же, в большинстве случаев нейроны в нейронных сетях одного типа. То есть они все имеют одну и ту же функцию активации.

Остается только один вариант – менять веса связей. Значит, обучение нейронной сети – поиск такого набора весовых коэффициентов, при котором входной сигнал после прохода по сети преобразуется в нужный нам выходной.

Такой подход к термину «обучение нейронной сети» соответствует и биологическим нейросетям. Наш мозг состоит из огромного количества связанных друг с другом нейросетей. Каждая из них в отдельности состоит из нейронов одного типа (функция активации одинаковая). Мы обучаемся благодаря изменению синапсов – элементов, которые усиливают/ослабляют входной сигнал.

Однако есть еще один важный момент. Если обучать сеть, используя только один входной сигнал, то сеть просто «запомнит правильный ответ». Со стороны будет казаться, что она очень быстро «обучилась». И как только вы подадите немного измененный сигнал, ожидая увидеть правильный ответ, то сеть выдаст бессмыслицу.

В самом деле, зачем нам сеть, определяющая лицо только на одном фото. Мы ждем от сети способности обобщать какие-то признаки и узнавать лица и на других фотографиях тоже.

Именно с этой целью и создаются обучающие выборки.

Обучающая выборка – конечный набор входных сигналов (иногда вместе с правильными выходными сигналами), по которым происходит обучение сети.

После обучения сети, то есть когда сеть выдает корректные результаты для всех входных сигналов из обучающей выборки, ее можно использовать на практике.

Однако прежде чем пускать свежеиспеченную нейросеть в бой, часто производят оценку качества ее работы на так называемой тестовой выборке.

Тестовая выборка – конечный набор входных сигналов (иногда вместе с правильными выходными сигналами), по которым происходит оценка качества работы сети.

Существует два разных подхода к обучению нейронной сети:

1. Обучение с учителем

Обучение с учителем – вид обучения сети, при котором ее веса меняются так, чтобы ответы сети минимально отличались от уже готовых правильных ответов.

Суть данного подхода заключается в том, что вы даете на вход сигнал, смотрите на ответ сети, а затем сравниваете его с уже готовым, правильным ответом.

Затем, с помощью специальных алгоритмов, вы меняете веса связей нейронной сети и снова даете ей входной сигнал. Сравниваете ее ответ с правильным и повторяете этот процесс до тех пор, пока сеть не начнет отвечать с приемлемой точностью.

1. Обучение без учителя

Обучение без учителя – вид обучения сети, при котором сеть самостоятельно классифицирует входные сигналы. Правильные (эталонные) выходные сигналы не демонстрируются.

Обучение без учителя применяют тогда, когда у нас нет правильных ответов на входные сигналы. В этом случае вся обучающая выборка состоит из набора входных сигналов.

Что же происходит при таком обучении сети? Оказывается, что при таком «обучении» сеть начинает выделять классы подаваемых на вход сигналов. Короче говоря – сеть начинает кластеризацию.

Например, вы демонстрируете сети конфеты, пирожные и торты. Вы никак не регулируете работу сети. Вы просто подаете на ее входы данные о данном объекте. Со временем сеть начнет выдавать сигналы трех разных типов, которые и отвечают за объекты на входе.

«Перцептрон»

Перцептро́н, или персептрон (англ. perceptron от лат. perceptio — восприятие) — математическая или компьютерная модель восприятия информации мозгом (кибернетическая модель мозга), предложенная Фрэнком Розенблаттом в 1957 году и впервые реализованная в виде электронной машины «Марк-1» в 1960 году. Перцептрон стал одной из первых моделей нейросетей, а «Марк-1» — первым в мире нейрокомпьютером.

Согласно современной терминологии, перцептроны могут быть классифицированы как искусственные нейронные сети:

* с одним скрытым слоем;
* с пороговой передаточной функцией;
* с прямым распространением сигнала.

В основе персептрона лежит математическая модель восприятия

информации мозгом. Персептроны стали очень активно исследовать. На них

возлагали большие надежды. Однако, как оказалось, они имели серьезные

ограничения. Был такой ученый Минский, который был сокурсником

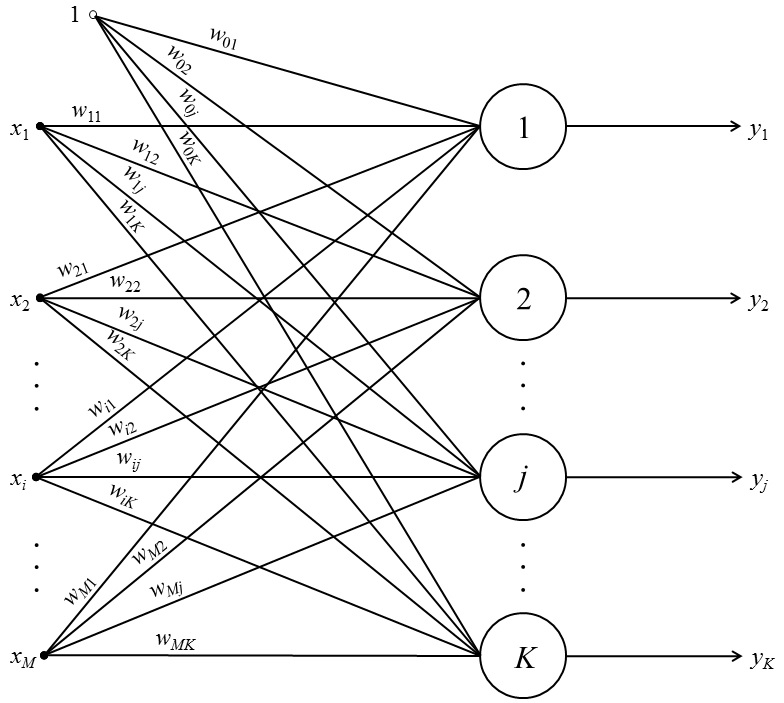
Розенблатта. Видимо, ему не очень понравилось, как все вокруг боготворили

персептроны, и он написал целую книгу (1971 год), в которой провел

детальнейший их анализ, попутно показав, что они не так уж много и умеют,

да и вообще сильно ограничены.

Однослойные перцептроны



На рисунке представлена структура однослойного перцептрона с M входами и K выходами. Очевидно, что каждый выход соответствует своему нейрону единственного слоя. Кроме того, ясно, что сложность структуры однослойной сети не может варьироваться ввиду отсутствия скрытых слоев нейронов.

Количество весовых коэффициентов Nw, настраиваемых в процессе обучения, рассчитывается следующим образом:



Для каждого нейрона сети помимо синаптических связей с элементами входного вектора настраивается связь с фиктивным единичным входом (коэффициент смещения).

Так как выходные переменные могут принимать как бинарные, так и аналоговые значения, выбор вида активационных функций ограничен только областью допустимых значений выходных сигналов, принятой для нормализации.

Однослойные перцептроны обучаются на основе итерационного **метода Уидроу–Хоффа**, иначе называемого дельта-правилом. Алгоритм данного метода следующий:

1. Весовые коэффициенты однослойного перцептрона выбранной структуры инициализируются небольшими по абсолютной величине (не более M –1) случайными значениями.

2. На входы перцептрона подается входной вектор одного из примеров обучающей выборки. Производится прямое распространение сигналов по сети с расчетом значений выходных переменных https://neuronus.com/images/theory/ins/24042015/03.jpg.

3. Для каждого рассчитанного значения выходной переменной вычисляется погрешность по сравнению со значениями элементов выходного вектора взятого обучающего примера: https://neuronus.com/images/theory/ins/24042015/04.jpg

https://neuronus.com/images/theory/ins/24042015/05.jpg                                          (1)

4. Выполняется коррекция старых значений весовых коэффициентов каждого нейрона https://neuronus.com/images/theory/ins/24042015/06.jpg на основе погрешности соответствующей выходной переменной:

https://neuronus.com/images/theory/ins/24042015/07.jpg                                 (2)

где v – коэффициент скорости обучения.

5. Цикл повторяется с шага 2 до выполнения одного или нескольких условий окончания:

– исчерпано заданное предельное количество эпох обучения;

– достигнут удовлетворительный уровень ошибки по всей обучающей выборке;

– не происходит уменьшения ошибки обучающей выборки на протяжении заданного предельного количества эпох обучения;

На протяжении одной эпохи обучения последовательно или в случайном порядке предъявляются все примеры обучающей выборки, причем каждый пример должен быть предъявлен однократно.

Коэффициент скорости обучения задается положительной константой или переменной величиной (0 < v <= 1), постепенно уменьшающейся в процессе обучения нейронной сети.

Общая ошибка работы нейронной сети с обучающей выборкой оценивается по соотношению:

https://neuronus.com/images/theory/ins/24042015/08.jpg                      (3)

Также полезно оценивать ошибку работы нейронной сети в отдельности по каждому ее выходу:

https://neuronus.com/images/theory/ins/24042015/09.jpg                             (4)

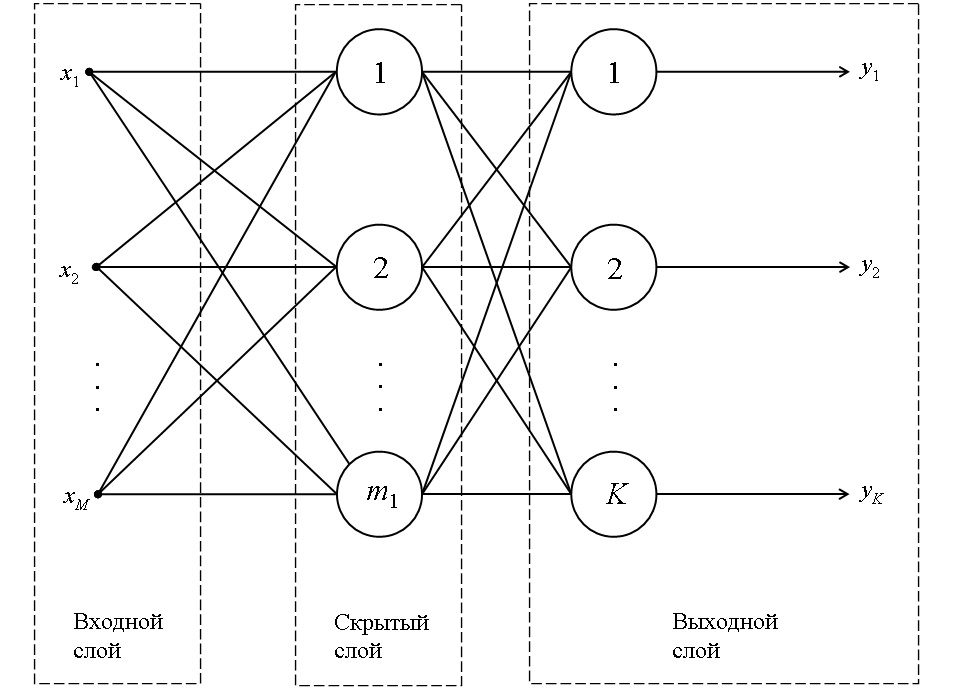
Порядок величины полученных в последнем случае ошибок сопоставим со средней абсолютной ошибкой нормализованных выходных переменных. Как следствие, с ее помощью можно оценить порядок величины абсолютной ошибки в исходных единицах:

https://neuronus.com/images/theory/ins/24042015/10.jpg                             (5)

или относительной ошибки в процентах:

https://neuronus.com/images/theory/ins/24042015/11.jpg                                              (6)

Многослойные перцептроны

[](https://neuronus.com/images/theory/ins/07052015/00.jpg)

Многослойные перцептроны эффективны при решении тех же самых задач, что и однослойные перцептроны, но обладают значительно большей вычислительной способностью в сравнении с однослойными перцептронами. Благодаря этой своей способности они могут гораздо точнее описывать многомерные зависимости с большой степенью нелинейности и высоким уровнем перекрестного и группового влияния входных переменных на выходные.

Пример двухслойного перцептрона с M входами и K выходами был ранее приведен на рисунке. При необходимости использования сетей более сложной структуры добавляются новые скрытые слои или наращивается количество нейронов в скрытых слоях.

Количество весовых коэффициентов, настраиваемых в процессе обучения многослойного перцептрона с L скрытыми слоями по ml нейронов в каждом, рассчитывается следующим образом:

https://neuronus.com/images/theory/ins/07052015/01.jpg.

Для каждого нейрона сети помимо синаптических связей с элементами входного вектора настраивается связь с фиктивным единичным входом (коэффициент смещения).

Для большинства задач, решаемых с помощью многослойных перцептронов, выбор структуры нейронной сети должен осуществляться на основе следующего правила («Правила 2–5»): количество настраиваемых в процессе обучения весовых коэффициентов должно быть в 2–5 раз меньше, чем количество примеров обучающей выборки. Если это соотношение меньше 2, сеть теряет способность к обобщению обучающей информации, а при достижении 1 и меньше просто запоминает ответы для каждого обучающего примера. Если же количество обучающих примеров слишком велико для выбранной структуры сети, нейросетевая модель во многих случаях просто усредняет выходные значения для различных комбинаций входных векторов, теряя способность к корректному отклику в отдельных частных случаях и повышая величину максимальной выборочной ошибки.

Кроме того, при выборе структуры многослойного перцептрона следует задавать количество нейронов в скрытом слое, предшествующем выходному слою, не меньшим, чем количество самих выходов.

Метод Уидроу–Хоффа, рассмотренный в предыдущей главе, не может быть использован для настройки весовых коэффициентов многослойных перцептронов, поскольку он позволяет сделать прямую коррекцию по величине ошибки только для нейронов выходного слоя, тогда как синаптические коэффициенты скрытых нейронов также требуют изменения.

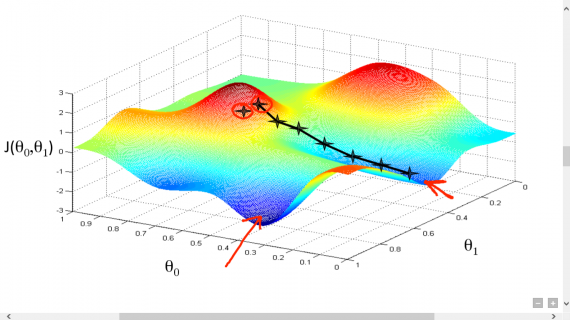
Задача обучения многослойных перцептронов может быть сформулирована как оптимизационная, в которой целевой функцией (критерием оптимизации) является общая ошибка, рассчитанная по обучающей выборке. Соответственно, и решать данную задачу можно как любую задачу многомерной оптимизации с использованием методов детерминированного, градиентного или стохастического поиска.

Градиентный спуск — самый используемый алгоритм обучения, он применяется почти в каждой модели машинного обучения. Градиентный спуск — это, по сути, и есть то, как обучаются модели. Без ГС машинное обучение не было бы там, где сейчас. Метод градиентного спуска с некоторой модификацией широко используется для обучения персептрона и глубоких нейронных сетей, и известен как метод обратного распространения ошибки.

Градиентный спуск — метод нахождения минимального значения функции потерь (существует множество видов этой функции). Минимизация любой функции означает поиск самой глубокой впадины в этой функции. Имейте в виду, что функция используется, чтобы контролировать ошибку в прогнозах модели машинного обучения. Поиск минимума означает получение наименьшей возможной ошибки или повышение точности модели. Мы увеличиваем точность, перебирая набор учебных данных при настройке параметров нашей модели (весов и смещений).

Итак, градиентный спуск нужен для минимизации функции потерь.

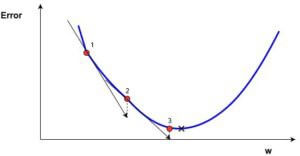
Суть алгоритма – процесс получения наименьшего значения ошибки. Аналогично это можно рассматривать как спуск во впадину в попытке найти золото на дне ущелья (самое низкое значение ошибки).



Поиск минимума функции

В дальнейшем, чтобы найти самую низкую ошибку (глубочайшую впадину) в функции потерь (по отношению к одному весу), нужно настроить параметры модели. Как мы их настраиваем? В этом поможет математический анализ. Благодаря анализу мы знаем, что наклон графика функции – производная от функции по переменной. Это наклон всегда указывает на ближайшую впадину!

На рисунке мы видим график функции потерь (названный «Ошибка» с символом «J») с одним весом. Теперь, если мы вычислим наклон (обозначим это dJ/dw) функции потерь относительно одного веса, то получим направление, в котором нужно двигаться, чтобы достичь локальных минимумов. Давайте пока представим, что наша модель имеет только один вес.



Функция потерь

Важно: когда мы перебираем все учебные данные, мы продолжаем добавлять значения  dJ/dw для каждого веса. Так как потери зависят от примера обучения, dJ/dw также продолжает меняться. Затем делим собранные значения на количество примеров обучения для получения среднего. Потом мы используем это среднее значение (каждого веса) для настройки каждого веса.

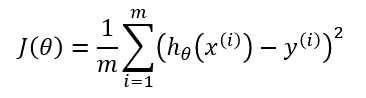
Также обратите внимание: Функция потерь предназначена для отслеживания ошибки с каждым примером обучениям, в то время как производная функции относительного одного веса – это то, куда нужно сместить вес, чтобы минимизировать ее для этого примера обучения. Вы можете создавать модели даже без применения функции потерь. Но вам придется использовать производную относительно каждого веса (dJ/dw).

Теперь, когда мы определили направление, в котором нужно подтолкнуть вес, нам нужно понять, как это сделать. Тут мы используем коэффициент скорости обучения, его называют гипер-параметром. Гипер-параметр – значение, требуемое вашей моделью, о котором мы действительно имеем очень смутное представление. Обычно эти значения могут быть изучены методом проб и ошибок. Здесь не так: одно подходит для всех гипер-параметров. Коэффициент скорости обучения можно рассматривать как «шаг в правильном направлении», где направление происходит от dJ/dw.

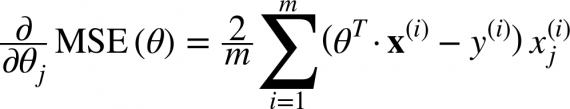
Это была функция потерь построенная на один вес. В реальной модели мы делаем всё перечисленное выше для всех весов, перебирая все примеры обучения. Даже в относительно небольшой модели машинного обучения у вас будет более чем 1 или 2 веса. Это затрудняет визуализацию, поскольку график будет обладать размерами, которые разум не может себе представить.

Подробнее о градиентах

Кроме функции потерь градиентный спуск также требует градиент, который является dJ/dw (производная функции потерь относительно одного веса, выполненная для всех весов). dJ/dw зависит от вашего выбора функции потерь. Наиболее распространена функция потерь среднеквадратичной ошибки.



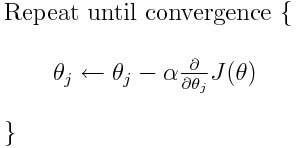
Производная этой функции относительно любого веса (эта формула показывает вычисление градиента для [*линейной регрессии*](https://neurohive.io/ru/osnovy-data-science/linejnaja-regressija/)):



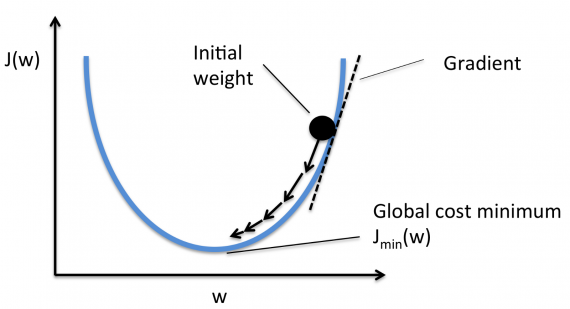
Это – вся математика в ГС. Глядя на это можно сказать, что по сути, ГС не содержит много математики. Единственная математика, которую он содержит  в себе – умножение и деление, до которых мы доберемся. Это означает, что ваш выбор функции повлияет на вычисление градиента каждого веса.

Коэффициент скорости обучения

Однако проблема у большинства моделей возникает с коэффициентом скорости обучения. Давайте взглянем на обновленное выражение для каждого веса (j лежит в диапазоне от 0 до количества весов, а Theta-j это j-й вес в векторе весов, k лежит в диапазоне от 0 до количества смещений, где B-k — это k-е смещение в векторе смещений). Здесь alpha – коэффициент скорости обучения. Из этого можно сказать, что мы вычисляем dJ/dTheta-j ( градиент веса Theta-j), и затем шаг размера alpha в этом направлении. Следовательно, мы спускаемся по градиенту. Чтобы обновить смещение, замените Theta-j на B-k.

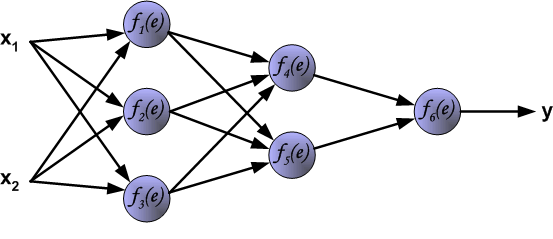


Если этот размера шага alpha слишком велик, мы преодолеем минимум: то есть промахнемся мимо минимума. Если alpha слишком мала, мы используем слишком много итераций, чтобы добраться до минимума.

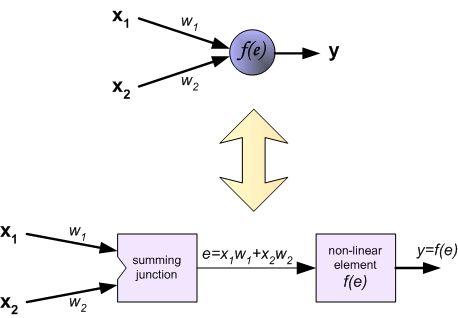
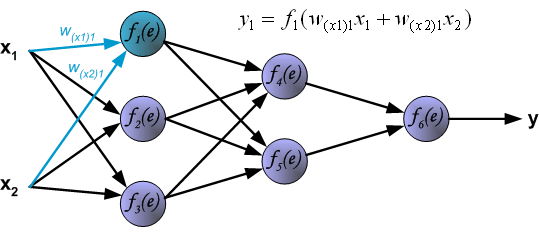
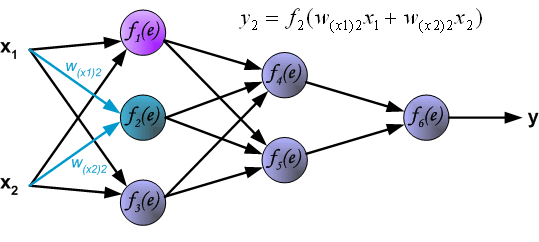
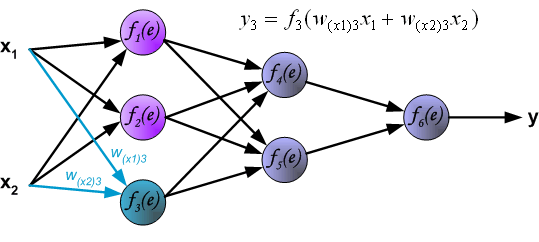
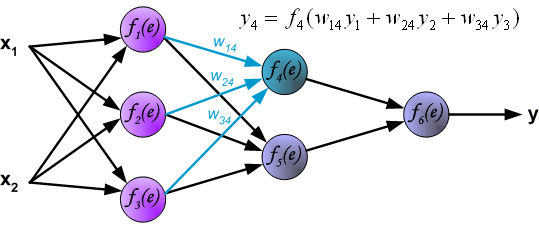
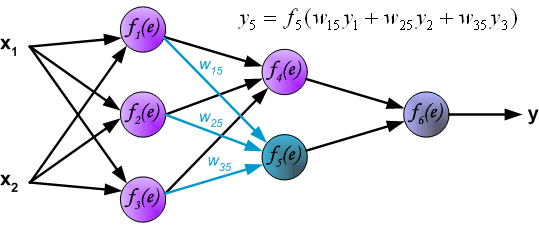
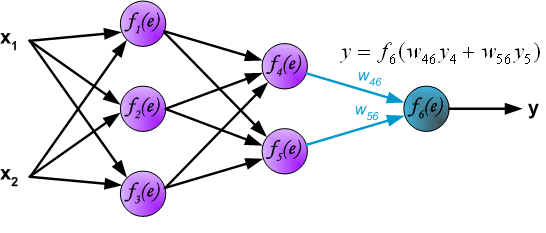
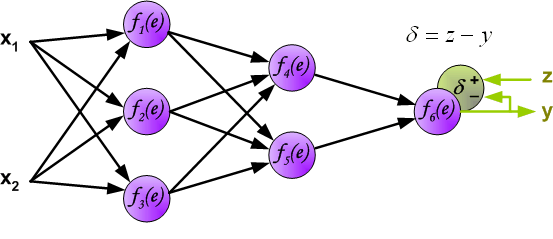


Алгоритм обратного распространения ошибки

Рассмотрим процесс обучения нейронной сети с использованием алгоритма обратного распространения ошибки (backpropagation).  
 Для иллюстрации этого процесса используем нейронную сеть состоящую из трёх слоёв и имеющую два входа и один выход:

  
 Каждый нейрон состоит из двух элементов. Первый элемент – дендриты — добавляют весовые коэффициенты ко входным сигналам.   
Второй элемент – тело — реализует нелинейную функцию, т.н. функцию активации нейрона. Сигнал е – это взвешенная сумма входных сигналов

у = f (е)

— выходной сигнал нейрона.   
  
  
 Чтобы обучить нейронную сеть мы должны подготовить обучающие данные(примеры).   
В нашем случае, тренировочные данные состоят из входных сигналов (х1 и х2) и желаемого результата z.   
 Обучение – это последовательность итераций (повторений).  
В каждой итерации весовые коэффициенты нейронов подгоняются с использованием новых данных из тренировочных примеров.   
Изменение весовых коэффициентов и составляют суть алгоритма, описанного ниже.  
  
 Каждый шаг обучения начинается с воздействия входных сигналов из тренировочных примеров. После этого мы можем определить значения выходных сигналов для всех нейронов в каждом слое сети.   
Иллюстрации ниже показывают, как сигнал распространяется по сети.   
Символы W(Xm)n представляют вес связи между сетевым входом Xm и нейрона n во входном слое.   
Символы y(n) представляют выходной сигнал нейрона n.  
  
  
  
  
 Распространение сигнала через скрытый слой. Символы Wmn представляют весовые множители связей между выходом нейрона m и входом нейрона n в следующем слое.  
  
  
 Распространение сигнала через выходной слой  
  
 На следующем шаге алгоритма, выходной сигнала сети y сравнивается с желаемым выходным сигналом z, который хранится в тренировочных данных.   
 Разница между этими двумя сигналами называется ошибкой d выходного слоя сети.  
  
  
 Невозможно непосредственно вычислить сигнал ошибки для внутренних нейронов, потому что выходные значения этих нейронов, неизвестны.

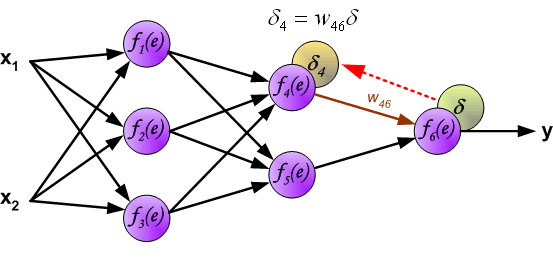
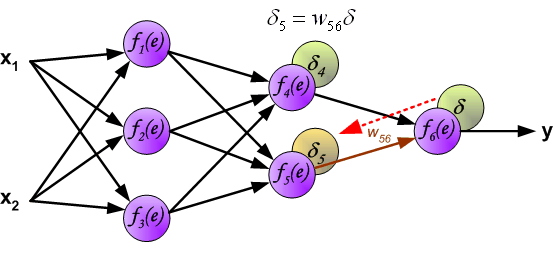
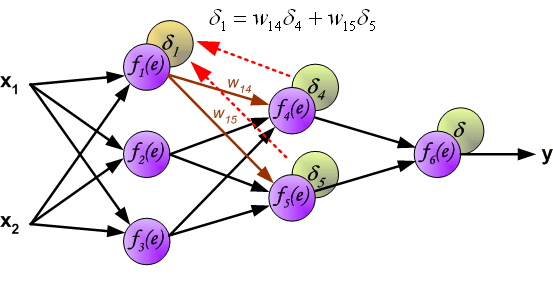
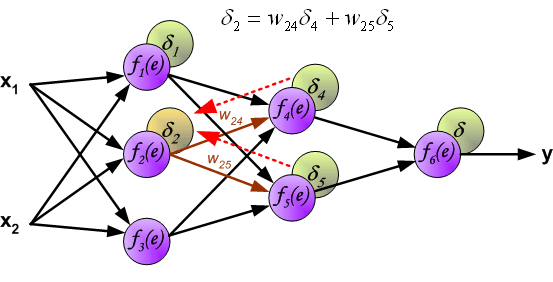
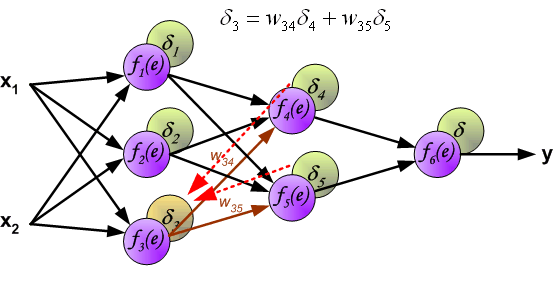
На протяжении многих лет был неизвестен эффективный метод для

обучения многослойной сети. Только в середине восьмидесятых годов был

разработан алгоритм обратного распространения ошибки.

Идея заключается в распространении сигнала ошибки d (вычисленного в

шаге обучения) обратно на все нейроны, чьи выходные сигналы были

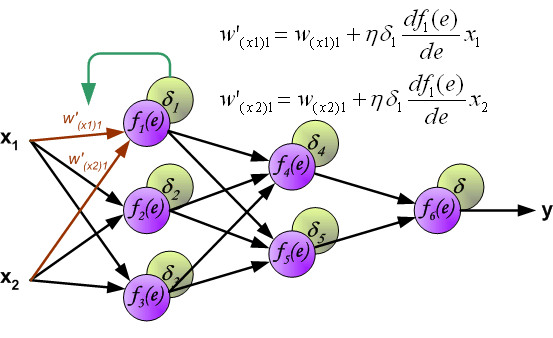
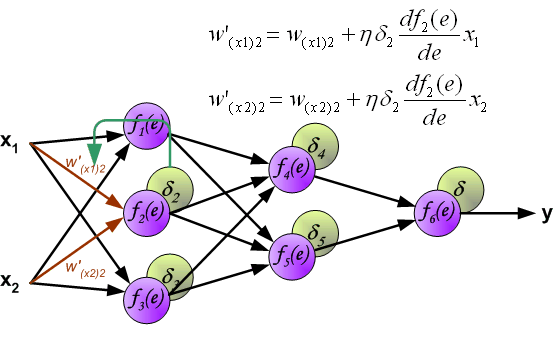
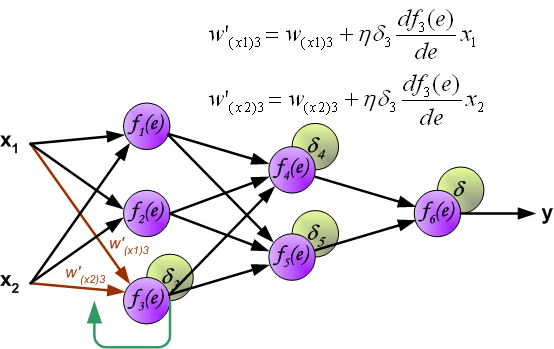
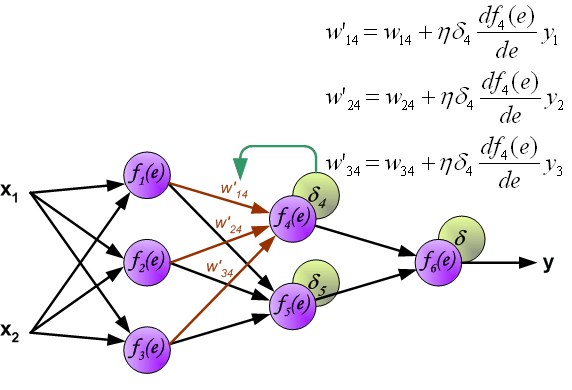
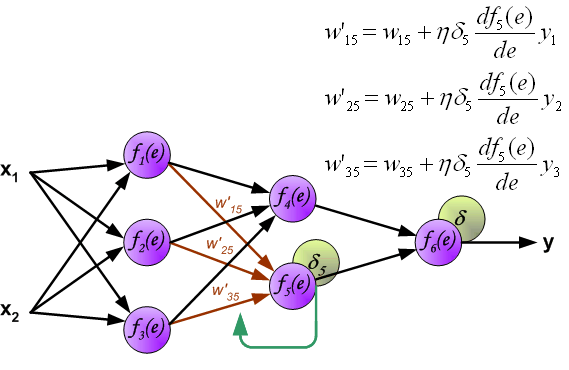
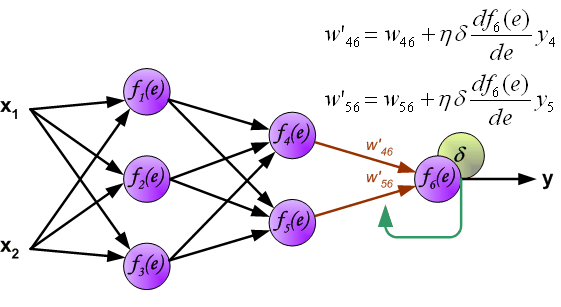
входящими для последнего нейрона.  
  
  
 Весовые коэффициенты Wmn, используемые для обратного распространения ошибки, равны тем же коэффициентам, что использовались во время вычисления выходного сигнала. Только изменяется направление потока данных (сигналы передаются от выхода ко входу).   
 Этот процесс повторяется для всех слоёв сети. Если ошибка пришла от нескольких нейронов — она суммируются:  
  
  
  
 Когда вычисляется величина ошибки сигнала для каждого нейрона – можно скорректировать весовые коэффициенты каждого узла ввода(дендрита) нейрона.  
 В формулах ниже df(e)/de — является производной от функции активации нейрона (чьи весовые коэффициенты корректируются).

как [помним](http://robocraft.ru/blog/algorithm/558.html), для активационной функции типа сигмоид   
            1  
S(x) = -----------  
       1 + exp(-x)

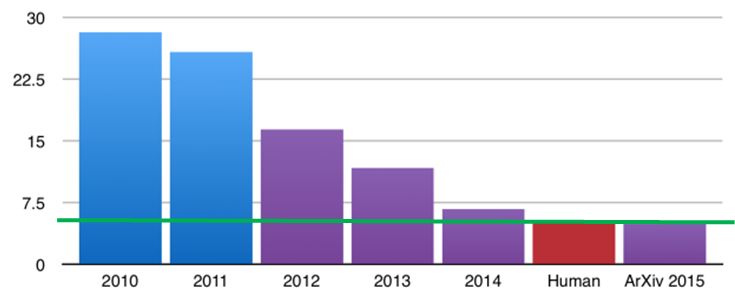
производная выражается через саму функцию:

S'(x) = S(x)\*(1 - S(x))

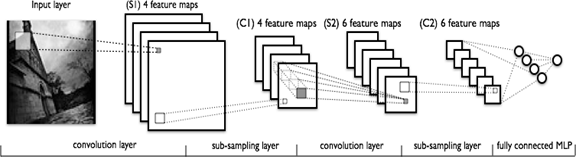
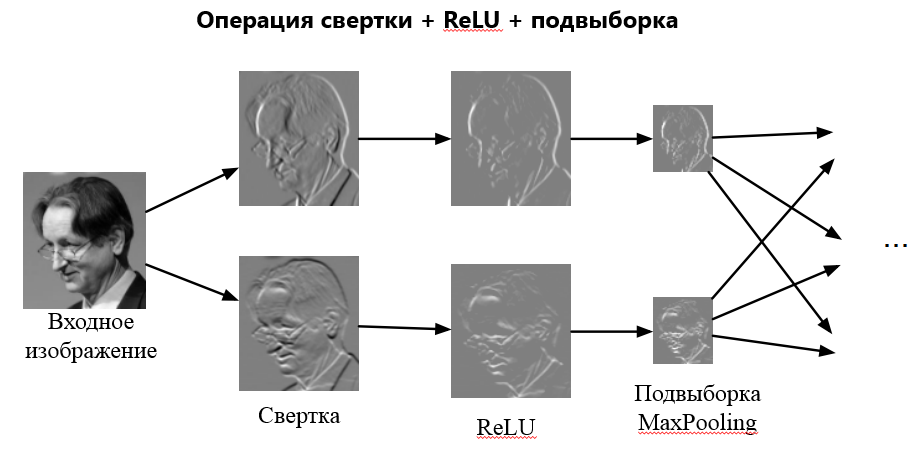
, что позволяет существенно сократить вычислительную сложность метода обратного распространения ошибки.

  
  
  
  
  
  
  
 Коэффициент h влияет на скорость обучения сети.   
Есть несколько методов для выбора этого параметра.   
Первый способ — начать учебный процесс с большим значением параметра h. Во время коррекции весовых коэффициентов, параметр постепенно уменьшают.   
 Второй — более сложный метод обучения, начинается с малым значением параметра h. В процессе обучения параметр увеличивается, а затем вновь уменьшается на завершающей стадии обучения.  
Начало учебного процесса с низким значением параметра h позволяет определить знак весовых коэффициентов.

«Сверточная нейронная сеть»

На данный момент сверточная нейронная сеть и ее модификации считаются лучшими по точности и скорости алгоритмами нахождения объектов на сцене. Начиная с 2012 года, нейросети занимают первые места на известном международном конкурсе по распознаванию образов ImageNet.  
  


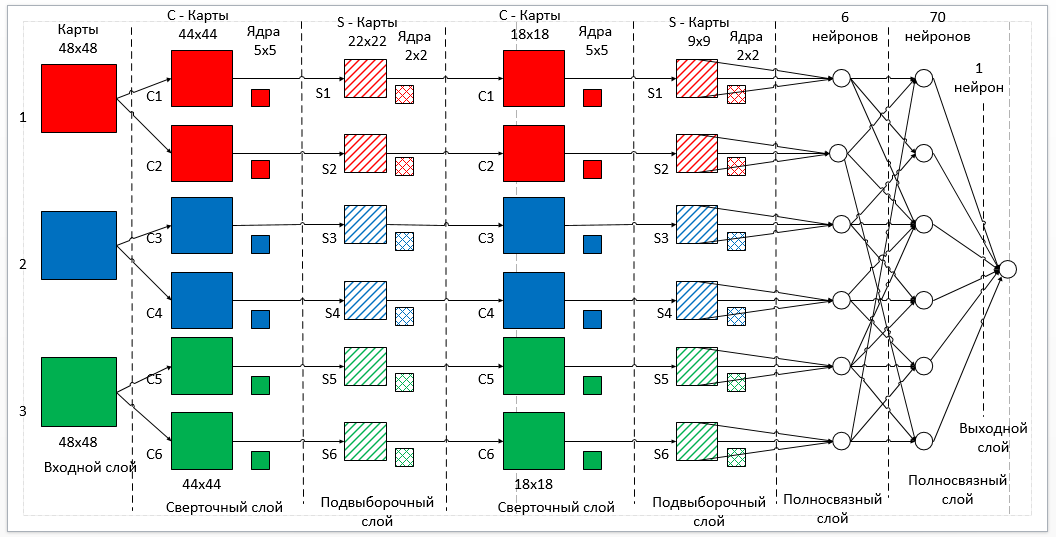
Структура сверточной нейронной сети

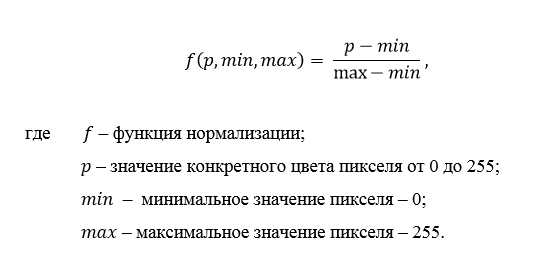
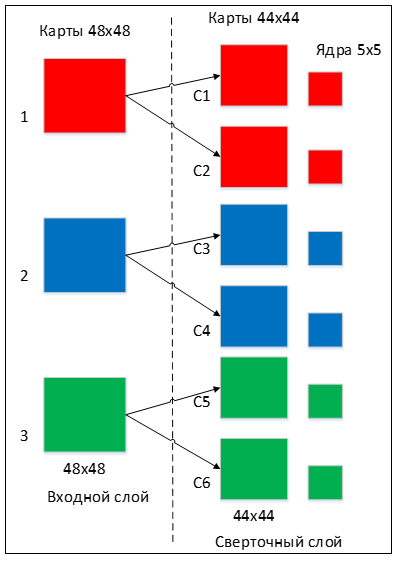
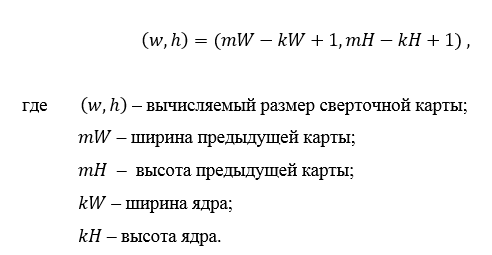
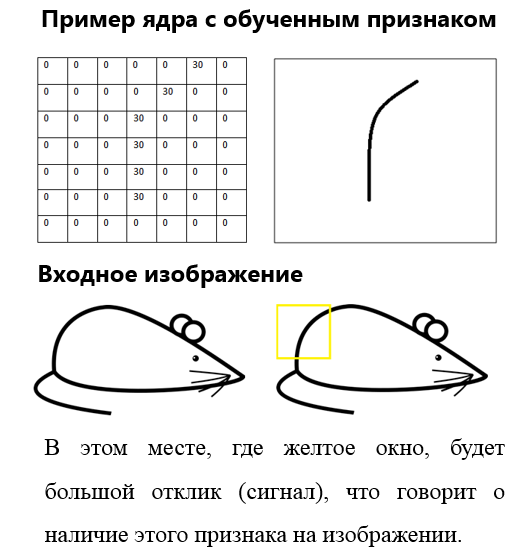
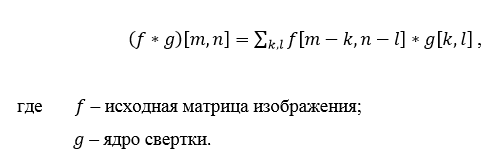
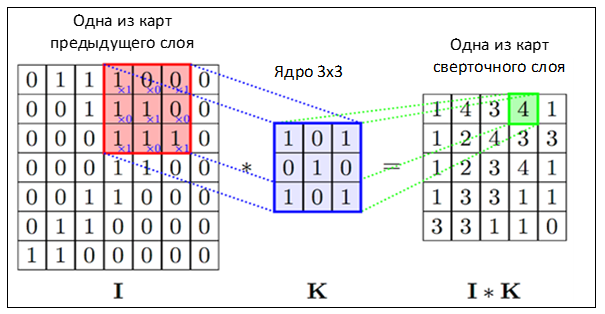
СНС состоит из разных видов слоев: сверточные (convolutional) слои, субдискретизирующие (subsampling, подвыборка) слои и слои «обычной» нейронной сети – персептрона, в соответствии с рисунком 1.  
  
  
Рисунок 1 – топология сверточной нейронной сети   
  
 Первые два типа слоев (convolutional, subsampling), чередуясь между собой, формируют входной вектор признаков для многослойного персептрона.  
 Свое название сверточная сеть получила по названию операции – свертка, суть которой будет описана дальше.   
 Сверточные сети являются удачной серединой между биологически правдоподобными сетями и обычным многослойным персептроном. На сегодняшний день лучшие результаты в распознавании изображений получают с их помощью. В среднем точность распознавания таких сетей превосходит обычные ИНС на 10-15%. СНС – это ключевая технология Deep Learning.   
 Основной причиной успеха СНС стало концепция общих весов. Несмотря на большой размер, эти сети имеют небольшое количество настраиваемых параметров по сравнению с их предком – неокогнитроном. Имеются варианты СНС (Tiled Convolutional Neural Network), похожие на неокогнитрон, в таких сетях происходит, частичный отказ от связанных весов, но алгоритм обучения остается тем же и основывается на обратном распространении ошибки. СНС могут быстро работать на последовательной машине и быстро обучаться за счет чистого распараллеливания процесса свертки по каждой карте, а также обратной свертки при распространении ошибки по сети.  
 На рисунке ниже продемонстрирована визуализация свертки и подвыборки:  
  


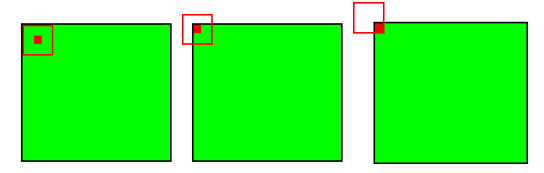
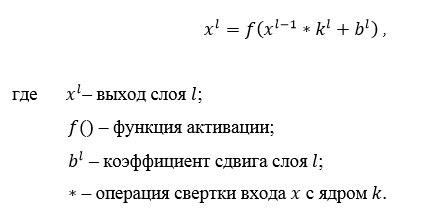
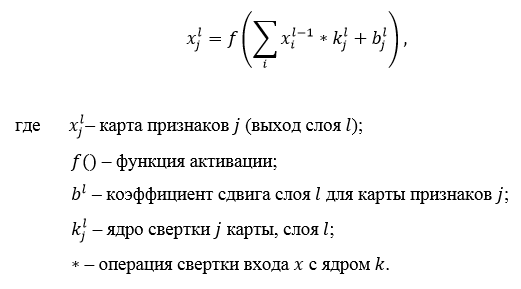
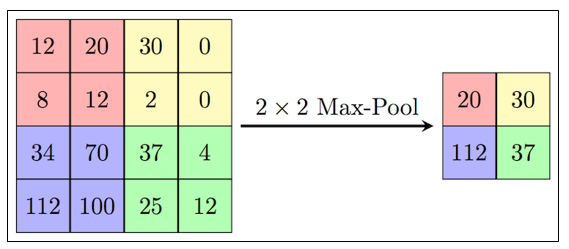
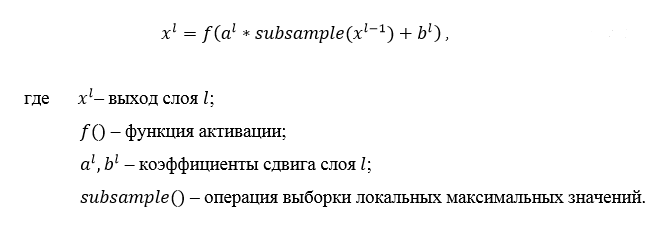
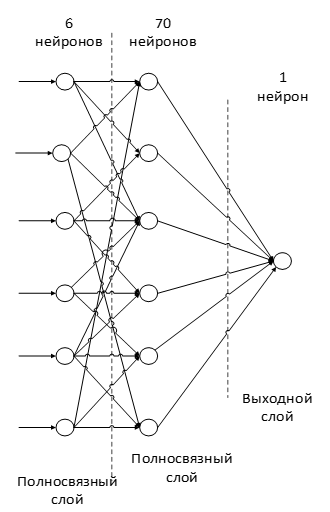
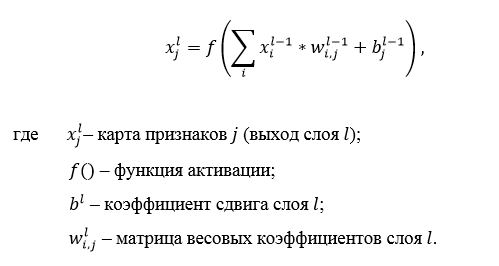
Топология сверточной нейросети

Определение топологии сети ориентируется на решаемую задачу, данные из научных статей и собственный экспериментальный опыт.  
  
Можно выделить следующие этапы влияющие на выбор топологии:

* определить решаемую задачу нейросетью (классификация, прогнозирование, модификация);
* определить ограничения в решаемой задаче (скорость, точность ответа);
* определить входные (тип: изображение, звук, размер: 100x100, 30x30, формат: RGB, в градациях серого) и выходных данные (количество классов).

  
Рисунок 2 — Топология сверточной нейросети

Входные данные представляют из себя цветные. Если размер будет слишком велик, то вычислительная сложность повысится, соответственно ограничения на скорость ответа будут нарушены, определение размера в данной задаче решается методом подбора. Если выбрать размер слишком маленький, то сеть не сможет выявить ключевые признаки.   
 Входной слой учитывает двумерную топологию изображений и состоит из нескольких карт (матриц), карта может быть одна, в том случае, если изображение представлено в оттенках серого, иначе их 3, где каждая карта соответствует изображению с конкретным каналом (красным, синим и зеленым).  
 Входные данные каждого конкретного значения пикселя нормализуются в диапазон от 0 до 1, по формуле:   
  
  
 Сверточный слой представляет из себя набор карт (другое название – карты признаков, в обиходе это обычные матрицы), у каждой карты есть синаптическое ядро (в разных источниках его называют по-разному: сканирующее ядро или фильтр).   
 Количество карт определяется требованиями к задаче, если взять большое количество карт, то повысится качество распознавания, но увеличится вычислительная сложность.   
  
Рисунок 3 — Организация связей между картами сверточного слоя и предыдущего  
 Размер у всех карт сверточного слоя – одинаковы и вычисляются по формуле 2:   
  
  
 Ядро представляет из себя фильтр или окно, которое скользит по всей области предыдущей карты и находит определенные признаки объектов. Например, если сеть обучали на множестве лиц, то одно из ядер могло бы в процессе обучения выдавать наибольший сигнал в области глаза, рта, брови или носа, другое ядро могло бы выявлять другие признаки. Размер ядра обычно берут в пределах от 3х3 до 7х7. Если размер ядра маленький, то оно не сможет выделить какие-либо признаки, если слишком большое, то увеличивается количество связей между нейронами. Также размер ядра выбирается таким, чтобы размер карт сверточного слоя был четным, это позволяет не терять информацию при уменьшении размерности в подвыборочном слое, описанном ниже.   
 Ядро представляет собой систему разделяемых весов или синапсов, это одна из главных особенностей сверточной нейросети. В обычной многослойной сети очень много связей между нейронами, то есть синапсов, что весьма замедляет процесс детектирования. В сверточной сети – наоборот, общие веса позволяет сократить число связей и позволить находить один и тот же признак по всей области изображения.  
  
  
  
 Изначально значения каждой карты сверточного слоя равны 0. Значения весов ядер задаются случайным образом в области от -0.5 до 0.5. Ядро скользит по предыдущей карте и производит операцию свертка, которая часто используется для обработки изображений, формула:  
  
  
  
 Неформально эту операцию можно описать следующим образом — окном размера ядра g проходим с заданным шагом (обычно 1) все изображение f, на каждом шаге поэлементно умножаем содержимое окна на ядро g, результат суммируется и записывается в матрицу результата, как на рисунке 4.  
  
  
Рисунок 4 — Операция свертки и получение значений сверточной карты

Операция свертки и получение значений сверточной карты. Ядро смещено, новая карта получается того же размера, что и предыдущая   
 При этом в зависимости от метода обработки краев исходной матрицы результат может быть меньше исходного изображения, такого же размера или большего размера, в соответствии с рисунком 5.  
  
  
Рисунок 5 — Три вида свертки исходной матрицы  
 В упрощенном виде этот слой можно описать формулой:   
  
  
  
 При этом за счет краевых эффектов размер исходных матриц уменьшается, формула:  
  
  
 Подвыборочный слой также, как и сверточный имеет карты, но их количество совпадает с предыдущим (сверточным) слоем, их 6. Цель слоя – уменьшение размерности карт предыдущего слоя. Если на предыдущей операции свертки 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.   
 В процессе сканирования ядром подвыборочного слоя (фильтром) карты предыдущего слоя, сканирующее ядро не пересекается в отличие от сверточного слоя. Обычно, каждая карта имеет ядро размером 2x2, что позволяет уменьшить предыдущие карты сверточного слоя в 2 раза. Вся карта признаков разделяется на ячейки 2х2 элемента, из которых выбираются максимальные по значению.  
 Обычно в подвыборочном слое применяется функция активации RelU. Операция подвыборки (или MaxPooling – выбор максимального) в соответствии с рисунком 6.  
  
  
Рисунок 6 — Формирование новой карты подвыборочного слоя на основе предыдущей карты сверточного слоя. Операция подвыборки (Max Pooling)  
  
 Формально слой может быть описан формулой:   
  
  
 Последний из типов слоев это слой обычного многослойного персептрона. Цель слоя – классификация, моделирует сложную нелинейную функцию, оптимизируя которую, улучшается качество распознавания.  
  
   
 Нейроны каждой карты предыдущего подвыборочного слоя связаны с одним нейроном скрытого слоя. Таким образом число нейронов скрытого слоя равно числу карт подвыборочного слоя, но связи могут быть не обязательно такими, например, только часть нейронов какой-либо из карт подвыборочного слоя быть связана с первым нейроном скрытого слоя, а оставшаяся часть со вторым, либо все нейроны первой карты связаны с нейронами 1 и 2 скрытого слоя. Вычисление значений нейрона можно описать формулой:  
  


Выходной слой

Выходной слой связан со всеми нейронами предыдущего слоя. Количество нейронов соответствует количеству распознаваемых классов.

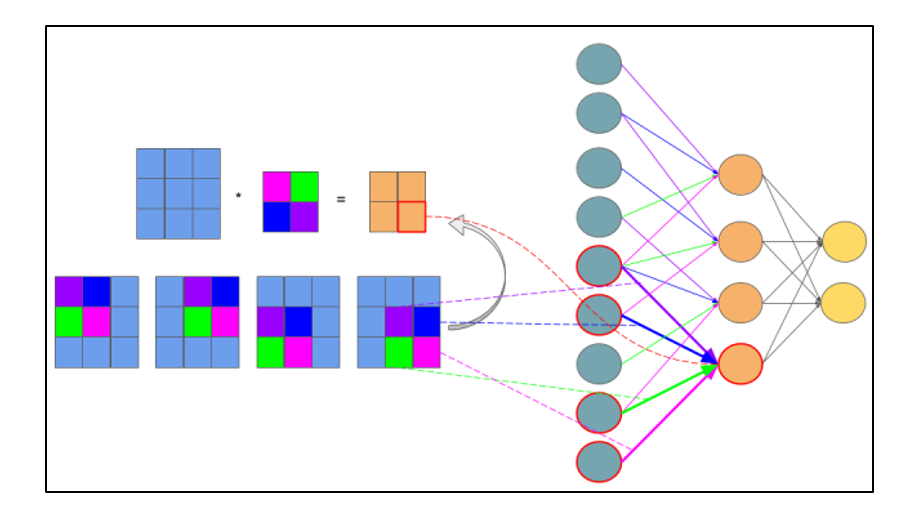
«Обучение сверточной нейросети»

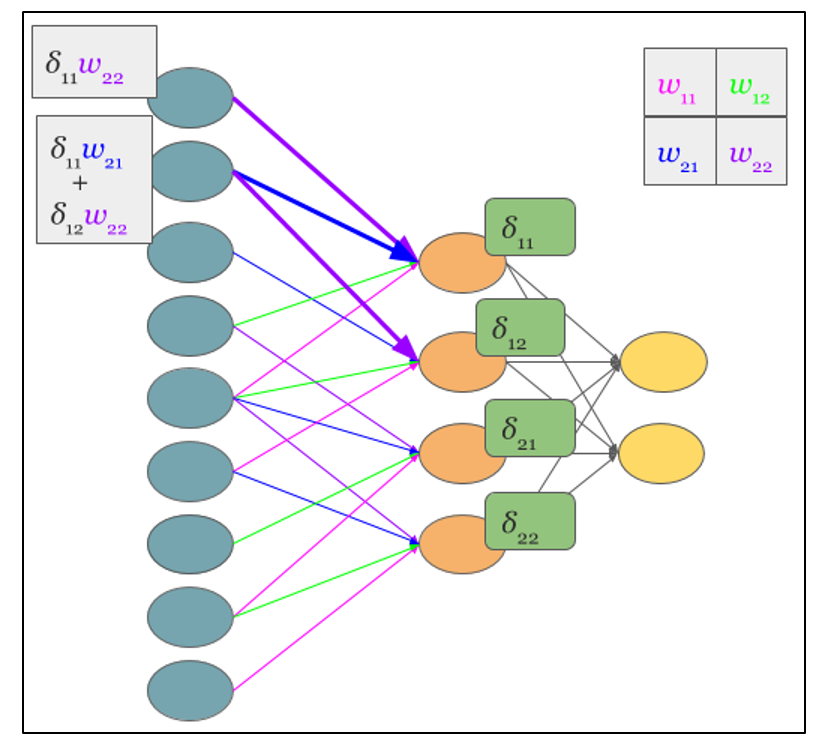
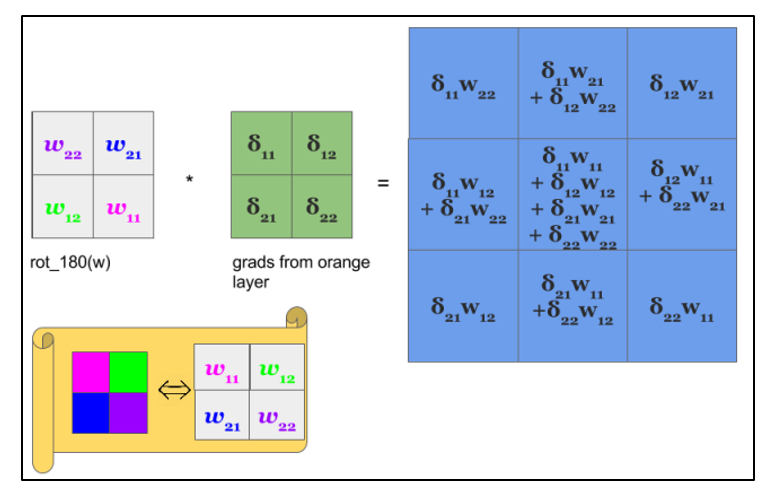
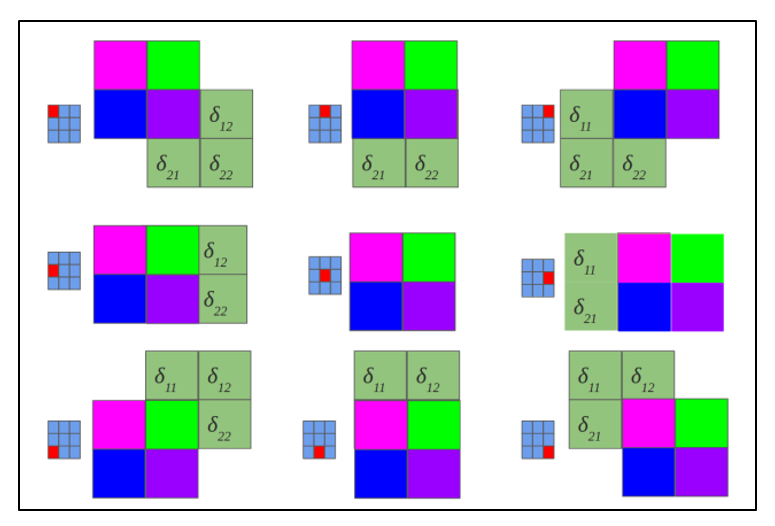
На начальном этапе нейронная сеть является необученной (ненастроенной). В общем смысле под обучением понимают последовательное предъявление образа на вход нейросети, из обучающего набора, затем полученный ответ сравнивается с желаемым выходом.

Затем эту дельту ошибки необходимо распространить на все связанные нейроны сети.

Таким образом обучение нейронной сети сводится к минимизации функции ошибки, путем корректировки весовых коэффициентов синаптических связей между нейронами. Под функцией ошибки понимается разность между полученным ответом и желаемым. Затем веса выходного слоя нейронов корректируются в соответствии с ошибкой. Для нейронов выходного слоя известны их фактические и желаемые значения выходов. Поэтому настройка весов связей для таких нейронов является относительно простой. Распространение ошибки на полно связные скрытые слоя производится с помощью метода обратного распространения ошибки.

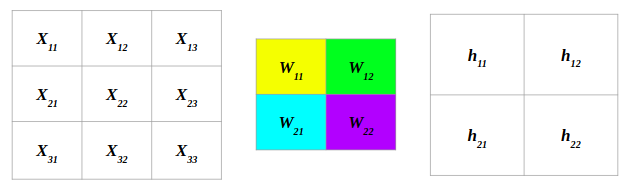
Расчет ошибки на подвыборочном слое

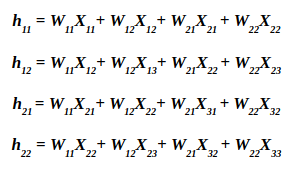
Расчет ошибки на подвыборочном слое представляется в нескольких вариантах. Первый случай, когда подвыборочный слой находится перед полносвязным, тогда он имеет нейроны и связи такого же типа, как в полносвязном слое, соответственно вычисление δ ошибки ничем не отличается от вычисления δ скрытого слоя. Второй случай, когда подвыборочный слой находится перед сверточным, вычисление δ происходит путем обратной свертки. Для понимания обратно свертки, необходимо сперва понять обычную свертку и то, что скользящее окно по карте признаков (во время прямого распространения сигнала) можно интерпретировать, как обычный скрытый слой со связями между нейронами, но главное отличие — это то, что эти связи разделяемы, то есть одна связь с конкретным значением веса может быть у нескольких пар нейронов, а не только одной. Интерпретация операции свертки в привычном многослойном виде в соответствии с рисунком 2.8.  
  
  
Рисунок 2.8 — Интерпретация операции свертки в многослойный вид, где связи с одинаковым цветом имеют один и тот же вес.

Синим цветом обозначена подвыборочная карта, разноцветным – синаптическое ядро, оранжевым – получившаяся свертка  
 Теперь, когда операция свертки представлена в привычном многослойном виде, можно интуитивно понять, что вычисление дельт происходит таким же образом, как и в скрытом слое полносвязной сети. Соответственно имея вычисленные ранее дельты сверточного слоя можно вычислить дельты подвыборочного, в соответствии с рисунком 2.9.  
  
Рисунок 2.9 — Вычисление δ подвыборочного слоя за счет δ сверточного слоя и ядра  
  
 Обратная свертка – это тот же самый способ вычисления дельт, только немного хитрым способом, заключающийся в повороте ядра на 180 градусов и скользящем процессе сканирования сверточной карты дельт с измененными краевыми эффектами. Простыми словами, нам необходимо взять ядро сверточной карты (следующего за подвыборочным слоем) повернуть его на 180 градусов и сделать обычную свертку по вычисленным ранее дельтам сверточной карты, но так чтобы окно сканирования выходило за пределы карты. Результат операции обратной свертки в соответствии с рисунком 2.10, цикл прохода обратной свертки в соответствии с рисунком 2.11.  
  
Рисунок 2.10 — Результат операции обратной свертки  
  
  
Рисунок 2.11 — Повернутое ядро на 180 градусов сканирует сверточную карту 

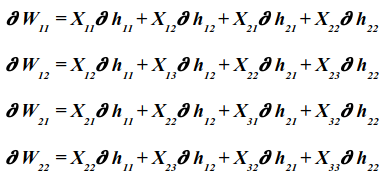
Расчет ошибки на сверточном слое

Обычно впередиидущий слой после сверточного это подвыборочный, соответственно наша задача вычислить дельты текущего слоя (сверточного) за счет знаний о дельтах подвыборочного слоя. На самом деле дельта ошибка не вычисляется, а копируется. При прямом распространении сигнала нейроны подвыборочного слоя формировались за счет неперекрывающегося окна сканирования по сверточному слою, в процессе которого выбирались нейроны с максимальным значением, при обратном распространении, мы возвращаем дельту ошибки тому ранее выбранному максимальному нейрону, остальные же получают нулевую дельту ошибки.

Применения градиентного спуска: 



Каждый вес в фильтре вносит свой вклад в каждый пиксель в выходной карте. Таким образом, любое изменение веса в фильтре повлияет на все выходные пиксели. Все эти изменения складываются, чтобы способствовать окончательной ошибке. Таким образом, мы можем легко вычислить производные следующим образом.



Соответственно, что бы вычислить конечную частную производную по корректируемым весам, нужно произвести операцию карты входов, картой ошибки текущего сверточного слоя. В результате получится матрица, того же размера что и матрица весов и каждая ее клетка будет соответствовать частной производной соответствующей клетке матрицы весов.

«Библиотека классов для работы с нейросетями»

Мной была реализована библиотека классов для работы с нейронными сетями двух типов: перцептроном и сверточной нейронной сети. Логика работы с библиотекой заключается в работе классов реализующих алгоритмы работы и обучения нейросетей со вспомогательными классами, представляющих данные в удобной форме. Все классы являются шаблонами. Это решение было обусловлено желанием сохранить гибкость библиотеки для работы как с дробными, как и с целочисленными данными.

Основным вспомогательным классом является класс Matrix.h представляющий собой динамический двухмерный контейнер. В нем реализованы все необходимые функции для работы с матрицей.

От него унаследовано два класса: Filter.h и Weights.h. Первый класс реализует функционал ядра свертки, а второй является абстракцией на веса искусственного нейрона.

Классы Base\_Perceptron.h и NeyronPerceptron.h содержат в себе функции необходимые для работы перцептрона. NeyronPerceptron.h унаследован от класса Base\_Perceptron.h с целью разбиения реализации функций нейросети на несколько классов. Если кто-то захочет изменить реализацию каких-либо функций но оставив нетронутой операцию суммирования, то ему нужно будет лишь наследовать класс Base\_Perceptron.h и переопределить метод FunckActiv (правда с трудом представляется такой вариант событий, наследование тут больше обусловлено моим желанием поработать с наследованием). В любом случае эти сложности скрыты от пользователя библиотеки.

Класс PerceptronLearning.h реализует алгоритмы обучения перцептрона (как однослойного так и многослойного).

В файле PLearns.h определены макросы объявления класса PerceptronLearning.h с разными типами шаблонных параметров.

В Файл Perceptrons.h подключены все нужные классы для работы с перцептроном и его обучения, а также определены уже готовые объявления класса NeyronPerceptron.h разными типами шаблонных параметров. По задумке они должны быть самыми используемыми и обычный пользователь сможет обойтись ими.

Классы Base\_Cnn.h и NeyronCnn.h содержат в себе функции необходимые для работы перцептрона. Они так же как классы для работы с перцептроном связаны отношением наследования. Оно обусловлено тут теми же самыми целями, что и в классах для работы с перцептроном.

Класс CNNLearning.h реализует алгоритмы обучения сверточной нейросети.

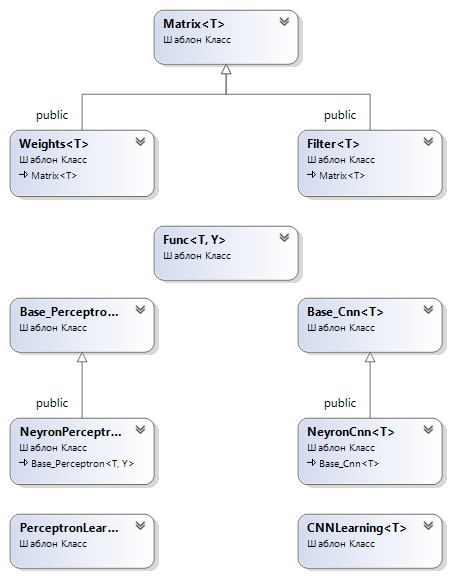
В файле CNNLearns.h определены макросы объявления класса CNNLearning.h с разными типами шаблонных параметров.

В Файл CNNs.h подключены все нужные классы для работы со сверточной нейросетью и её обучения, а также определены уже готовые объявления класса NeyronCnn.h разными типами шаблонных параметров.

Класс Func.h реализует абстрактный класс реализующий интерфейс функции активации. Он нужен для работы и обучения перцептрона. Что бы создать функтор, нужно унаследовать этот класс и переопределить оператор operator().

В файле Functors.h определены макросы объявления класса Func.h с разными типами шаблонных параметров.

Диаграмма классов:



Вся библиотека протестирована модульной системой тестирования взятой из книги “ Философия C++. Практическое программирование” Брюс Эккель, Чак Эллисон. Это помогло мне преждевременно найти множество ошибок, которые трудно обнаружить при обычном функционирование нейросети. Впрочем, от всех ошибок она меня не уберегла и я насобирал их довольно большое множество в процессе реализаций различных алгоритмов нейросетей. Одной из причин этого была не очень удачная в некоторых случаях реализация тестов.

Код библиотеки представлен в конце работы.

«Однослойный перцептрон»

Постановка задачи

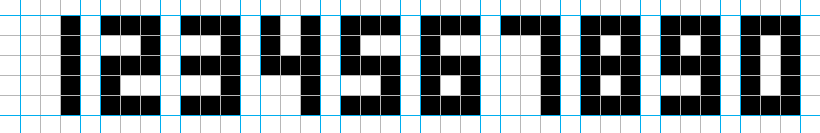
Целью данной программы являлась реализация самого простого типа нейросети: однослойного перцептрона с одним нейроном. Данной нейросети была поставлена задача распознавания одной цифры.

Картинка, как всем известно, состоит из пикселей. Так вот, очень удобно на каждый пиксель выделять один сенсор (для его распознавания).

Сейчас мы запрограммируем простейшую нейросеть, которая будет распознавать цифры. На самом деле это трудная задача, поэтому мы ее в очередной раз упростим:

1. Будем распознавать только черно-белые цифры от 0 до 9.

2. Цифры будут состоять из черных квадратиков в табличке 3х5 квадратов.

3. Распознать нейросеть должна будет научиться только одну цифру.

Например, пусть это будет цифра 4 (можно и любую другую). Только и всего.

Нашей обучающей выборкой будут все цифры от 0 до 9. Когда нейросеть обучиться безошибочно распознавать нужную нам цифру (5), тогда мы проверим ее «интеллект» уже на тестовой выборке. Она будет уже похитрее: на вход будут подаваться уже искаженные изображения четверки:

*1 0 1 1 0 1 1 0 1 0 0 1*

*0 0 1 1 0 0 1 0 1 1 0 0*

*1 1 1 1 1 1 1 1 0 1 1 1*

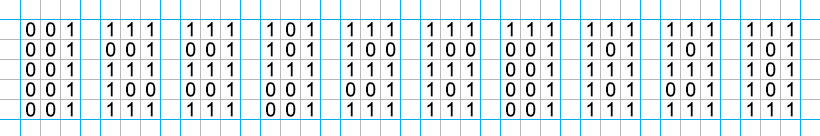
*0 0 1 0 0 0 0 0 1 0 0 0*

*0 0 1 0 0 1 0 0 1 0 0 1*

Цифры в матричном формате

Чтобы работать с нейросетью, мы должны на ее входы подавать сигналы в виде чисел (0 или 1). Таким образом изображение цифры мы должны перевести в последовательность сигналов в виде чисел. Это легко сделать, если представить цифры в строковом формате.

Каждая цифра представляет собой всего пятнадцать квадратиков, причем только двух возможных цветов. Как и говорилось в предыдущем разделе, за белый квадратик отвечает 0, а черный квадратик – 1. Поэтому наши десять цифр от 0 до 9 в матричном формате будут выглядеть вот так:



Для записи каждой цифры у нас используется по 5 матриц размером: 5х3.

*1 1 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1*

*1 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1 0 0 1 0 0 0 0 1 1 0 1 1 0 1*

*1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1*

*1 0 1 0 0 1 1 0 0 0 0 1 0 0 1 0 0 1 1 0 1 0 0 1 1 0 1 0 0 1*

*1 1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1*

Цифры в таком строковом формате уже можно использовать для работы с нейросетью.

Алгоритм обучения

В общих чертах процесс обучения понятен. Мы будем случайным образом выбирать цифру и прогонять ее через сеть, модифицируя ее веса. Но как их модифицировать?

Мы знаем, что важность тем или иным входам придают веса, связывающие их со скрытыми нейронами . Таким образом, чем сильнее повлиял какой-то вес связи на результат, тем сильнее надо его изменить.

Следовательно, мы должны учесть следующие важные моменты:

* Если наша нейросеть правильно распознала/отвергла цифру 4, то мы ничего не предпринимаем (все ведь замечательно!).
* Если нейросеть ошиблась и распознала неверную цифру как 4, то мы должны ее наказать – мы уменьшаем веса тех связей, через которые прошел сигнал. Другими словами веса, связанные с возбудившимися входами, уменьшаются.
* Если нейросеть ошиблась и не распознала цифру 4, то мы должны увеличить все веса, через которые прошел сигнал. Таким образом мы как бы говорим сети, что такие связи, а значит и связанные с ними входы – правильные.

Теперь запишем непосредственно алгоритм обучения, который мы будем реализовывать в программе:

1. Подать на входы нейросети цифру в матричном формате.
2. Если цифра распознана/отвергнута верно, то перейти к шагу 1.
3. Если сеть ошиблась и распознала неверную цифру как 4, то вычесть из всех связей, связанных с возбудившимися S-элементами единицу.
4. Если сеть ошиблась и отвергла цифру 4, то добавить единицу ко всем связям, связанным с возбудившимися S-элементами.

Обучающая выборка находится в текстовом файле: TeachChoose.txt.

Тестовая выборка находится в текстовом файле: Tests.txt.

После обучения веса нейросети сохраняются в файл: Weights.txt и в последствии могут быть извлечены от туда.

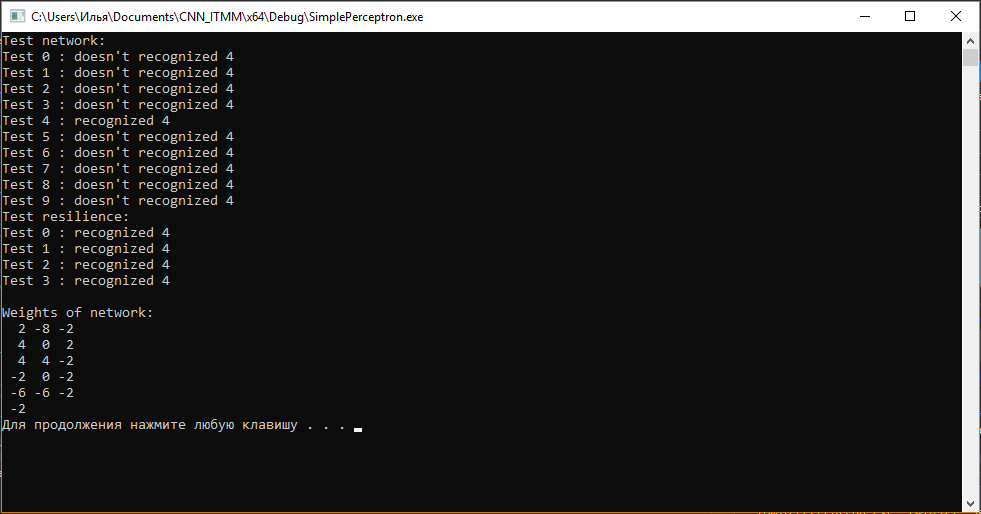
Программа поддерживает 2 режима работы: обучение + тест, считывание весов + тест, регулирующийся макросом Teach.

Программа 1

1. //: Нейросеть распознающая 4
3. #include "Perceptrons.h"
4. #include <vector>
5. #include <iostream>
6. #include <fstream>
8. // Макрос режима работы программы (с обучением или без)
9. //#define Teach
11. // функтор
12. class Sign : public II\_Func
13. {
15. public:
16. Sign() : II\_Func(){};
17. int operator()(const int& x) {
18. if (x <= 0) {
19. return -1;
20. }
21. else {
22. return 1;
23. }
24. }
25. ~Sign() {};
26. };
28. using namespace std;
29. int main()
30. {
31. // Создание перцептрона
32. II\_Perceptron Neyron;
34. // Создание обучателя сети
35. II\_Leaning Teacher;
37. // Создание функтора
38. Sign F;
40. // Создание весов нейрона
41. Weights<int> Weight(5,3);
42. #ifdef Teach
44. // Последовательность цифр, тасуемая для получения равномерной рандомизации
45. int nums[10] = { 0,1,2,3,4,5,6,7,8,9 };
47. // Создание обучающей выборки
48. vector<Matrix<int>> Nums(10);
50. // Создание обучающей выборки
51. // Считываем матрицы обучающей выборки
52. ifstream TeachChoose;
54. TeachChoose.open("TeachChoose.txt");
55. for (int i = 0; i < Nums.size(); i++) {
56. TeachChoose >> Nums[i];
57. }
58. TeachChoose.close();
60. // Обучение сети
61. long int k = 167; // Количество обучений нейросети
62. int summ; // Переменная суммы
63. int y; // Переменная выхода сети
65. for (long int i = 0; i < k; i++) {
66. Teacher.shuffle(nums, 10);
67. for (int j = 0; j < 10; j++) {
68. summ = Neyron.Summator(Nums[nums[j]], Weight); // Получение взвешенной суммы
69. y = Neyron.FunkActiv(summ, F); // Получение ответа нейрона
70. if (nums[j] != 4) {
71. Teacher.WTSimplePerceptron(-1, y, Weight, Nums[nums[j]]);
72. }
73. else {
74. Teacher.WTSimplePerceptron(1, y, Weight, Nums[nums[j]]);
75. }
76. }
77. }
79. // Сохраняем веса
80. ofstream fWeights;
81. fWeights.open("Weights.txt");
82. fWeights << Weight;
83. fWeights.close();
85. #else
86. // Считывание весов
87. ifstream fWeights;
88. fWeights.open("Weights.txt");
89. fWeights >> Weight;
90. fWeights.close();
92. #endif // Teach
93. // Создание тестовой выборки
94. vector<Matrix<int>> Tests(14);

97. // Считывание тестовой выборки из файла
98. ifstream Testsnums;
99. Testsnums.open("Tests.txt");
100. for (int i = 0; i < 14; i++) {
101. Testsnums >> Tests[i];
102. }
104. // Проверка работы сети
105. cout << "Test network:" << endl;
106. for (int i = 4; i < 14; i++) {
107. Neyron.FunkActiv(Neyron.Summator(Tests[i], Weight), F) == 1 ? cout << "Test " << i-4 << " : " << "recognized 4" << endl : cout << "Test " << i-4 << " : " << "doesn't recognized 4" << endl;
108. }
109. // Вывод на экран реультатов тестирования сети
110. cout << "Test resilience:" << endl;
111. for (int i = 0; i < 4; i++) {
112. Neyron.FunkActiv(Neyron.Summator(Tests[i], Weight), F) == 1 ? cout << "Test " << i << " : " << "recognized 4" << endl : cout << "Test " << i << " : " << "doesn't recognized 4" << endl;
113. }
115. // Вывод весов сети
116. cout << endl << "Weights of network: " << endl;
117. Weight.Out();
118. system("pause");
119. return 0;
120. }

Мои результаты: сеть обучается очень быстро и хорошо. Результат выдает правильный и стабильный. Количество эпох обучения и размер коэффициента скорости обучения влияет на обобщающие способности сети.



«Однослойный перцептрон (усовершенствование)»

Постановка задачи

Теперь, используя полученный опыт, напишем программу, которая распознает все числа. Для этого реализуем однослойный перцептрон с 10 нейронами.

Алгоритм обучения

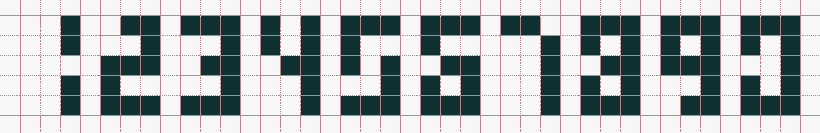
Мы будем последовательным образом прогонять цифру через сеть, модифицируя ее веса.

Следовательно, мы должны учесть следующие важные моменты:

* Если наша нейросеть правильно распознала/отвергла цифру n, то мы ничего не предпринимаем (все ведь замечательно!).
* Если нейросеть ошиблась и распознала цифру, но не ту которую мы ожидали, то мы должны ее наказать – мы уменьшаем веса для соответствующего числа тех связей, через которые прошел сигнал.
* Если нейросеть ошиблась и вообще не распознала цифру, то мы должны увеличить все веса для соответствующего числа, через которые прошел сигнал.

Теперь запишем непосредственно алгоритм обучения, который мы будем реализовывать в программе:

1. Подать на входы нейросети цифру в матричном формате.
2. Если цифра распознана/отвергнута верно, то перейти к шагу 1.
3. Если сеть ошиблась и распознала цифру, но не ту которую мы подавали на вход, то вычесть из всех связей соответствующей цифры, связанных с возбудившимися S-элементами единицу.
4. Если сеть ошиблась и не смогла ничего распознать, то добавить единицу ко всем связям, которые соответствуют входной цифре и связанным с возбудившимися S-элементами.

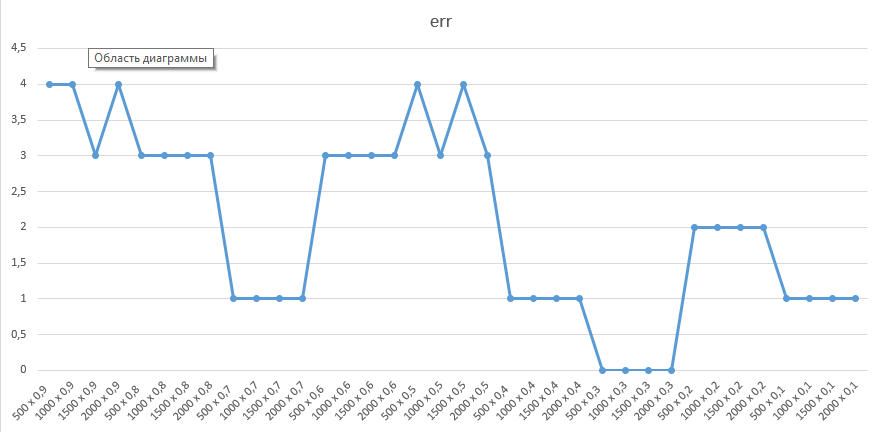
Нашей обучающей выборкой будут все те же цифры от 0 до 9. Для тестовой выборки используем искаженные изображения цифр.

Структура программы сохранилась прежней, только увеличилось количество нейронов (а соответственно и их весов).

Программа 2

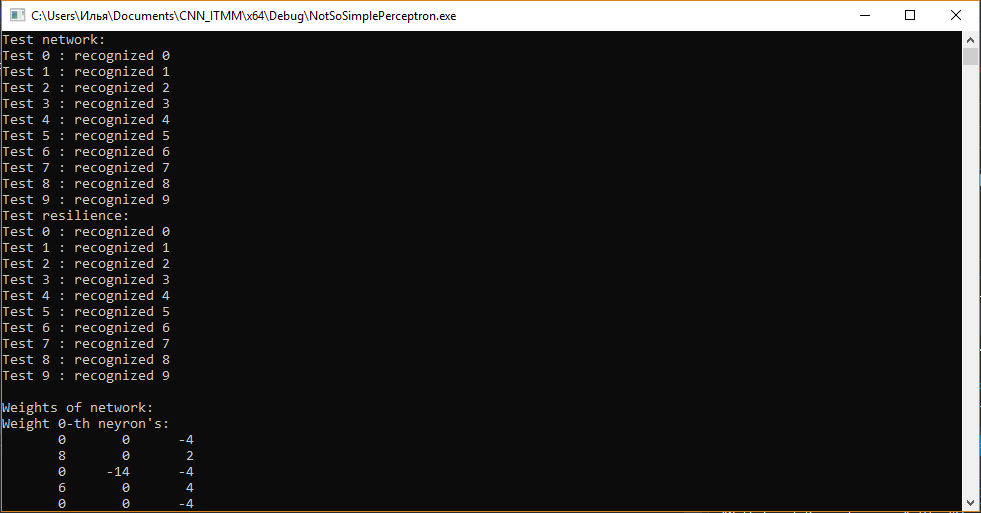
1. //: Нейросеть распознающая все цифры
3. #include "Perceptrons.h"
4. #include <vector>
5. #include <iostream>
6. #include <fstream>
8. // Макрос режима работы программы (с обучением или без)
9. //#define Teach
11. // функтор
12. class Sign : public DD\_Func
13. {
14. public:
15. Sign() : DD\_Func() {};
16. double operator()(const double& x) {
17. if (x <= 0) {
18. return -1;
19. }
20. else {
21. return 1;
22. }
23. }
24. ~Sign() {};
25. };
27. using namespace std;
28. int main()
29. {
30. // Создание перцептрона
31. DD\_Perceptron Neyron;
33. // Создание обучателя сети
34. DD\_Leaning Teacher;
35. Teacher.getE() = 1;
37. // Создание функтора
38. Sign F;
40. // Создание весов нейросети
41. vector<Weights<double>> W(10);
42. for (int i = 0; i < 10; i++) {
43. W[i] = Weights<double>(5, 3);
44. }
46. #ifdef Teach
48. // Последовательность цифр, тасуемая для получения равномерной рандомизации
49. int nums[10] = { 0,1,2,3,4,5,6,7,8,9 };
51. // Создание обучающей выборки
52. vector<Matrix<double>> Nums(10);
54. // Считываем матрицы обучающей выборки
55. ifstream TeachChoose;
56. TeachChoose.open("TeachChoose.txt");
57. for (int i = 0; i < Nums.size(); i++) {
58. TeachChoose >> Nums[i];
59. }
60. TeachChoose.close();

63. // Обучение сети
64. long int k = 167; // Количество обучений нейросети
65. double summ; // Переменная суммы
66. double y; // Переменная выхода сети
68. for (long int i = 0; i < k; i++) {
69. Teacher.shuffle(nums, 10); // Тасование последовательности
70. for (int j = 0; j < 10; j++) {
71. for (int l = 0; l < 10; l++) {
72. summ = Neyron.Summator(Nums[nums[j]], W[l]); // Получение взвешенной суммы
73. y = Neyron.FunkActiv(summ, F); // Получение ответа нейрона
74. if (nums[j] != l) { // Если текущии веса не совпадают с поданной на вход цифрой то ожидаемый ответ сети -1
75. Teacher.WTSimplePerceptron(-1, y, W[l], Nums[nums[j]]);
76. }
77. else { // Если совпадают, то ожидаемый ответ сети 1
78. Teacher.WTSimplePerceptron(1, y, W[l], Nums[nums[j]]);
79. }
80. }
81. }
82. }
84. // Сохраняем веса
85. ofstream fWeights;
86. fWeights.open("Weights.txt");
87. for (int i = 0; i < 10; i++) {
88. fWeights << W[i];
89. }
90. fWeights.close();
92. #else
93. // Считывание весов
94. ifstream fWeights;
95. fWeights.open("Weights.txt");
96. for (int i = 0; i < 10; i++) {
97. fWeights >> W[i];
98. }
99. fWeights.close();
100. #endif // Teach
102. // Создание тестовой выборки
103. vector<Matrix<double>> Tests(20);
105. // Считывание тестовой выборки из файла
106. ifstream Testsnums;
107. Testsnums.open("Tests.txt");
108. for (int i = 0; i < 20; i++) {
109. Testsnums >> Tests[i];
110. }
112. // Вывод на экран реультатов тестирования сети
113. cout << "Test network:" << endl;
114. for (int i = 0; i < 10; i++) {
115. for (int j = 0; j < 10; j++) {
116. if (Neyron.FunkActiv(Neyron.Summator(Tests[i], W[j]), F) == 1)
117. cout << "Test " << i << " : " << "recognized " << j << endl;
118. }
119. }
121. // Вывод на экран реультатов тестирования сети
122. cout << "Test resilience:" << endl;
123. for (int i = 10; i < 20; i++) {
124. for (int j = 0; j < 10; j++) {
125. if (Neyron.FunkActiv(Neyron.Summator(Tests[i], W[j]), F) == 1)
126. cout << "Test " << i-10 << " : " << "recognized " << j << endl;
127. }
128. }
130. // Вывод весов сети
131. cout << endl << "Weights of network: " << endl;
132. for (int i = 0; i < 10; i++) {
133. cout << "Weight " << i << "-th neyron's:" << endl;
134. W[i].Out();
135. cout << endl;
136. }
138. system("pause");
139. return 0;
141. }

Мои результаты: в данной сети используется функция активации единичного скачка. Для данной задачи она плохо подходит: не хватает гибкости. Простота алгоритма однослойной сети, плохо подходит для реализации такой, уже более сложной задачи. По этому что бы сеть начала распознавать тестовую выборку, пришлось некоторое время подбирать коэффициент скорости обучения сети и количества обучений:

Проведя небольшой эксперимент, можно построить график ошибок. Слева количество ошибок, внизу (кол-во обучений \* коэффициент скорости обучения). Можно заметить что при кофиценте 0.3 сеть обучается лучше всего. Но оказалось, что эксперимент в корне неверный, так как для полного обучения сети вполне достаточно 180 эпох обучения с коэффициентом скорости обучения 1. Эти цифры были получены опытным путем, причем, была замечена зависимость между скоростью обучения сети и количеством.

Наилучшие результаты сеть выдает когда отношение количества эпох обучения к коэффициенту скорости обучения равно примерно 167. Видимо при таком соотношении сеть не успевает переобучаться и сохраняет свои обобщающие способности.

Результаты: 

Я так же написал такую-же нейросеть, но уже с сигмоидальной функцией активации. Не буду приводить весь код (так как он одинаков) лишь приведу код функтора:

1. // функтор
2. class Sigm : public DD\_Func
3. {
4. public:
5. Sigm(const double& a\_) : DD\_Func(), a(a\_) {};
6. double a;
7. double operator()(const double& x) {
8. double f = 1;
9. const double e = 2.7182818284;
10. for (int i = 0; i < a\*x; i++)
11. {
12. f \*= 1 / e;
13. }
14. f++;
15. return 1 / f;
16. }
17. ~Sigm() {};
18. };

Мои результаты: Используя сигмоидальную функцию активации, можно добиться наибольшей гибкости в настройке нейросети. К тому же в сеть добавляется еще один параметр, влияющий на кривизну сигмоиды и соответственно влияющий на результат. Обучить такую сеть было немного проще. Но, несмотря не правильные ответы, нужно сказать, что тестовая выборка была немного подогнана под сеть. Из-за небольшого количества нейронов и слоев, сеть по сути выделяла ключевые пиксели в цифрах. Если убрать даже один такой пиксель, сеть перестанет распознавать цифру.

«Многослойный перцептрон»

Постановка задачи

На основе предыдущих сетей мною была реализована многослойная нейронная сеть. Задача, математические модели, обучающие и тестовые выборки остались прежними.

Так как задача по распознаванию всех цифр уже не является простейшей, для ее решения рационально применить более сложную топологию. Я использовал 2-х слойную нейронную сеть, так как по найденной мною информации увеличение слоев больше 3х в многослойном перцептроне не приводит к улучшению качества ответа сети. А так же я ограничен вычислительными возможностями своего компьютера (причем ограничен довольно сильно, т.к. я не использовал паралельное программирование и не проводил оптимизацию скорости выполнения своих алгоритмов).

Алгоритм обучения

Я использовал алгоритм обратного распространения ошибки. В качестве функции ошибки используется сумма квадратов разностей координат выходного вектора и ожидаемого вектора:

Где E – функция ошибки на конкретном примере

Y – Ответ нейронной сети (вектор)

A – Правильный выходной вектор

Методы реализующие обратное распространение ошибки и корректировку весов можно найти в коде библиотеки, представленной в приложении.

Программа 3

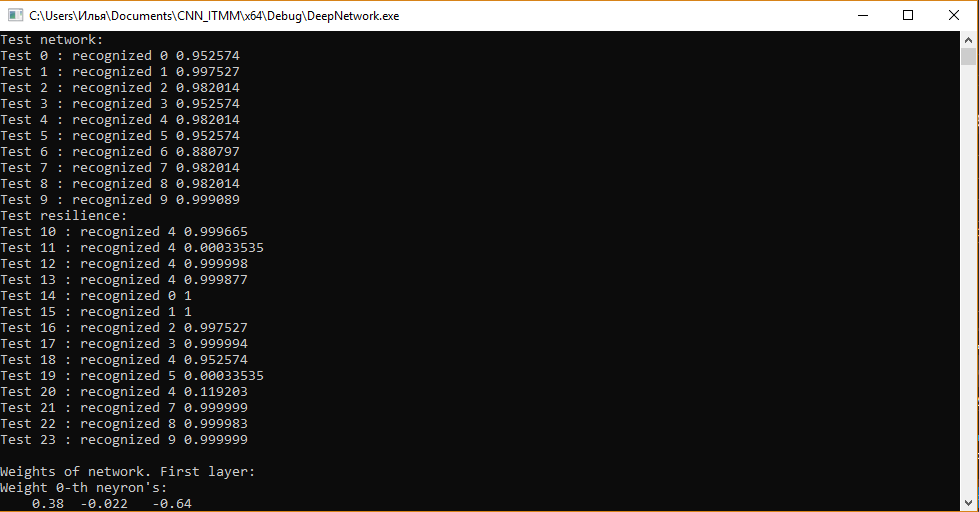
2. //: Нейросеть распознающая все цифры
4. #include "Perceptrons.h"
5. #include <vector>
6. #include <iostream>
7. #include <fstream>
9. // Макрос режима работы программы (с обучением или без)
10. #define Teach
11. #define OUT (X) std::cout << X;
12. #define NUMBER nums[j]
14. // функтор
15. // Сигмоида
16. class Sigm : public DD\_Func
17. {
18. public:
19. Sigm(const double& a\_) : DD\_Func(), a(a\_) {};
20. double a;
21. double operator()(const double& x) {
22. double f = 1;
23. const double e = 2.7182818284;
24. if (x >= 0) {
25. for (int i = 0; i < a\*x; i++)
26. {
27. f \*= 1 / e;
28. }
29. }
30. else {
31. for (int i = 0; i < abs(a\*x); i++)
32. {
33. f \*= e;
34. }
35. }
36. f++;
37. return 1 / f;
38. }
39. ~Sigm() {};
40. };
42. // Производная сигмоиды
43. class SigmD : public Sigm
44. {
45. public:
46. SigmD(const double& a\_) : Sigm(a\_) {};
47. double operator()(const double& x) {
48. double f = 1;
49. f = Sigm::operator()(x)\*(1 - Sigm::operator()(x));
50. return f;
51. }
52. ~SigmD() {};
53. };
55. using namespace std;
57. int main()
58. {
59. // Создание перцептрона
60. DD\_Perceptron Neyron;
62. // Создание обучателя сети
63. DD\_Leaning Teacher;
64. Teacher.getE() = 0.09;
66. // Создание функтора
67. Sigm F(3.4);
68. SigmD f(3.4);
70. // Установка зерна для выдачи рандомных значений
71. srand(time(0));
73. // Количество нейронов первого слоя нейросети
74. const int w1\_count = 100;
76. // Создание весов нейросети
77. Matrix<Weights<double>> W(1, w1\_count);
78. for (int i = 0; i < w1\_count; i++) {
79. W[0][i] = Weights<double>(5, 3);
80. for (int j = 0; j < W[0][i].getN(); j++) {
81. for (int p = 0; p < W[0][i].getM(); p++) {
82. W[0][i][j][p] = (p % 2 ? ((double)rand() / RAND\_MAX) : -((double)rand() / RAND\_MAX));
83. }
84. }
85. W[0][i].GetWBias() = (i % 2 ? ((double)rand() / RAND\_MAX) : -((double)rand() / RAND\_MAX));
86. }
88. // Создания весов для второго слоя сети
89. Matrix<Weights<double>> W1(1, 10);
90. for (int i = 0; i < 10; i++) {
91. W1[0][i] = Weights<double>(1, w1\_count);
92. for (int j = 0; j < W1[0][i].getN(); j++) {
93. for (int p = 0; p < W1[0][i].getM(); p++) {
94. W1[0][i][j][p] = (p % 2 ? ((double)rand() / RAND\_MAX) : -((double)rand() / RAND\_MAX));
95. }
96. }
97. W1[0][i].GetWBias() = (i % 2 ? ((double)rand() / RAND\_MAX) : -((double)rand() / RAND\_MAX));
98. }
100. // Массив выходов первого слоя сети
101. Matrix<double> m(1, w1\_count);
103. double summ; // Переменная суммы
104. double y[w1\_count]; // Переменная выхода сети
106. #ifdef Teach
108. // Последовательность цифр, тасуемая для получения равномерной рандомизации
109. int nums[10] = { 0,1,2,3,4,5,6,7,8,9 };
111. // Создание обучающей выборки
112. vector<Matrix<double>> Nums(10);
114. // Считываем матрицы обучающей выборки
115. ifstream TeachChoose;
116. TeachChoose.open("TeachChoose.txt");
117. for (int i = 0; i < Nums.size(); i++) {
118. TeachChoose >> Nums[i];
119. }
120. TeachChoose.close();
122. // Обучение сети
123. long int k = 70; // Количество обучений нейросети
125. for (long int i = 1; i < k; i++) {
126. Teacher.shuffle(nums, 10); // Тасование последовательности
127. cout << i << endl;
128. for (int j = 0; j < 10; j++) { // Цикл прохода по обучающей выборке
129. for (int u = 0; u < 3; u++) {
130. for (int l = 0; l < w1\_count; l++) { // Цикл прохода по сети
131. summ = Neyron.Summator(Nums[NUMBER], W[0][l]); // Получение взвешенной суммы
132. m[0][l] = Neyron.FunkActiv(summ, F);
133. }
134. for (int l = 0; l < 10; l++) { // Цикл прохода по сети
135. summ = Neyron.Summator(m, W1[0][l]); // Получение взвешенной суммы
136. y[l] = Neyron.FunkActiv(summ, F); // Запись выхода l-того нейрона в массив выходов сети
137. }
138. for (int l = 0; l < 10; l++) { // Расчет ошибки для выходного слоя
139. if (l == NUMBER) { // Если номер нейрона совпадает с поданной на вход цифрой, то ожидаеммый ответ 1
140. W1[0][l].GetD() = Teacher.PartDOutLay(1, y[l]); // Расчет ошибки
141. }
142. else {// Если номер нейрона совпадает с поданной на вход цифрой, то ожидаеммый ответ 1
143. W1[0][l].GetD() = Teacher.PartDOutLay(0, y[l]); // Расчет ошибки
144. }
145. }
147. for (int l = 0; l < 10; l++) { // Распространение ошибки на скрытые слои нейросети
148. Teacher.BackPropagation(W, W1[0][l]);
149. }
150. for (int l = 0; l < w1\_count; l++) { // Примемение градиентного спуска по всем нейроннам первого слоя
151. Teacher.GradDes(W[0][l], Nums[NUMBER], f, m[0][l]);
152. }
153. for (int l = 0; l < 10; l++) { // Примемение градиентного спуска по всем нейроннам второго слоя
154. summ = Neyron.Summator(m, W1[0][l]);
155. Teacher.GradDes(W1[0][l], m, f, summ);
156. }
157. for (int l = 0; l < w1\_count; l++) { // Обнуление ошибки нейронов 1 слоя
158. W[0][l].GetD() = 0;
159. }
160. Teacher.retract(W, 4);
161. Teacher.retract(W1, 4);
162. }
163. }
164. }
166. // Сохраняем веса
167. ofstream fWeights;
168. fWeights.open("Weights.txt");
169. fWeights << W;
170. fWeights << W1;
171. fWeights.close();
173. #else
174. // Считывание весов
175. ifstream fWeights;
176. fWeights.open("Weights.txt");
177. fWeights >> W;
178. fWeights >> W1;
179. fWeights.close();
180. #endif // Teach
182. // Создание тестовой выборки
183. vector<Matrix<double>> Tests(25);
185. // Считывание тестовой выборки из файла
186. ifstream Testsnums;
187. Testsnums.open("Tests.txt");
188. for (int i = 0; i < 24; i++) {
189. Testsnums >> Tests[i];
190. }
191. double max;
193. // Вывод на экран реультатов тестирования сети
194. cout << "Test network:" << endl;
195. for (int j = 0; j < 10; j++) { // Цикл прохода по тестовой выборке
196. for (int l = 0; l < w1\_count; l++) { // Цикл прохода по сети
197. summ = Neyron.Summator(Tests[j], W[0][l]); // Получение взвешенной суммы
198. m[0][l] = Neyron.FunkActiv(summ, F);
199. }
200. for (int l = 0; l < 10; l++) { // Цикл прохода по сети
201. summ = Neyron.Summator(m, W1[0][l]); // Получение взвешенной суммы
202. y[l] = Neyron.FunkActiv(summ, F); // Запись выхода l-того нейрона в массив выходов сети
203. }
204. int max = 0;
205. for (int l = 1; l < 10; l++) { // Получение результатов сети
206. if (y[l] > y[max]) {
207. max = l;
208. }
209. }
210. cout << "Test " << j << " : " << "recognized " << max << ' ' << y[max] << endl;
211. }
213. cout << "Test resilience:" << endl;
214. for (int j = 10; j < 24; j++) { // Цикл прохода по тестовой выборке
215. for (int l = 0; l < w1\_count; l++) { // Цикл прохода по сети
216. summ = Neyron.Summator(Tests[j], W[0][l]); // Получение взвешенной суммы
217. m[0][l] = Neyron.FunkActiv(summ, F);
218. }
219. for (int l = 0; l < 10; l++) { // Цикл прохода по сети
220. summ = Neyron.Summator(m, W1[0][l]); // Получение взвешенной суммы
221. y[l] = Neyron.FunkActiv(summ, F); // Запись выхода l-того нейрона в массив выходов сети
222. }
223. int max = 0;
224. for (int l = 1; l < 10; l++) { // Получение результатов сети
225. if (y[l] > y[max]) {
226. max = l;
227. }
228. }
229. cout << "Test " << j << " : " << "recognized " << max << ' ' << y[max] << endl;
230. }
232. // Вывод весов сети
233. cout << endl << "Weights of network. First layer: " << endl;
234. for (int i = 0; i < 10; i++) {
235. cout << "Weight " << i << "-th neyron's:" << endl;
236. W[0][i].Out();
237. cout << endl;
238. }
239. cout << endl << "Weights of network. First layer: " << endl;
240. for (int i = 0; i < 10; i++) {
241. cout << "Weight " << i << "-th neyron's:" << endl;
242. W1[0][i].Out();
243. cout << endl;
244. }
245. system("pause");
246. return 0;
248. }

Мои результаты: многослойная нейронная сеть намного превосходит однослойную. Сеть написанная и обученная мной имеет на данной задаче отличный обобщающий потенциал, отлично распознает тестовую выборку, устойчива к сильным шумам, искривлением и искажениям изображения.

К примеру:

Распознал как 4 Распознала как 9 Распознала 2

Результаты: 

Нужно уточнить, что в этот раз тестовая выборка немного изменилась и первые 4 цифры – видоизменная четверка.

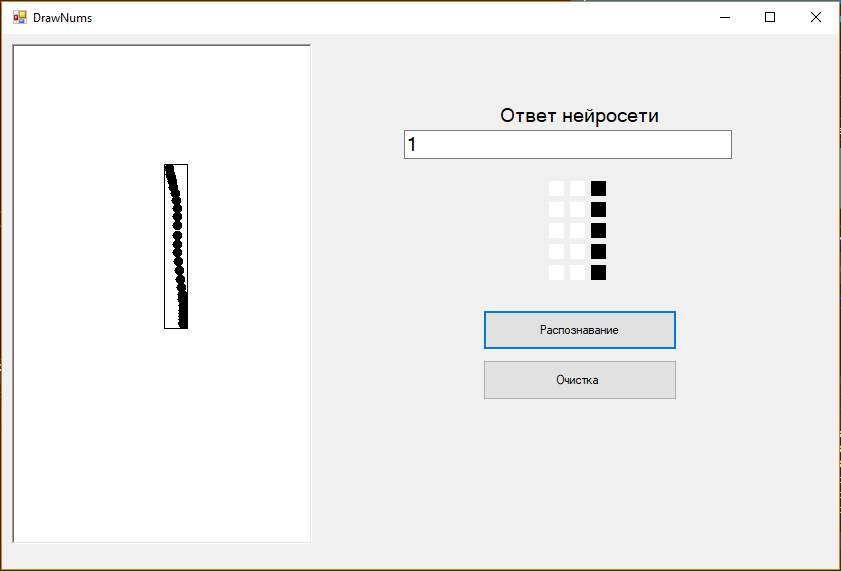
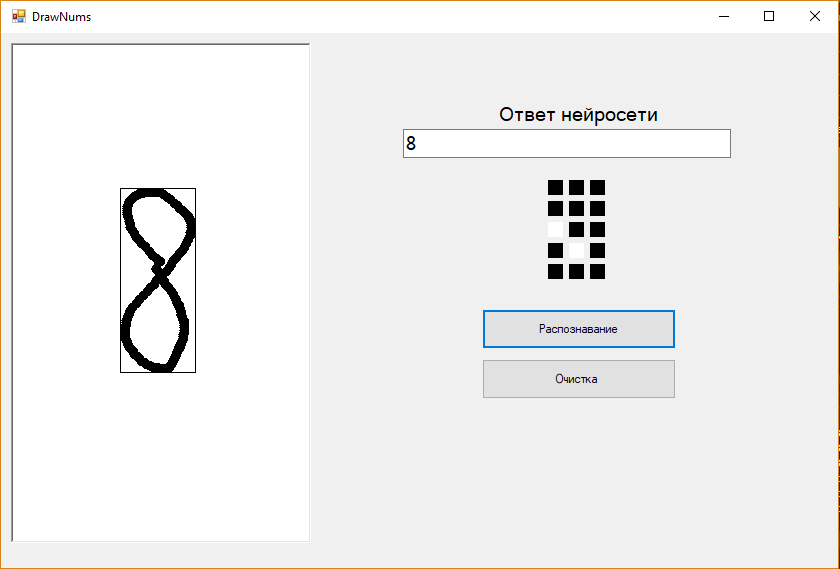
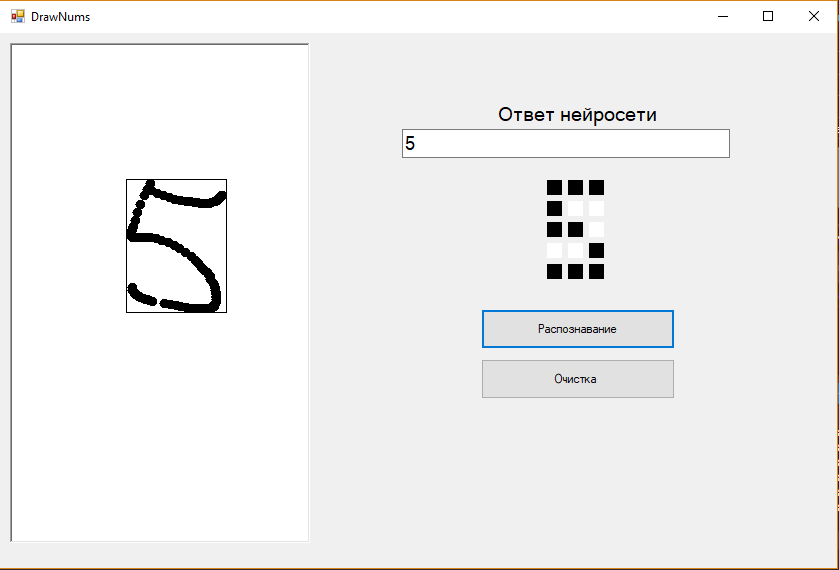
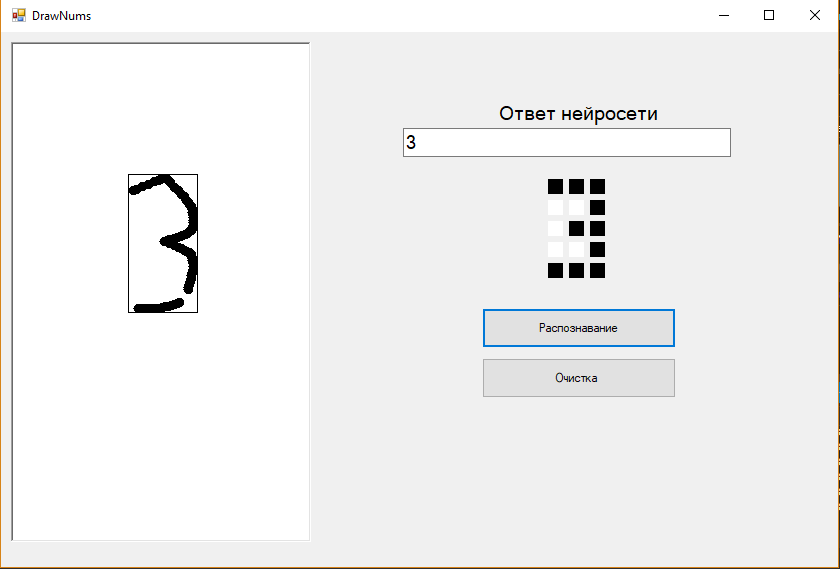
Хоть результаты очень хорошие, но обучить эту сеть было труднее чем предыдущие. Во-первых: проход по одной эпохе обучения занимает в несколько раз больше времени, чем обучение однослойной сети. Во вторых сеть сильно зависит от всех своих параметров: коэффициента скорости обучения, количества обучения и коэффициента кривизны сигмоиды. Их пришлось подбирать опытным путем, причем неверные параметры давали отнюдь не такой отличный результат. Кроме того, неверное выставление этих параметров влекло за собой увеличение весовых коэффициентов до огромных чисел, что в свою очередь влекло за собой длительное вычисление сигмоиды или вообще выход за границы максимального/минимального значения.

«Рукописный ввод цифр на базе многослойного перцептрона»

Постановка задачи

На основе пред идущих программ, попробуем найти применение нашей нейронной сети в распознавание рукописного ввода. То есть, ввод цифр осуществляется путем рисования мышкой изображения числа, преобразование его в массив пикселей, подача на вход в нейронную сеть и вывод результата. Конечно, для этого нам понадобится оформить примитивный пользовательский интерфейс и написать алгоритм преобразования изображения в массив пикселей. Программа была выполнена с помощью общеязыковой средой выполнения CLR, предоставляющей доступ к библиотеки классов .NET Framework.

Я решил не приводить текст программы из-за того что в нем нет ничего научно-ценного или интересного. Но в моем репозитории существует исходный код этой программы.

Для программы был написан класс, реализующий работу ранее обученной нейросети с теми же параметрами. Перед началом программы веса нейросети инициализируются «готовыми» весами из файла Weights.txt. Мои результаты:

Несмотря не очень хорошие результаты данная реализация имеет несколько ограничений:

* Сеть распознает только цифры определенного шрифта
* С единичкой пришлось немного схитрить, и добавлять отступ слева что бы получился нужный сети шрифт
* Обнаружение цифры на экране осуществлено простейшим поиском крайних пикселей, что работает только в данной упрощенной задаче, допускающей только 2 цвета пикселей.
* При рисовании необходимо сохранять «пропорции цифр»
* Сеть неустойчива к повороту цифр

«Сверточная нейросеть»

Постановка задачи

Целью данной программы являлась реализация сверточной нейронной сети на базе ранее написанной библиотеки классов. На этот раз усложним задачу: пусть сеть распознает все числа, но формат изображения будет 28\*28. Для этой задачи я использовал базу данных MNIST, имеющая подготовленный набор обучающих значений, в размере 60000 изображений для обучения и 10000 изображений для тестирования.

В своей работе я не использовал все множество изображений, ввиду ограничений вычислительных возможностей моего компьютера.

Для работы с изображениями была написана небольшая программа, переводящая пиксели из изображения в текстовый файл. Работа с изображениями происходит с помощью библиотеки OpenCV:

1. #include <fstream>
2. #include <iostream>
3. #include <string>
4. #include <opencv2/opencv.hpp>
5. #include <opencv2/core.hpp>
6. #include <opencv2/imgcodecs.hpp>
7. #include <opencv2/highgui.hpp>
8. #include <fstream>
10. using namespace cv;
11. using namespace std;
13. int main()
14. {
15. string folder;
16. Mat image;
17. string file;
18. string path;
19. for (int i = 0; i < 10; i++) {
20. ofstream out(to\_string(i)+".txt");
21. string folder = "E:\\CNN\_ITMM\\mnist\_png\\training\\" + to\_string(i) + "\\";
22. for (int j = 1; j < 1001; j++) {
23. file = " (" + to\_string(j) + ").png";
24. path = folder + file;
25. image = imread(path, IMREAD\_GRAYSCALE);
26. out << image.rows << ' ' << image.cols << endl;
27. for (int i = 0; i < image.rows; i++) {
28. for (int j = 0; j < image.cols; j++) {
29. if ((int)image.at<uchar>(i, j) == 0) {
30. out << -1 << ' ';
31. }
32. else {
33. out << 1 << ' ';
34. }
35. }
36. out << endl;
37. }
38. }
39. }
40. return 0;
41. }

В реализованной мной программе настройка топологии и параметров сети происходит в начале программы путем выставления константных коэффициентов.

В процессе работы с сетью было принято решение при больших значениях аргумента округлять функцию активации что бы ускорить процесс обучения.

Реализуя сеть, я перебрал множество топологий с различными параметрами. К сожалению, я был сильно ограничен вычислительными возможностями своего компьютера и по этому, при увеличении топологии скорость работы сети сильно замедлялась. Обучить сеть с такой скоростью обучения очень тяжело. В итоге я остановился на небольшой, но достаточно емкой сети, скорость работы которой была в пределах разумного.

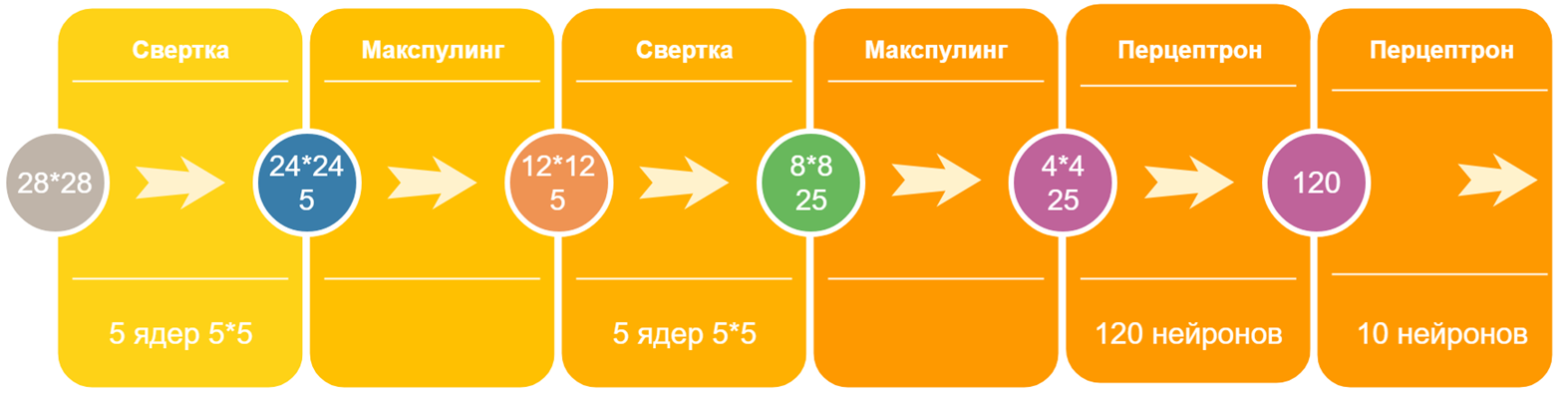
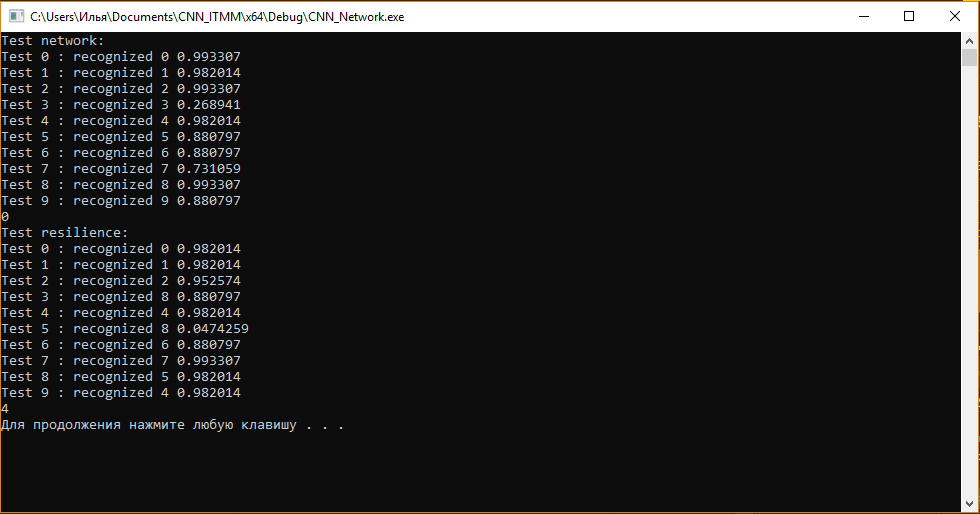


Схема выбранной мной топологии

Обучая сеть, я воспользовался способностью глубинных нейросетей доучиваться. Сначала я подобрал параметры сети, при которых она хорошо обучалась распознавать все числа с одним шрифтом. Конечно при этом обобщающие способности сети были удручающими. После этого, инициализировав веса уже переобученными значениями, уменьшив в 2 раза коэффициент скорости обучения и немного сгладив производную, я обучил сеть на 100 разных шрифтах каждой цифры. После этого ошибки нейросети на обучающей выборки значительно снизились, но на тестовой выборке сеть все еще выдавала ошибку в 60%. Снова уменьшив коэффициент скорости обучения в 2 раза я доучил сеть на 1000 различных шрифтов для каждой цифры. Обучение заняло примерно день.

## Программа 4



Мои результаты: после 3х циклов обучения сеть делает 10% ошибок на обучающей выборке и 22-25% ошибок на тестовой выборке. Я провел тесты на 40 разных шрифтах: в обучающей выборке сеть сделала 30 ошибок из 400 цифр, в тестовой выборке сеть сделала 66 ошибок из 400 цифр.

Нужно заметить, что сеть в силу небольшой топологии имеет недостаточную емкость для данной задачи. Из-за этого некоторые цифры сеть распознает почти без ошибок (например 0 и 1) а некоторые (9 и 5) распознает плохо. К примеру при тестировании сеть сделала всего пару ошибок при распознавании 0 и 1, а при распознавании 9 сеть ошибается почти в 50% случаев. Видимо, данной топологии не хватает для выделения всех особенностей цифр. Возможно так же что сеть нужно обучать большее количество раз, постепенно уменьшая коэффициенты.

Программа 4

1. //: Нейросеть распознающая все цифры
2. #include <Windows.h>
3. #include "Perceptrons.h"
4. #include "CNNs.h"
5. #include <vector>
6. #include <iostream>
7. #include <fstream>
9. using namespace std;
10. // Макрос режима работы программы (с обучением или без)
12. #define Teach
14. // Улучшение читабильности программы
15. #define NUMBER nums[j]
17. // функтор
18. // Сигмоида
19. class Sigm : public DD\_Func
20. {
21. public:
22. Sigm(const double& a\_) : DD\_Func(), a(a\_) {};
23. double a;
24. double operator()(const double& x) {
25. double f = 1;
26. const double e = 2.7182818284;
27. if (x >= 0) {
28. if (x > 10000) {
29. return 0.99999999;
30. }
31. for (int i = 0; i < a\*x; i++)
32. {
33. f \*= 1 / e;
34. }
35. }
36. else {
37. if (x < -10000) {
38. return 0.00000001;
39. }
40. for (int i = 0; i < abs(a\*x); i++)
41. {
42. f \*= e;
43. }
44. }
45. f++;
46. return 1 / f;
47. }
48. ~Sigm() {};
49. };
51. // Производная сигмоиды
52. class SigmD : public Sigm
53. {
54. public:
55. SigmD(const double& a\_) : Sigm(a\_) {};
56. double operator()(const double& x) {
57. double f = 1;
58. f = Sigm::operator()(x)\*(1 - Sigm::operator()(x));
59. return f;
60. }
61. ~SigmD() {};
62. };
64. using namespace std;
66. int main()
67. {
68. // Создание перцептрона
69. DD\_Perceptron Neyron;
71. // Создание обучателя сети
72. DD\_Leaning Teacher;
73. Teacher.getE() = 0.00064;
75. // Создание CNN
76. D\_NeyronCnn NeyronCNN;
78. // Создание обучателя CNN сети
79. D\_CNNLeaning TeacherCNN;
80. TeacherCNN.getE() = 0.0000006;
82. // Создание функтора
83. Sigm F(0.8);
85. // Производная функтора
86. SigmD f(0.8);
88. // Установка зерна для выдачи рандомных значений
89. srand(time(0));
91. // Размер входной матрицы
92. const int image\_width = 28;
93. const int image\_height = 28;
95. // Размер фильтров (ядер свертки)
96. const int filter\_width = 5;
97. const int filter\_height = 5;
98. const int filter1\_width = 5;
99. const int filter1\_height = 5;
101. // Размер матрицы нейронов
102. const int neyron\_width = 100;
103. const int neyron\_height = 4;
104. const int neyron1\_width = 120;
105. const int neyron1\_height = 1;
107. // Количество фильтров
108. const int f1\_count = 5;
109. const int k = 5;
110. const int f2\_count = f1\_count \* k;
112. // Количество нейронов
113. const int w1\_count = 120;
114. const int w2\_count = 10;
116. // Кофицент создания весов
117. const int decade = 1;
119. // Создание весов фильтров первого слоя
120. vector<Filter<double>> FILTERS(f1\_count);
121. for (int i = 0; i < f1\_count; i++) {
122. FILTERS[i] = Filter<double>(filter\_height, filter\_width);
123. for (int j = 0; j < FILTERS[i].getN(); j++) {
124. for (int p = 0; p < FILTERS[i].getM(); p++) {
125. FILTERS[i][j][p] = (p % 2 ? ((double)rand() / (RAND\_MAX\*decade)) : -((double)rand() / (RAND\_MAX \* decade)));
126. }
127. }
128. }
130. // Создание весов фильтров второго слоя
131. vector<Filter<double>> FILTERS1(f2\_count);
132. for (int i = 0; i < f2\_count; i++) {
133. FILTERS1[i] = Filter<double>(filter1\_height, filter1\_width);
134. for (int j = 0; j < FILTERS1[i].getN(); j++) {
135. for (int p = 0; p < FILTERS1[i].getM(); p++) {
136. FILTERS1[i][j][p] = (p % 2 ? ((double)rand() / (RAND\_MAX \* decade)) : -((double)rand() / (RAND\_MAX \* decade)));
137. }
138. }
139. }
141. // Создание весов перового слоя перцептрона
142. Matrix<Weights<double>> WEIGHTS(1, w1\_count);
143. for (int i = 0; i < w1\_count; i++) {
144. WEIGHTS[0][i] = Weights<double>(neyron\_height, neyron\_width);
145. for (int j = 0; j < WEIGHTS[0][i].getN(); j++) {
146. for (int p = 0; p < WEIGHTS[0][i].getM(); p++) {
147. WEIGHTS[0][i][j][p] = (p % 2 ? ((double)rand() / (RAND\_MAX \* decade)) : -((double)rand() / (RAND\_MAX \* decade)));
148. }
149. }
150. WEIGHTS[0][i].GetWBias() = (i % 2 ? ((double)rand() / (RAND\_MAX \* decade)) : -((double)rand() / (RAND\_MAX \* decade)));
151. }
153. // Создания весов для второго слоя перцептрона
154. Matrix<Weights<double>> WEIGHTS1(1, w2\_count);
155. for (int i = 0; i < w2\_count; i++) {
156. WEIGHTS1[0][i] = Weights<double>(neyron1\_height, neyron1\_width);
157. for (int j = 0; j < WEIGHTS1[0][i].getN(); j++) {
158. for (int p = 0; p < WEIGHTS1[0][i].getM(); p++) {
159. WEIGHTS1[0][i][j][p] = (p % 2 ? ((double)rand() / (RAND\_MAX \* decade)) : -((double)rand() / (RAND\_MAX \* decade)));
160. }
161. }
162. WEIGHTS1[0][i].GetWBias() = (i % 2 ? ((double)rand() / (RAND\_MAX \* decade)) : -((double)rand() / (RAND\_MAX \* decade)));
163. }
165. // Матрица выхода сети
166. Matrix<double> MATRIX\_OUT(1, w1\_count);
168. double summ; // Переменная суммы
169. double y[w2\_count]; // Переменная выхода сети
171. // Матрицы изображений
172. // Матрица входного изображения
173. Matrix<double> IMAGE\_1(image\_height, image\_width);
174. // Вектор матриц изображений после первого сверточного слоя
175. vector< Matrix<double>> IMAGE\_2(f1\_count);
176. // Вектор матриц изображений после первого подвыборочного слоя
177. vector< Matrix<double>> IMAGE\_3(f1\_count);
178. // Вектор матриц изображений после второго сверточного слоя
179. vector< Matrix<double>> IMAGE\_4(f2\_count);
180. // Вектор матриц изображений после второго подвыборочного слоя
181. vector< Matrix<double>> IMAGE\_5(f2\_count);
183. // Вектор, передающийся в перцептрон (состоит из всех карт последнего подвыборочного слоя)
184. Matrix<double> IMAGE\_OUT(neyron\_height, neyron\_width);
186. // Переменная максимума
187. int max = 0;
189. #ifdef Teach
191. // Матрицы ошибок сверточной сети
192. // Вектор матриц ошибок первого сверточного слоя
193. vector< Matrix<double>> IMAGE\_2\_D(f1\_count);
194. // Вектор матриц ошибок первого подвыборочного слоя
195. vector< Matrix<double>> IMAGE\_3\_D(f1\_count);
196. // Вектор матриц ошибок второго сверточного слоя
197. vector< Matrix<double>> IMAGE\_4\_D(f2\_count);
198. // Вектор матриц ошибок второго подвыборочного слоя
199. vector< Matrix<double>> IMAGE\_5\_D(f2\_count);
201. // Матрица ошибки выхода изображения
202. Matrix<double> IMAGE\_OUT\_D(neyron\_height, neyron\_width);
203. IMAGE\_OUT.Fill(0);
205. // Последовательность цифр, тасуемая для получения равномерной рандомизации
206. // Может как использоваться или не использоваться
207. int nums[10] = { 0,1,2,3,4,5,6,7,8,9 };
209. long int koll = 1000; // Количество обучений нейросети (по совместительству количество разных шрифтов)
211. // Создание обучающей выборки
212. vector<vector<Matrix<double>>> Nums(10);
213. for (int i = 0; i < 10; i++) {
214. Nums[i] = vector<Matrix<double>>(koll);
215. }
217. // Считывание весов
218. // Опционально, используется для обучения
219. //ifstream fWeightss;
220. //fWeightss.open("Weights.txt");
221. //for (int i = 0; i < f1\_count; i++) {
222. //  fWeightss >> FILTERS[i];
223. //}
224. //for (int i = 0; i < f2\_count; i++) {
225. //  fWeightss >> FILTERS1[i];
226. //}
227. //fWeightss >> WEIGHTS;
228. //fWeightss >> WEIGHTS1;
229. //fWeightss.close();
231. // Массив, нужный для подсчета ошибки
232. double a[10];
234. // Считывание обучающей выборки
235. string folder = "..\\Image\_to\_txt\\";
236. string file;
237. string path;
238. ifstream input;
239. for (int i = 0; i < 10; i++) {
240. file = to\_string(i) + ".txt";
241. path = folder + file;
242. input.open(path);
243. for (int j = 0; j < koll; j++) {
244. input >> Nums[i][j];
245. }
246. input.close();
247. }
249. // Обучение сети
250. for (long int i = 0; i < koll; i++) {
251. //Teacher.shuffle(nums, 10); // Тасование последовательности
252. for (int j = 0; j < 10; j++) { // Цикл прохода по обучающей выборке
253. for (int u = 0; u < 3; u++) { // Количество проходов по одной цифре
254. // Работа сети
255. // Обнуление переменной максимума
256. max = 0;
257. // Считывание картика поданной на вход сети
258. IMAGE\_1 = Nums[NUMBER][i];
259. // Проход картинки через первый сверточный слой
260. for (int l = 0; l < f1\_count; l++) {
261. IMAGE\_2[l] = NeyronCNN.Svertka(FILTERS[l], IMAGE\_1);
262. }
263. // Операция макспулинга
264. for (int l = 0; l < f1\_count; l++) {
265. IMAGE\_3[l] = NeyronCNN.Pooling(IMAGE\_2[l], 2, 2);
266. }
267. // Проход картинки через второй сверточный слой
268. for (int l = 0; l < f1\_count; l++) {
269. for (int ll = 0; ll < k; ll++) {
270. IMAGE\_4[l\*k + ll] = NeyronCNN.Svertka(FILTERS1[l\*k + ll], IMAGE\_3[l]);
271. }
272. }
273. // Операция макспулинга
274. for (int l = 0; l < f2\_count; l++) {
275. IMAGE\_5[l] = NeyronCNN.Pooling(IMAGE\_4[l], 2, 2);
276. }
277. for (int l = 0; l < f2\_count; l++) {
278. for (int li = 0; li < 4; li++) {
279. for (int lj = 0; lj < 4; lj++) {
280. IMAGE\_OUT[li][l \* 4 + lj] = IMAGE\_5[l][li][lj];
281. }
282. }
283. }
284. // Проход по перцептрону
285. // Проход по первому слою
286. for (int l = 0; l < w1\_count; l++) { // Цикл прохода по сети
287. summ = Neyron.Summator(IMAGE\_OUT, WEIGHTS[0][l]); // Получение взвешенной суммы
288. MATRIX\_OUT[0][l] = Neyron.FunkActiv(summ, F);
289. }
290. for (int l = 0; l < w2\_count; l++) { // Цикл прохода по сети
291. summ = Neyron.Summator(MATRIX\_OUT, WEIGHTS1[0][l]); // Получение взвешенной суммы
292. y[l] = Neyron.FunkActiv(summ, F); // Запись выхода l-того нейрона в массив выходов сети
293. }
294. for (int l = 1; l < w2\_count; l++) { // Получение результатов сети
295. if (y[l] > y[max]) {
296. max = l;
297. }
298. }
299. // Вывод распознанной цифры на экран для визуализации процесса обучения
300. cout << max << ' ';
301. // Расчет ошибки
302. for (int i = 0; i < w2\_count; i++) {
303. if (i == NUMBER)
304. a[i] = 1;
305. if (i != NUMBER)
306. a[i] = 0;
307. }
308. // Вывод ошибки на экран
309. cout << Teacher.RMS\_error(a, y, w2\_count) << endl;
310. // Если ошибка мала, пропускаем цикл обучения, что бы избежать переобучения сети
311. if (Teacher.RMS\_error(a, y, w2\_count) < 0.3) {
312. continue;
313. }
314. // Обучение сети
315. for (int l = 0; l < w2\_count; l++) { // Расчет ошибки для выходного слоя
316. if (l == NUMBER) { // Если номер нейрона совпадает с поданной на вход цифрой, то ожидаеммый ответ 1
317. WEIGHTS1[0][l].GetD() = Teacher.PartDOutLay(1, y[l]); // Расчет ошибки
318. }
319. else {// Если номер нейрона совпадает с поданной на вход цифрой, то ожидаеммый ответ 1
320. WEIGHTS1[0][l].GetD() = Teacher.PartDOutLay(0, y[l]); // Расчет ошибки
321. }
322. }
323. // Распространение ошибки на скрытые слои перцептрона
324. for (int l = 0; l < w2\_count; l++) {
325. Teacher.BackPropagation(WEIGHTS, WEIGHTS1[0][l]);
326. }
327. // Распространение ошибки на выход картинки
328. for (int l = 0; l < w2\_count; l++) {
329. TeacherCNN.Revers\_Perceptron\_to\_CNN(IMAGE\_OUT\_D, WEIGHTS[0][l]);
330. }
331. // Копирование ошибки на подвыборочный слой
332. for (int l = 0; l < f2\_count; l++) {
333. for (int li = 0; li < 4; li++) {
334. for (int lj = 0; lj < 4; lj++) {
335. IMAGE\_5\_D[l][li][lj] = IMAGE\_OUT\_D[li][l \* 4 + lj];
336. }
337. }
338. }
339. // Распространение ошибки на сверточный слой
340. for (int l = 0; l < f2\_count; l++) {
341. IMAGE\_4\_D[l] = TeacherCNN.ReversPooling(IMAGE\_5\_D[l], 2, 2);
342. }
343. // Распространение ошибки на подвыборочный слой
344. for (int l = 0; l < f1\_count; l++) {
345. IMAGE\_3\_D[l] = TeacherCNN.ReversConvolution(IMAGE\_4\_D[l\*k], FILTERS1[l\*k]);
346. for (int ll = 1; ll < k; ll++) {
347. IMAGE\_3\_D[l] = IMAGE\_3\_D[l] + TeacherCNN.ReversConvolution(IMAGE\_4\_D[l\*k + ll], FILTERS1[l\*k + ll]);
348. }
349. }
350. // Распространение ошибки на сверточный слой
351. for (int l = 0; l < f1\_count; l++) {
352. IMAGE\_2\_D[l] = TeacherCNN.ReversPooling(IMAGE\_3\_D[l], 2, 2);
353. }
354. // Примемение градиентного спуска
355. // Первый сверточный слой
356. for (int l = 0; l < f1\_count; l++) {
357. TeacherCNN.GradDes(IMAGE\_1, IMAGE\_2\_D[l], FILTERS[l]);
358. }
359. // Второй сверточный слой
360. for (int l = 0; l < f1\_count; l++) {
361. for (int ll = 0; ll < k; ll++) {
362. TeacherCNN.GradDes(IMAGE\_3[l], IMAGE\_4\_D[l\*k + ll], FILTERS1[l\*k + ll]);
363. }
364. }
365. // Перцептрон
366. // Первый слой
367. for (int l = 0; l < w1\_count; l++) { // Примемение градиентного спуска по всем нейроннам первого слоя
368. Teacher.GradDes(WEIGHTS[0][l], IMAGE\_OUT, f, MATRIX\_OUT[0][l]);
369. }
370. // Второй слой
371. for (int l = 0; l < w2\_count; l++) { // Примемение градиентного спуска по всем нейроннам второго слоя
372. summ = Neyron.Summator(MATRIX\_OUT, WEIGHTS1[0][l]);
373. Teacher.GradDes(WEIGHTS1[0][l], MATRIX\_OUT, f, summ);
374. }
375. // Обнуление ошибок
376. for (int l = 0; l < w1\_count; l++) { // Обнуление ошибки нейронов 1 слоя
377. WEIGHTS[0][l].GetD() = 0;
378. }
379. // Обнуления вектора ошибок
380. IMAGE\_OUT\_D.Fill(0);
381. // "Замедление обучения сети"
382. Teacher.getE() -= Teacher.getE() \* 0.00001;
383. TeacherCNN.getE() -= TeacherCNN.getE() \* 0.00001;
384. }
385. }
386. }
388. // Сохранение весов
389. ofstream fWeights;
390. fWeights.open("Weights.txt");
391. for (int i = 0; i < f1\_count; i++) {
392. fWeights << FILTERS[i];
393. }
394. for (int i = 0; i < f2\_count; i++) {
395. fWeights << FILTERS1[i];
396. }
397. fWeights << WEIGHTS;
398. fWeights << WEIGHTS1;
399. fWeights.close();
401. #else
402. //Считывание весов
403. ifstream fWeights;
404. fWeights.open("Weights.txt");
405. for (int i = 0; i < f1\_count; i++) {
406. fWeights >> FILTERS[i];
407. }
408. for (int i = 0; i < f2\_count; i++) {
409. fWeights >> FILTERS1[i];
410. }
411. fWeights >> WEIGHTS;
412. fWeights >> WEIGHTS1;
413. fWeights.close();
414. string folder;
415. string file;
416. string path;
418. #endif // Teach
420. // Создание тестовой выборки
421. vector<vector<Matrix<double>>> TestNums(10);
422. for (int i = 0; i < 10; i++) {
423. TestNums[i] = vector<Matrix<double>>(100);
424. }
425. // Считывание тестовой выборки
426. folder = "..\\Image\_to\_txt\\";
427. for (int i = 0; i < 10; i++) {
428. file = to\_string(i) + ".txt";
429. path = folder + file;
430. ifstream inputt(path);
431. for (int j = 0; j < 30; j++) {
432. inputt >> TestNums[i][j];
433. }
434. inputt.close();
435. }
436. // Переменная ошибок сети
437. int errors\_network = 0;
438. // Вывод на экран реультатов тестирования сети
439. cout << "Test network:" << endl;
440. for (int i = 0; i < 10; i++) { // Цикл прохода по тестовой выборке
441. for (int j = 0; j < 30; j++) {
442. int max = 0;
443. // Работа сети
444. // Считывание картика поданной на вход сети
445. IMAGE\_1 = TestNums[i][j];
446. // Проход картинки через первый сверточный слой
447. for (int l = 0; l < f1\_count; l++) {
448. IMAGE\_2[l] = NeyronCNN.Svertka(FILTERS[l], IMAGE\_1);
449. }
450. // Операция макспулинга
451. for (int l = 0; l < f1\_count; l++) {
452. IMAGE\_3[l] = NeyronCNN.Pooling(IMAGE\_2[l], 2, 2);
453. }
454. // Проход картинки через второй сверточный слой
455. for (int l = 0; l < f1\_count; l++) {
456. for (int ll = 0; ll < k; ll++) {
457. IMAGE\_4[l\*k + ll] = NeyronCNN.Svertka(FILTERS1[l\*k + ll], IMAGE\_3[l]);
458. }
459. }
460. // Операция макспулинга
461. for (int l = 0; l < f2\_count; l++) {
462. IMAGE\_5[l] = NeyronCNN.Pooling(IMAGE\_4[l], 2, 2);
463. }
464. for (int l = 0; l < f2\_count; l++) {
465. for (int li = 0; li < 4; li++) {
466. for (int lj = 0; lj < 4; lj++) {
467. IMAGE\_OUT[li][l \* 4 + lj] = IMAGE\_5[l][li][lj];
468. }
469. }
470. }
471. // Проход по перцептрону
472. // Проход по первому слою
473. for (int l = 0; l < w1\_count; l++) { // Цикл прохода по сети
474. summ = Neyron.Summator(IMAGE\_OUT, WEIGHTS[0][l]); // Получение взвешенной суммы
475. MATRIX\_OUT[0][l] = Neyron.FunkActiv(summ, F);
476. }
477. for (int l = 0; l < w2\_count; l++) { // Цикл прохода по сети
478. summ = Neyron.Summator(MATRIX\_OUT, WEIGHTS1[0][l]); // Получение взвешенной суммы
479. y[l] = Neyron.FunkActiv(summ, F); // Запись выхода l-того нейрона в массив выходов сети
480. }
481. for (int l = 1; l < w2\_count; l++) { // Получение результатов сети
482. if (y[l] > y[max]) {
483. max = l;
484. }
485. }
486. // Вывод результатов на экран
487. cout << "Test " << i << " : " << "recognized " << max << ' ' << y[max] << endl;
488. // Подсчет ошибок
489. if (max != i) {
490. errors\_network++;
491. }
492. }
493. }
494. // Вывод количества ошибок на экран
495. cout << errors\_network << endl;
497. // Считывание тестовой выборки
498. folder = "..\\Image\_to\_txt\\";
499. for (int i = 0; i < 10; i++) {
500. file = to\_string(i) + "\_tests.txt";
501. path = folder + file;
502. ifstream inputt(path);
503. for (int j = 0; j < 40; j++) {
504. inputt >> TestNums[i][j];
505. }
506. inputt.close();
507. }
508. // Переменная количества ошибок на тестовой выборке
509. int errors\_resilience = 0;
510. // Вывод на экран реультатов тестирования сети
511. cout << "Test resilience:" << endl;
512. for (int i = 0; i < 10; i++) { // Цикл прохода по тестовой выборке
513. for (int j = 0; j < 40; j++) {
514. max = 0;
515. // Работа сети
516. // Считывание картика поданной на вход сети
517. IMAGE\_1 = TestNums[i][j];
518. // Проход картинки через первый сверточный слой
519. for (int l = 0; l < f1\_count; l++) {
520. IMAGE\_2[l] = NeyronCNN.Svertka(FILTERS[l], IMAGE\_1);
521. }
522. // Операция макспулинга
523. for (int l = 0; l < f1\_count; l++) {
524. IMAGE\_3[l] = NeyronCNN.Pooling(IMAGE\_2[l], 2, 2);
525. }
526. // Проход картинки через второй сверточный слой
527. for (int l = 0; l < f1\_count; l++) {
528. for (int ll = 0; ll < k; ll++) {
529. IMAGE\_4[l\*k + ll] = NeyronCNN.Svertka(FILTERS1[l\*k + ll], IMAGE\_3[l]);
530. }
531. }
532. // Операция макспулинга
533. for (int l = 0; l < f2\_count; l++) {
534. IMAGE\_5[l] = NeyronCNN.Pooling(IMAGE\_4[l], 2, 2);
535. }
536. for (int l = 0; l < f2\_count; l++) {
537. for (int li = 0; li < 4; li++) {
538. for (int lj = 0; lj < 4; lj++) {
539. IMAGE\_OUT[li][l \* 4 + lj] = IMAGE\_5[l][li][lj];
540. }
541. }
542. }
543. // Проход по перцептрону
544. // Проход по первому слою
545. for (int l = 0; l < w1\_count; l++) { // Цикл прохода по сети
546. summ = Neyron.Summator(IMAGE\_OUT, WEIGHTS[0][l]); // Получение взвешенной суммы
547. MATRIX\_OUT[0][l] = Neyron.FunkActiv(summ, F);
548. }
549. for (int l = 0; l < w2\_count; l++) { // Цикл прохода по сети
550. summ = Neyron.Summator(MATRIX\_OUT, WEIGHTS1[0][l]); // Получение взвешенной суммы
551. y[l] = Neyron.FunkActiv(summ, F); // Запись выхода l-того нейрона в массив выходов сети
552. }
553. for (int l = 1; l < w2\_count; l++) { // Получение результатов сети
554. if (y[l] > y[max]) {
555. max = l;
556. }
557. }
558. // Вывод результатов на экран
559. cout << "Test " << i << " : " << "recognized " << max << ' ' << y[max] << endl;
560. // Подсчет ошибок
561. if (max != i) {
562. errors\_resilience++;
563. }
565. }
566. }
567. // Вывод на экран реультатов тестирования сети
568. cout << errors\_resilience << endl;
569. system("pause");
570. return 0;
572. }

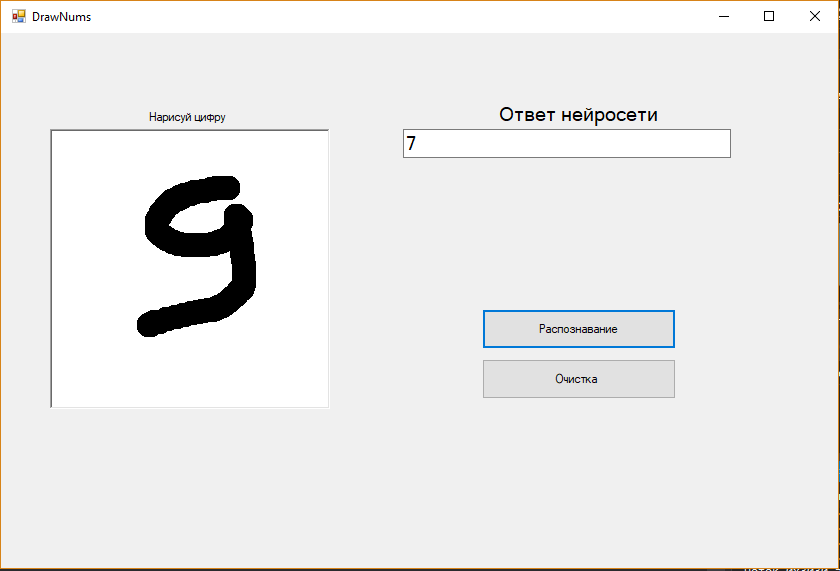
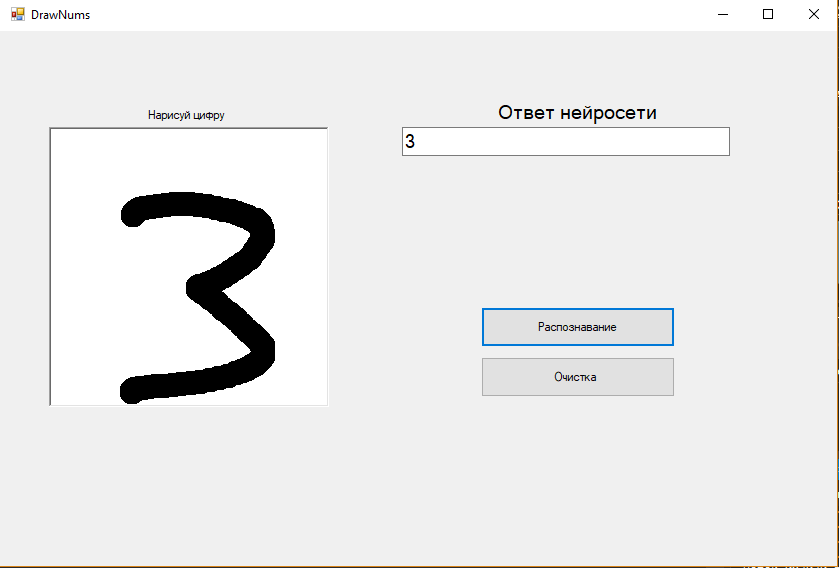
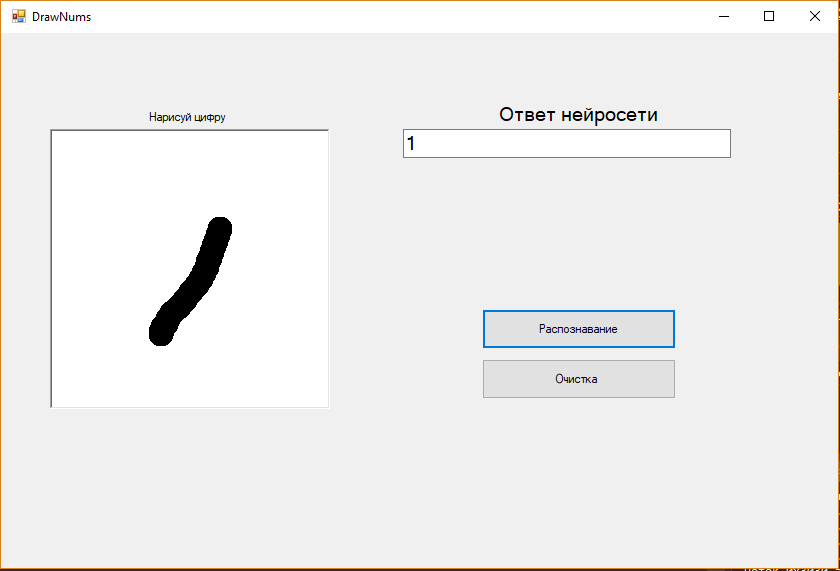
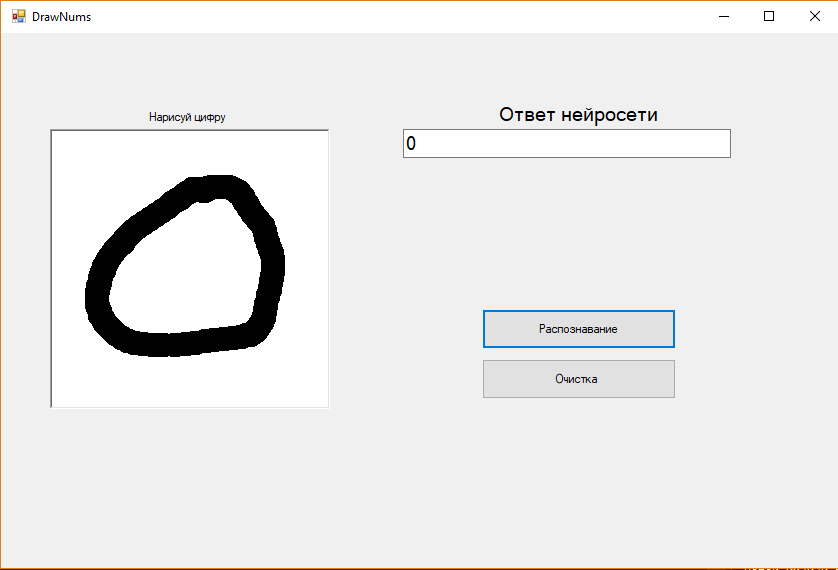
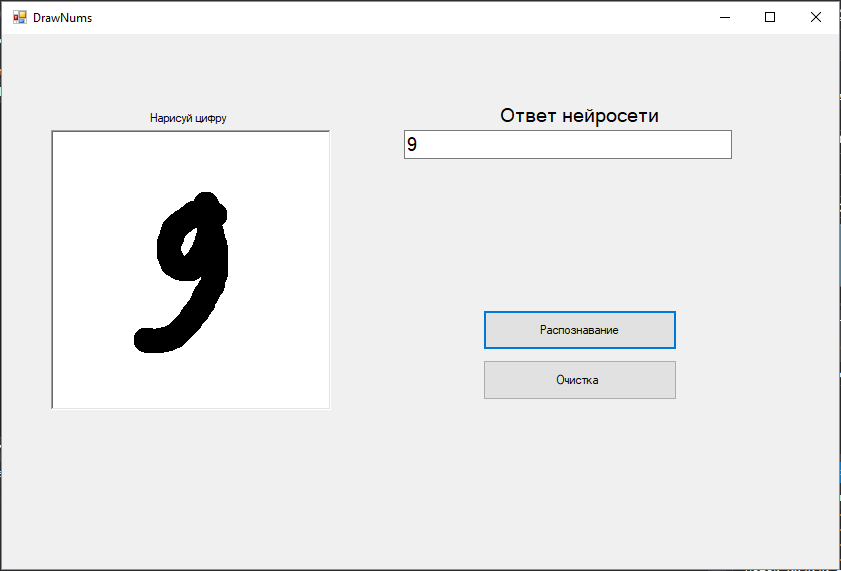
Рукописный ввод цифр на базе сверточной нейросети»

Постановка задачи

На основе сверточной сети я решил написать аналогичную программу рукописного ввода цифр. Конечно, для этого я так же написал примитивный пользовательский интерфейс и алгоритм преобразования изображения в матрицу пикселей и запись ее в текстовый файл. Программа была выполнена с помощью общеязыковой средой выполнения CLR, предоставляющей доступ к библиотеки классов .NET Framework.

Я решил не приводить текст программы из-за того что в нем нет ничего научно-ценного или интересного. Но в моем репозитории существует исходный код этой программы.

Для программы был написан класс, реализующий работу ранее обученной нейросети с теми же параметрами. Перед началом программы веса нейросети инициализируются «готовыми» весами из файла Weights.txt.

Мои результаты:

Как видно из примеров, сеть устойчива к небольшому изменению масштаба и небольшому повороту цифр. Сеть с переменным успехом распознает цифры с измененными пропорциями.

«Заключение»

Нейронные сети являются очень полезным алгоритмом решения определенного рода задач. Причем, количество таких задач с каждым годом увеличивается.

В своей работе я рассмотрел несколько видов нейросетей, для того, чтобы по ближе ознакомиться с основами этих алгоритмов и принципами их работы.

Конечно, в реальной жизни существует множество готовых «Фреймворков» для работы с нейросетями. Но я считаю что подробное изучение основ этих алгоритмов поможет мне разобраться с работой более сложных сетей и упростит разработку существующих.

Я намереваюсь продолжить изучение более сложных нейросетей, разных типов а также попробовать их разработку с помощью готовых решений.

«Исходный код библиотеки для работы с нейросетями»

Сверточная нейронная сеть

Base\_Cnn.h

1. #pragma once
2. #include "Matrix.h"
3. #include "Filter.h"
5. template<typename T>
6. class Base\_Cnn
7. {
8. public:
9. // Конструкторы ----------------------------------------------------------
10. Base\_Cnn();
11. Base\_Cnn(const Base\_Cnn& copy) = delete; // Запрет копирования
13. // Методы класса ---------------------------------------------------------
14. // Добавление "полей" к матрице
15. virtual void Padding(Matrix<T>&);
17. // Операция "Макс пулинга"
18. virtual Matrix<T> Pooling(const Matrix<T>&, const int&, const int&);
20. // Операция свертки над матрицей значений
21. virtual Matrix<T> Svertka(const Matrix<T>&, const Matrix<T>&) = 0;
23. // Перегрузка операторов -------------------------------------------------
24. Base\_Cnn<T>& operator= (const Base\_Cnn<T>& copy) = delete; // Запрет копирования
26. // Деструктор ------------------------------------------------------------
27. virtual ~Base\_Cnn();
29. // Класс исключения ------------------------------------------------------
30. class Base\_CnnExeption : public std::runtime\_error {
31. public:
32. Base\_CnnExeption(std::string str) : std::runtime\_error(str) {};
33. ~Base\_CnnExeption() {};
34. };
35. };
37. template<typename T>
38. Base\_Cnn<T>::Base\_Cnn()
39. {
40. }
42. template<typename T>
43. Base\_Cnn<T>::~Base\_Cnn()
44. {
45. }
47. template<typename T>
48. Matrix<T> Base\_Cnn<T>::Pooling(const Matrix<T>& a, const int& n\_, const int& m\_)
49. {
50. if ((n\_ < 0) || (m\_ < 0) || (n\_ > a.getN()) || (m\_ > a.getM())) {
51. throw Base\_Cnn<T>::Base\_CnnExeption("Неверный размер ядра!");
52. }
54. Matrix<T> copy(a.getN() / n\_, a.getM() / m\_);
56. for (int i = 0; i < copy.getN(); i++) {
57. for (int j = 0; j < copy.getM(); j++) {
58. copy[i][j] = a.getPodmatrix(i\*n\_, j\*m\_, n\_, m\_).Max();
59. }
60. }
61. return copy;
62. }

65. template<typename T>
66. void Base\_Cnn<T>::Padding(Matrix<T>& a)
67. {
68. Matrix<T> copy(a.getN() + 2, a.getM() + 2);
70. for (int i = 0; i < copy.getN(); i++) {
71. for (int j = 0; j < copy.getM(); j++) {
72. if ((i == 0) || (j == 0) || (j == (copy.getM() - 1)) || (i == (copy.getN() - 1))) {
73. copy[i][j] = 0;
74. }
75. else {
76. copy[i][j] = a[i - 1][j - 1];
77. }
78. }
79. }
81. a = copy;
82. }

NeyronCnn.h

1. #pragma once
2. #include "Base\_Cnn.h"
3. #include <string>

6. template<typename T>
7. class NeyronСnn : public Base\_Cnn<T>
8. {
9. public:
10. // Конструкторы ----------------------------------------------------------
11. NeyronСnn();
12. explicit NeyronСnn(const int& step\_ );
13. NeyronСnn(const NeyronСnn<T>& copy) = delete; // Запрет копирования
15. // Методы класса ---------------------------------------------------------
16. // Операция свертки над матрицей значений
17. Matrix<T> Svertka(const Matrix<T>& F, const Matrix<T>& a);
19. // Получение доступа к шагу свертки
20. int& GetStep() { return step; }
22. // Перегрузка операторов -------------------------------------------------
23. NeyronСnn& operator= (const NeyronСnn<T>& copy) = delete; // Запрет копирования
25. // Деструктор ------------------------------------------------------------
26. ~NeyronСnn();
28. // Класс исключения ------------------------------------------------------
29. class NeyronСnnExeption : public Base\_Cnn<T>::Base\_CnnExeption {
30. public:
31. NeyronСnnExeption(std::string str) : Base\_Cnn<T>::Base\_CnnExeption(str) {};
32. ~NeyronСnnExeption() {};
33. };
34. private:
35. // Поля класса -----------------------------------------------------------
36. int step; // Шаг свертки
37. };

40. template<typename T>
41. NeyronСnn<T>::NeyronСnn() : Base\_Cnn<T>(), step(1)
42. {
43. }
45. template<typename T>
46. NeyronСnn<T>::NeyronСnn(const int& step\_) : Base\_Cnn<T>(), step(step\_)
47. {
48. }

51. template<typename T>
52. Matrix<T> NeyronСnn<T>::Svertka(const Matrix<T>& F, const Matrix<T>& a)
53. {
55. if ((step > a.getN()) || (step > a.getM())||(step < 1)) {
56. throw NeyronСnnExeption("Задан невозможный шаг свертки!");
57. }
59. Matrix<T> rez((a.getN() - F.getN()) / step + 1, (a.getM() - F.getM()) / step + 1);
61. double sum;
62. Matrix<T> fokus;
63. for (int i = 0; i < rez.getN(); i++) {
64. for (int j = 0; j < rez.getM(); j++) {
65. sum = 0;
66. fokus = a.getPodmatrix(i\*step, j\*step, F.getN(), F.getM());
67. for (int ii = 0; ii < F.getN(); ii++) {
68. for (int jj = 0; jj < F.getM(); jj++) {
69. sum += fokus[ii][jj] \* F[ii][jj];
70. }
71. }
72. rez[i][j] = sum;
73. }
74. }
75. return rez;
76. }

79. template<typename T>
80. NeyronСnn<T>::~NeyronСnn()
81. {
82. }

CNNLearning.h

1. #pragma once
2. #include "Weights.h"
3. #include "Filter.h"
4. #include "NeyronCnn.h"
5. #include <cstdlib>
7. template<typename T>
8. class CNNLearning
9. {
10. public:
11. // Конструкторы ----------------------------------------------------------
12. CNNLearning(const int& s\_ = 1, const double& E\_ = 1);
14. // Методы класса ---------------------------------------------------------
15. // Метод обратного распространения ошибки
16. Matrix<T> ReversConvolution(const Matrix<T>& D, const Filter<T>& f);
18. // Метод градиентного спуска
19. void GradDes(const Matrix<T>& X, const Matrix<T>& D, Filter<T>& F);
21. // Получение доступа к шагу свертки
22. int& getStep() { return s; }
24. // Метод получения доступа к кофиценту обучения
25. double& getE() { return E; };
27. // Операция обратного распространение ошибки c перцептрона на подвыборочный слой
28. void Revers\_Perceptron\_to\_CNN(Matrix<T>& a, const Weights<T>& w);
30. // Операция обратного распространение ошибки на слое "Макс пулинга"
31. Matrix<T> ReversPooling(const Matrix<T>& a, const int& n\_, const int& m\_);
33. // Класс исключения ------------------------------------------------------
34. class CNNLearningExeption : public std::runtime\_error {
35. public:
36. CNNLearningExeption(std::string str) : std::runtime\_error(str) {};
37. ~CNNLearningExeption() {};
38. };
40. // Деструктор ------------------------------------------------------------
41. ~CNNLearning();
42. private:
43. NeyronСnn<T> neyron;
44. int s;
45. double E;
46. };
48. template<typename T>
49. inline CNNLearning<T>::CNNLearning(const int& s\_, const double& E\_):neyron(), s(s\_), E(E\_)
50. {
51. }
53. template<typename T>
54. inline Matrix<T> CNNLearning<T>::ReversConvolution(const Matrix<T>& D, const Filter<T>& f)
55. {
56. if (s < 1) {
57. throw CNNLearning<T>::CNNLearningExeption("Задан невозможный шаг свертки!");
58. }
59. auto F = f.roate\_180();
60. Matrix<T> O((D.getN() - 1) / s + f.getN(), (D.getM() - 1) / s + f.getM());
61. if (s != 1) {
62. int stepJ = 0, stepI = 0;
63. int ii = 0, jj = 0;
64. for (int i = 0; i < O.getN(); i++) {
65. stepJ = 0;
66. jj = 0;
67. if (stepI) {
68. for (int j = 0; j < O.getM(); j++) {
69. O[i][j] = 0;
70. }
71. stepI--;
72. }
73. else {
74. for (int j = 0; j < O.getM(); j++) {
75. if (stepJ) {
76. stepJ--;
77. O[i][j] = 0;
78. }
79. else {
80. O[i][j] = D[ii][jj++];
81. stepJ = s;
82. }
83. }
84. stepI = s;
85. }
86. ii++;
87. }
88. }
89. else {
90. O = D;
91. }
92. for (int i = 0; i < f.getN()-1; i++) {
93. neyron.Padding(O);
94. }
95. return neyron.Svertka(F, O);
96. }
98. template<typename T>
99. void CNNLearning<T>::GradDes(const Matrix<T>& X, const Matrix<T>& D, Filter<T>& F) {
100. Matrix<T> Delta = neyron.Svertka(D, X);
101. if ((Delta.getN() != F.getN()) || (Delta.getM() != F.getM())) {
102. throw CNNLearning<T>::CNNLearningExeption("Задана неверная размерность! После свертки размеры матрицы фильтра и матрицы ошибки не совпадают!");
103. }
104. T delt;
105. for (int i = 0; i < Delta.getN(); i++) {
106. for (int j = 0; j < Delta.getM(); j++) {
107. delt = E \* Delta[i][j];
108. if (delt > 1000) {
109. throw CNNLearning<T>::CNNLearningExeption("Слишком большая производная!");
110. }
111. F[i][j]-= delt;
112. }
113. }
114. }
116. template<typename T>
117. inline void CNNLearning<T>::Revers\_Perceptron\_to\_CNN(Matrix<T>& a, const Weights<T>& w)
118. {
119. if ((a.getN() < 0) || (a.getM() < 0)||(a.getN() != w.getN()) || (a.getM() != w.getM())) {
120. throw Base\_Cnn<T>::Base\_CnnExeption("Неверный размер матрицы ошибки!");
121. }
122. for (int i = 0; i < w.getN(); i++) {
123. for (int j = 0; j < w.getM(); j++) {
124. a[i][j] += w.GetD() \* w[i][j];
125. }
126. }
127. }
129. template<typename T>
130. inline Matrix<T> CNNLearning<T>::ReversPooling(const Matrix<T>& D, const int & n\_, const int & m\_)
131. {
132. if ((n\_ < 0) || (m\_ < 0) || (n\_ > D.getN()) || (m\_ > D.getM())) {
133. throw Base\_Cnn<T>::Base\_CnnExeption("Неверный размер ядра!");
134. }
136. Matrix<T> copy(D.getN() \* n\_, D.getM() \* m\_);
138. for (int i = 0; i < D.getN(); i++) {
139. for (int j = 0; j < D.getM(); j++) {
140. for (int ii = i \* n\_; ii < i\*n\_ + n\_; ii++) {
141. for (int jj = j \* m\_; jj < j\*m\_ + m\_; jj++) {
142. copy[ii][jj] = D[i][j];
143. }
144. }
145. }
146. }
147. return copy;
148. }

151. template<typename T>
152. inline CNNLearning<T>::~CNNLearning()
153. {
154. }

CNNLearns.h

1. #pragma once
2. #include "CNNLearning.h"
3. #define D\_CNNLeaning CNNLearning<double>
4. #define I\_CNNLeaning CNNLearning<int>

CNNs.h

1. #pragma once
2. #include "NeyronCnn.h"
3. #include "CNNLearns.h"
4. #include "Filter.h"
5. #define D\_NeyronCnn NeyronСnn<double>
6. #define I\_NeyronCnn NeyronСnn<int>

Перцептрон

Base\_Perceptron.h

1. #pragma once
2. #include "Weights.h"
3. #include "Func.h"
4. #include <vector>

7. template <typename T, typename Y>
8. class Base\_Perceptron
9. {
10. public:
11. // Конструкторы ----------------------------------------------------------
12. Base\_Perceptron();
13. Base\_Perceptron(const Base\_Perceptron& copy) = delete; // Запрет копирования
15. // Методы класса ---------------------------------------------------------
16. // Операция суммированию произведений входов на веса нейрона
17. virtual T Summator(const Matrix<T>& a, const Weights<T>& w);
18. virtual T Summator(std::vector<T> a, const std::vector<T>& w);
20. // Функция активации нейрона
21. virtual Y FunkActiv(const T&, Func<T,Y>&) = 0;
23. // Перегрузка операторов -------------------------------------------------
24. Base\_Perceptron& operator= (const Base\_Perceptron& copy) = delete; // Запрет копирования
26. // Деструктор ------------------------------------------------------------
27. virtual ~Base\_Perceptron();
29. // Класс исключения ------------------------------------------------------
30. class NeyronPerceptronExeption : public std::runtime\_error {
31. public:
32. NeyronPerceptronExeption(std::string str) : std::runtime\_error(str) {};
33. ~NeyronPerceptronExeption() {};
34. };
35. };
37. template <typename T, typename Y>
38. Base\_Perceptron<T, Y>::Base\_Perceptron()
39. {
40. }
42. template <typename T, typename Y>
43. T Base\_Perceptron<T, Y>::Summator(const Matrix<T> & a, const Weights<T> & w)
44. {
45. if ((a.getN() != w.getN()) || (a.getM() != w.getM())) {
46. throw Base\_Perceptron<T, Y>::NeyronPerceptronExeption("Несовпадение размера матрицы весов и размера матрицы входных сигналов!");
47. }
48. T sum = 0;
49. for (int i = 0; i < a.getN(); i++) {
50. for (int j = 0; j < a.getM(); j++) {
51. sum += a[i][j] \* w[i][j];
52. }
53. }
54. sum += w.GetWBias();
55. return sum;
56. }
58. template <typename T, typename Y>
59. inline T Base\_Perceptron<T, Y>::Summator(std::vector<T> a, const std::vector<T>& w)
60. {
61. if (a.size() != w.size()) {
62. throw Base\_Perceptron<T,Y>::NeyronPerceptronExeption("Несовпадение размера матрицы весов и размера матрицы входных сигналов!");
63. }
64. T sum = 0;
65. for (int i = 0; i < a.size(); i++) {
66. sum += a[i] \* w[i];
67. }
68. return sum;
69. }
71. template <typename T, typename Y>
72. Base\_Perceptron<T,Y>::~Base\_Perceptron()
73. {
74. }

NeyronPerceptron.h

1. #pragma once
2. #include "Base\_Perceptron.h"
4. template <typename T, typename Y>
5. class NeyronPerceptron : public Base\_Perceptron<T, Y>
6. {
7. public:
8. // Конструкторы ----------------------------------------------------------
9. NeyronPerceptron();
10. NeyronPerceptron(const NeyronPerceptron& copy) = delete; // Запрет копирования
12. // Методы класса ---------------------------------------------------------
13. // Функция активации нейрона
14. Y FunkActiv(const T& e, Func<T,Y>& f);
16. // Перегрузка операторов -------------------------------------------------
17. NeyronPerceptron& operator= (const NeyronPerceptron& copy) = delete; // Запрет копирования
19. // Деструктор ------------------------------------------------------------
20. ~NeyronPerceptron();
21. };
23. template <typename T, typename Y>
24. NeyronPerceptron<T,Y>::NeyronPerceptron() : Base\_Perceptron<T,Y>()
25. {
26. }
28. template<typename T, typename Y>
29. inline Y NeyronPerceptron<T, Y>::FunkActiv(const T & e, ::Func<T,Y>& f)
30. {
31. return f(e);
32. }
34. template <typename T, typename Y>
35. NeyronPerceptron<T, Y>::~NeyronPerceptron()
36. {
37. }

PerceptronLearning.h

1. #pragma once
2. #include "Weights.h"
3. #include "NeyronPerceptron.h"
4. #include <cstdlib>
6. template<typename T, typename Y>
7. class PerceptronLearning
8. {
9. public:
10. // Конструкторы ----------------------------------------------------------
11. PerceptronLearning(); // По умолчанию
12. PerceptronLearning(const double& E\_); // Инициализатор
13. PerceptronLearning(const NeyronPerceptron<T,Y>& copy) = delete; // Запрет копирования
15. // Методы класса ---------------------------------------------------------
16. // Обучение однослойного перцептрона методом обратного распространения ошибки
17. void WTSimplePerceptron(const Y& a, const Y& y, Weights<T>& w, const Matrix<T>& in);
19. // Метод обратного распространения ошибки
20. static void BackPropagation(Matrix<Weights<T>>& w, const Weights<T>& y);
21. static void BackPropagation(Matrix<Weights<T>>& w, const Matrix<Weights<T>>& y);
23. // Метод градиентного спуска
24. void GradDes(Weights<T>& w, Matrix<T>& in, Func<T, Y>& F, const T& x);
26. // Метод вычисления средней квадратичной ошибки
27. static Y RMS\_error(const Y\* a, const Y\* y, const int& lenth);
29. // Метод вычисления ошибки выходного слоя
30. static Y PartDOutLay(const Y& a, const Y& y);
32. // Метод стягивания весов
33. void retract(Matrix<Weights<T>>& weights,const int& decs);
34. void retract(Weights<T>& weights, const int& decs);
36. // Тасование последовательности
37. void shuffle(int\* arr, const int& lenth);
39. // Метод получения доступа к кофиценту обучения
40. double& getE() { return E; };
42. // Перегрузка операторов -------------------------------------------------
43. PerceptronLearning& operator= (const PerceptronLearning& copy) = delete; // Запрет копирования
44. friend std::ostream& operator<<(std::ostream& out, const PerceptronLearning& mat) = delete; // Запрет вывода в поток
45. friend std::istream& operator>>(std::istream& out, PerceptronLearning& mat) = delete; // Запрет считывания из потока
47. // Деструктор ------------------------------------------------------------
48. ~PerceptronLearning();
50. // Класс исключения ------------------------------------------------------
51. class LearningExeption : public std::runtime\_error {
52. public:
53. LearningExeption(std::string str) : std::runtime\_error(str) {};
54. ~LearningExeption() {};
55. };
56. protected:
57. // Поля класса ----------------------------------
58. double E; // Кофицент обучения
59. };
61. template<typename T, typename Y>
62. inline PerceptronLearning<T,Y>::PerceptronLearning() : E(1)
63. {
64. }
66. template<typename T, typename Y>
67. inline PerceptronLearning<T, Y>::PerceptronLearning(const double & E\_) : E(E\_)
68. {
69. }
71. template<typename T, typename Y>
72. inline void PerceptronLearning<T, Y>::WTSimplePerceptron(const Y & a, const Y & y, Weights<T> & w, const Matrix<T>& in)
73. {
74. if ((w.getN() != in.getN()) || (w.getM() != in.getM())) {
75. throw LearningExeption("Несовпадение размеров входной матрицы и матрицы весов!");
76. }
77. T delta = a - y;
78. T ii = 0;
79. if (delta == 0) {
80. return;
81. }
82. for (int i = 0; i < w.getN(); i++) {
83. for (int j = 0; j < w.getM(); j++) {
84. ii = w[i][j] + E \* delta\*in[i][j];
85. w[i][j] = ii;
86. }
87. }
88. w.GetWBias() += E \* delta;
89. }
91. template<typename T, typename Y>
92. inline void PerceptronLearning<T, Y>::BackPropagation(Matrix<Weights<T>>& w, const Weights<T>& y)
93. {
94. for (int i = 0; i < y.getN(); i++) {
95. for (int j = 0; j < y.getM(); j++) {
96. w[i][j].GetD() += (y[i][j] \* y.GetD());
97. }
98. }
99. }
101. template<typename T, typename Y>
102. inline void PerceptronLearning<T, Y>::BackPropagation(Matrix<Weights<T>>& w, const Matrix<Weights<T>>& y)
103. {
104. for (int o = 0; o < y.getN(); o++) {
105. for (int u = 0; u < y.getM(); u++) {
106. for (int i = 0; i < y[o][u].getN(); i++) {
107. for (int j = 0; j < y[o][u].getM(); j++) {
108. w[i][j].GetD() += (y[o][u][i][j] \* y.GetD());
109. }
110. }
111. }
112. }
113. }
115. template<typename T, typename Y>
116. inline void PerceptronLearning<T, Y>::GradDes(Weights<T>& w, Matrix<T>& in, Func<T, Y>& F, const T& x)
117. {
118. if ((w.getN() != in.getN()) || (w.getM() != in.getM())) {
119. throw LearningExeption("Несовпадение размеров входной матрицы и матрицы весов!");
120. }
122. for (int i = 0; i < w.getN(); i++) {
123. for (int j = 0; j < w.getM(); j++) {
124. w[i][j] -= (w.GetD() \* E \* F(x) \* in[i][j]);
125. }
126. }
127. w.GetWBias() -= E \* F(x) \* w.GetD();
128. }

131. template<typename T, typename Y>
132. inline Y PerceptronLearning<T, Y>::RMS\_error(const Y \* a, const Y \* y, const int & lenth)
133. {
134. Y err = 0;
135. for (int i = 0; i < lenth; i++) {
136. err += (a[i] - y[i])\*(a[i] - y[i]);
137. }
138. err /= 2;
139. return err;
140. }
142. template<typename T, typename Y>
143. inline Y PerceptronLearning<T, Y>::PartDOutLay(const Y & a, const Y & y)
144. {
145. return -2\*(a - y);
146. }
148. template<typename T, typename Y>
149. inline void PerceptronLearning<T, Y>::retract(Matrix<Weights<T>>& weights, const int & decs)
150. {
151. int d = 1;
152. for (int i = 0; i < decs; i++) {
153. d \*= 0.1;
154. }
155. for (int i = 0; i < weights.getN(); i++) {
156. for (int j = 0; j < weights.getM(); j++) {
157. for (int k = 0; k < weights[i][j].getN(); k++) {
158. for (int y = 0; y < weights[i][j].getM(); y++) {
159. if (weights[i][j][k][y] > 0) {
160. weights[i][j][k][y] -= d;
161. }
162. else {
163. weights[i][j][k][y] += d;
164. }
165. }
166. }
167. }
168. }
169. }
171. template<typename T, typename Y>
172. inline void PerceptronLearning<T, Y>::retract(Weights<T>& weights, const int & decs)
173. {
174. int d = 1;
175. for (int i = 0; i < decs; i++) {
176. d \*= 0.1;
177. }
178. for (int k = 0; k < weights.getN(); k++) {
179. for (int y = 0; y < weights.getM(); y++) {
180. if (weights[k][y] > 0) {
181. weights[k][y] -= d;
182. }
183. else {
184. weights[k][y] += d;
185. }
186. }
187. }
188. }


192. template<typename T, typename Y>
193. inline void PerceptronLearning<T, Y>::shuffle(int \* arr, const int & lenth)
194. {
195. srand(time(0));
196. int j = 0;
197. int tmp = 0;
198. for (int i = 0; i < lenth; i++) {
199. j = ((double)rand()/INT\_MAX\*lenth);
200. tmp = arr[i];
201. arr[i] = arr[j];
202. arr[j] = tmp;
203. }
204. }
206. template<typename T, typename Y>
207. inline PerceptronLearning<T, Y>::~PerceptronLearning()
208. {
209. }

PLearns.h

1. #pragma once
2. #include "PerceptronLearning.h"
3. #define DD\_Leaning PerceptronLearning<double, double>
4. #define ID\_Leaning PerceptronLearning<int, double>
5. #define DI\_Leaning PerceptronLearning<double, int>
6. #define II\_Leaning PerceptronLearning<int, int>

Perceptrons.h

1. #pragma once
2. #include "NeyronPerceptron.h"
3. #include "Functors.h"
4. #include "PLearns.h"
5. #include "Weights.h"
7. #define DD\_Perceptron NeyronPerceptron<double, double>
8. #define ID\_Perceptron NeyronPerceptron<int, double>
9. #define DI\_Perceptron NeyronPerceptron<double, int>
10. #define II\_Perceptron NeyronPerceptron<int, int>

Вспомогательные классы

Matrix.h

1. #pragma once
2. #include <string>
4. template <typename T>
5. class Matrix
6. {
7. public:
8. // Конструкторы ---------------------------------
9. Matrix(); // Конструктор по умолчанию -----------
10. Matrix(T\*\* arr\_, const int& i, const int& j); // Конструктор инициализатор
11. Matrix(T\* arr\_, const int& i, const int& j); // Конструктор инициализатор
12. Matrix(const int& i, const int& j); // Конструктор инициализатор (создает матрицу заданного размера заполненную 0)
13. Matrix(const Matrix<T>& copy); // Конструктор копирования
15. // Методы класса --------------------------------
16. // Получение количества строк
17. int getN() const
18. {
19. return n;
20. }
22. // Получение колисчества столбцов
23. int getM() const
24. {
25. return m;
26. }
28. // Поиск максимума в массиве того же типа
29. static T Max(T\*\* arr\_, const int& n\_, const int& m\_);
31. // Поиск максимума в матрице
32. T Max() const;
34. // Получение копии матрицы в виде массива
35. T\*\* getCopy();
37. // Заполнение матрицы заданным значением
38. void Fill(const T& a);
40. // Получение подматрицы
41. Matrix<T> getPodmatrix(const int& poz\_n\_, const int& poz\_m\_, const int& n\_, const int& m\_) const;
43. // Перегрузки операторов ------------------------
44. Matrix<T>& operator= (const Matrix<T>& copy); // Оператор присваивания
45. Matrix<T> operator+ (const Matrix<T>& mat) const; // Оператор суммы
46. Matrix<T> operator\* (const Matrix<T>& mat) const; // Оператор произведения
47. Matrix<T> operator\* (const int k) const; // Оператор произведения на число
48. template <typename T1> friend Matrix<T1> operator\* (const int k, const Matrix<T1>& mat); // Оператор произведения на число
49. template <typename T1> friend std::ostream& operator<< (std::ostream& out, const Matrix<T1>& mat); // Оператор вывод матрицы в поток
50. template <typename T1> friend std::istream& operator>> (std::istream& out, Matrix<T1>& mat); // Оператор чтение матрицы из потока
51. std::shared\_ptr<T[]> operator[] (int index); // Оператор индексации
52. const std::shared\_ptr<T[]> operator[] (int index) const; // Оператор индексации константы
53. bool operator==(const Matrix<T>& mat) const; // Оператор сравнения матриц

56. // Деструктор -----------------------------------
57. virtual ~Matrix();
59. // Класс исключений ----------------------------
60. class MatrixExeption : public std::runtime\_error
61. {
62. public:
63. MatrixExeption(std::string s) : std::runtime\_error(s) {}
64. ~MatrixExeption() {}
65. };
66. protected:
68. // Поля класса ----------------------------------
69. int n, // Количество строк в матрице
70. m; // Количество столбцов с матрице
71. std::shared\_ptr < std::shared\_ptr<T []>[] > arr; // Матрица
73. // Скрытые матоды класса ------------------------
74. void initMat(); // Выделение памяти для матрицы
75. void isInRange(int index) const; // Проверяет, находится ли индекс в допустимых границах
76. };

79. // Реализация ---------------------------------------
80. template<typename T>
81. Matrix<T>::Matrix() : n(0), m(0)
82. {
83. arr = nullptr;
84. }
86. template<typename T>
87. Matrix<T>::Matrix(T\*\* arr\_, const int& i, const int& j) : n(i), m(j)
88. {
89. if ((n < 0) || (m < 0)) {
90. throw Matrix::MatrixExeption("Неверный размер матрицы!");
91. }
92. initMat();
93. for (int i = 0; i < n; i++) {
94. for (int j = 0; j < m; j++) {
95. arr[i][j] = arr\_[i][j];
96. }
97. }
98. }
100. template<typename T>
101. Matrix<T>::Matrix(T\* arr\_, const int& i, const int& j) : n(i), m(j)
102. {
103. if ((n < 0) || (m < 0)) {
104. throw Matrix::MatrixExeption("Неверный размер матрицы!");
105. }
106. initMat();
107. for (int i = 0; i < n; i++) {
108. for (int j = 0; j < m; j++) {
109. arr[i][j] = arr\_[i\*m + j];
110. }
111. }
112. }
114. template<typename T>
115. Matrix<T>::Matrix(const int& i, const int& j) : n(i), m(j)
116. {
117. if ((n < 0) || (m < 0)) {
118. throw Matrix::MatrixExeption("Неверный размер матрицы!");
119. }
120. initMat();
121. for (int i = 0; i < n; i++) {
122. for (int j = 0; j < m; j++) {
123. arr[i][j] = T();
124. }
125. }
126. }
128. template<typename T>
129. Matrix<T>::Matrix(const Matrix<T> & copy) : n(copy.n), m(copy.m)
130. {
131. initMat();
132. for (int i = 0; i < n; i++) {
133. for (int j = 0; j < m; j++) {
134. arr[i][j] = copy.arr[i][j];
135. }
136. }
137. }
139. template<typename T>
140. Matrix<T> Matrix<T>::getPodmatrix(const int& poz\_n\_, const int& poz\_m\_, const int& n\_, const int& m\_) const {
141. if ((poz\_n\_ < 0) || (poz\_m\_ < 0)||(poz\_n\_ >= n)||(poz\_m\_ >= m)) {
142. throw Matrix::MatrixExeption("Неверная позиция верхнего левого элемента подматрицы!");
143. }
144. if (((poz\_n\_ + n\_) > n) || ((poz\_m\_ + m\_) > m)) {
145. throw Matrix::MatrixExeption("Подматрица выходит за границы матрицы!");
146. }
148. Matrix<T> rez(n\_, m\_);
150. for (int i = 0; i < n\_; i++) {
151. for (int j = 0; j < m\_; j++) {
152. rez[i][j] = arr[poz\_n\_ + i][poz\_m\_ + j];
153. }
154. }
155. return rez;
156. }
158. template<typename T>
159. T Matrix<T>::Max(T\*\* arr\_, const int& n\_, const int& m\_) {
160. T max = arr\_[0][0];
161. for (int i = 0; i < n\_; i++) {
162. for (int j = 0; j < m\_; j++) {
163. if (arr\_[i][j] > max) {
164. max = arr\_[i][j];
165. }
166. }
167. }
168. return max;
169. }
171. template<typename T>
172. T Matrix<T>::Max() const {
173. T max = arr[0][0];
174. for (int i = 0; i < n; i++) {
175. for (int j = 0; j < m; j++) {
176. if (arr[i][j] > max) {
177. max = arr[i][j];
178. }
179. }
180. }
181. return max;
182. }
184. template<typename T>
185. T\*\* Matrix<T>::getCopy()
186. {
187. T\*\* copy;
188. copy = new T\*[n];
189. for (int i = 0; i < n; i++) {
190. copy[i] = new T[m];
191. }
192. for (int i = 0; i < n; i++) {
193. for (int j = 0; j < m; j++) {
194. copy[i][j] = arr[i][j];
195. }
196. }
197. return copy;
198. }
200. template<typename T>
201. void Matrix<T>::Fill(const T& a) {
202. for (int i = 0; i < n; i++) {
203. for (int j = 0; j < m; j++) {
204. arr[i][j] = a;
205. }
206. }
207. }
209. template<typename T>
210. Matrix<T>& Matrix<T>::operator=(const Matrix<T> & copy)
211. {
212. if (this == &copy) {
213. return \*this;
214. }
215. if ((copy.n > n) || (copy.m > m)) {
216. for (int i = 0; i < n; i++) {
217. arr[i].reset();
218. }
219. arr.reset();
220. n = copy.n;
221. m = copy.m;
222. initMat();
223. }
224. else {
225. n = copy.n;
226. m = copy.m;
227. }
229. for (int i = 0; i < n; i++) {
230. for (int j = 0; j < m; j++) {
231. arr[i][j] = copy.arr[i][j];
232. }
233. }
234. return \*this;
235. }
237. template<typename T>
238. Matrix<T> Matrix<T>::operator+(const Matrix<T> & mat) const
239. {
240. Matrix<T> tmp(\*this);
241. if ((n != mat.n) || (m != mat.m)) {
242. throw MatrixExeption("Невозможно выполнить сложение матриц разного размера");
243. }
244. for (int i = 0; i < n; i++) {
245. for (int j = 0; j < m; j++) {
246. tmp[i][j] += mat.arr[i][j];
247. }
248. }
249. return tmp;
250. }
252. template<typename T>
253. Matrix<T> Matrix<T>::operator\*(const Matrix<T> & mat) const
254. {
255. if (m != mat.n) {
256. throw MatrixExeption("Невозможно выполнить умножение матриц с несовпадающим количеством столбцов в первой и строк во второй");
257. }
258. Matrix<T> tmp(n, mat.m);
259. for (int i = 0; i < n; i++) {
260. for (int j = 0; j < mat.m; j++) {
261. for (int o = 0; o < m; o++) {
262. tmp[i][j] += (arr[i][o] \* mat.arr[o][j]);
263. }
264. }
265. }
266. return tmp;
267. }
269. template<typename T>
270. Matrix<T> Matrix<T>::operator\*(const int k) const
271. {
272. Matrix<T> tmp(\*this);
273. for (int i = 0; i < n; i++) {
274. for (int j = 0; j < m; j++) {
275. tmp[i][j] \*= k;
276. }
277. }
278. return tmp;
279. }
281. template<typename T>
282. std::shared\_ptr<T[]> Matrix<T>::operator[](int index)
283. {
284. isInRange(index);
285. return arr[index];
286. }
288. template<typename T>
289. const std::shared\_ptr<T[]> Matrix<T>::operator[](int index) const
290. {
291. isInRange(index);
292. return arr[index];
293. }
295. template<typename T>
296. bool Matrix<T>::operator==(const Matrix<T> & mat) const
297. {
298. if ((n != mat.n) || (m != mat.m)) {
299. return false;
300. }
301. for (int i = 0; i < n; i++) {
302. for (int j = 0; j < m; j++) {
303. if (arr[i][j] != mat.arr[i][j]) {
304. return false;
305. }
306. }
307. }
308. return true;
309. }
311. template<typename T>
312. Matrix<T>::~Matrix()
313. {
314. for (int i = 0; i < n; i++) {
315. arr[i].reset();
316. arr[i] = nullptr;
317. }
318. arr.reset();
319. arr = nullptr;
320. }
322. template<typename T>
323. void Matrix<T>::initMat()
324. {
325. arr.reset(new std::shared\_ptr<T []>[n]);
326. for (int i = 0; i < n; i++) {
327. arr[i].reset(new T[m]);
328. }
329. }
331. template<typename T>
332. void Matrix<T>::isInRange(int index) const
333. {
334. if ((index > n) || (index < 0)) {
335. throw MatrixExeption("Индекс выходит за размер матрицы!");
336. }
337. }
339. template<typename T>
340. Matrix<T> operator\*(const int k, const Matrix<T> & mat)
341. {
342. Matrix<T> tmp(mat);
343. for (int i = 0; i < mat.n; i++) {
344. for (int j = 0; j < mat.m; j++) {
345. tmp[i][j] \*= k;
346. }
347. }
348. return tmp;
349. }
351. template<typename T>
352. std::ostream& operator<<(std::ostream& out, const Matrix<T> & mat)
353. {
354. out << mat.n << ' ' << mat.m << std::endl; // Для совместимости с вводом из файла
356. for (int i = 0; i < mat.n; i++) {
357. for (int j = 0; j < mat.m; j++) {
358. out << mat.arr[i][j] << ' ';
359. }
360. out << std::endl;
361. }
362. return out;
363. }
365. template<typename T>
366. std::istream& operator>>(std::istream & in, Matrix<T> & mat)
367. {
368. in >> mat.n;
369. in >> mat.m;
370. if ((mat.n < 0) || (mat.m < 0)) {
371. throw Matrix<T>::MatrixExeption("Неверный размер матрицы!");
372. }
373. mat.initMat();
374. for (int i = 0; i < mat.n; i++) {
375. for (int j = 0; j < mat.m; j++) {
376. in >> mat.arr[i][j];
377. }
378. }
379. return in;
380. }

Filter.h

1. #pragma once
2. #include "Matrix.h"
3. #include <iomanip>
5. template <typename T>
6. class Filter : public Matrix<T>
7. {
8. public:
9. // Конструкторы ----------------------------------------------------------
10. Filter(); // По умолчанию
11. Filter(const int& i\_, const int& j\_); // Инициализатор (нулевая матрица)
12. Filter(T\*\* arr\_, const int& i\_, const int& j\_); // Инициализатор
13. Filter(const Filter<T>& copy); // Копирования
15. // Методы класса ---------------------------------------------------------
16. // Поворот фильтра на 180
17. Filter<T> roate\_180() const;
19. // Вывод фильтра на консоль в красивом виде
20. void Out() const;
22. // Перегрузки операторов ------------------------
23. Filter<T>& operator= (const Filter<T>& copy); // Оператор присваивания
25. // Деструктор ------------------------------------------------------------
26. ~Filter<T>();
27. };
29. template <typename T>
30. Filter<T>::Filter() : Matrix<T>()
31. {
32. }
34. template <typename T>
35. Filter<T>::Filter(const int & i\_, const int & j\_) : Matrix<T>(i\_, j\_)
36. {
37. }
39. template <typename T>
40. Filter<T>::Filter(T \*\* arr\_, const int & i\_, const int & j\_) : Matrix<T>(arr\_, i\_, j\_)
41. {
42. }
44. template <typename T>
45. Filter<T>::Filter(const Filter<T> & copy) : Matrix<T>(copy)
46. {
47. }
49. template<typename T>
50. inline Filter<T> Filter<T>::roate\_180() const
51. {
52. Filter<T> F(this->n, this->m);
53. for (int i = this->n-1; i >= 0; i--) {
54. for (int j = this->m-1; j >= 0; j--) {
55. F[i][j] = this->arr[this->n-1 - i][this->m-1 - j];
56. }
57. }
58. return F;
59. }
61. template<typename T>
62. inline Filter<T>& Filter<T>::operator=(const Filter<T>& copy)
63. {
64. if (this == &copy) {
65. return \*this;
66. }
67. if ((copy.n > this->n) || (copy.m > this->m)) {
68. for (int i = 0; i < this->n; i++) {
69. this->arr[i].reset();
70. }
71. this->arr.reset();
72. this->n = copy.n;
73. this->m = copy.m;
74. this->initMat();
75. }
76. else {
77. this->n = copy.n;
78. this->m = copy.m;
79. }
81. for (int i = 0; i < this->n; i++) {
82. for (int j = 0; j < this->m; j++) {
83. this->arr[i][j] = copy.arr[i][j];
84. }
85. }
86. return \*this;
87. }
89. template <typename T>
90. Filter<T>::~Filter()
91. {
92. }
94. template<typename T>
95. inline void Filter<T>::Out() const
96. {
97. for (int i = 0; i < this->n; i++) {
98. for (int j = 0; j < this->m; j++) {
99. std::cout << this->arr[i][j] << " ";
100. }
101. std::cout << std::endl;
102. }
103. }
105. inline void Filter<int>::Out() const
106. {
107. using std::cout;
108. int max = this->Max();
109. int k = 2;
110. while (max > 0) {
111. k++;
112. max = max / 10;
113. }
114. for (int i = 0; i < this->n; i++) {
115. for (int j = 0; j < this->m; j++) {
116. cout << std::setw(k) << this->arr[i][j];
117. }
118. cout << std::endl;
119. }
120. }
122. inline void Filter<double>::Out() const
123. {
124. using std::cout;
125. int max = (int)this->Max();
126. int k = 2;
127. while (max > 0) {
128. k++;
129. max = max / 10;
130. }
131. for (int i = 0; i < this->n; i++) {
132. for (int j = 0; j < this->m; j++) {
133. cout << std::setw(k + 5) << std::setprecision(2) << this->arr[i][j];
134. }
135. cout << std::endl;
136. }
137. }

Weights.h

1. #pragma once
2. #include "Matrix.h"
3. #include <iostream>
4. #include <iomanip>
6. template<typename T>
7. class Weights : public Matrix<T>
8. {
9. public:
10. // Конструкторы ----------------------------------------------------------
11. Weights(); // По умолчанию
12. Weights(const int& i\_, const int& j\_, const int& wbisas\_ = 0); // Инициализатор (нулевая матрица)
13. Weights(T\*\* arr\_, const int& i\_, const int& j\_, const int& wbisas\_ = 0); // Инициализатор
14. Weights(const Weights<T>& copy); // Копирования
16. // Методы класса ---------------------------------------------------------
17. // Вывод весов на консоль в красивом виде
18. void Out();
20. // Получение доступа к d
21. T& GetD() { return d; };
22. const T& GetD() const { return d; };
24. // Получение доступа к wbisas
25. T& GetWBias() { return wbias; };
26. const T& GetWBias() const { return wbias; };
28. // Перегрузки операторов ------------------------
29. Weights<T>& operator= (const Weights<T>& copy); // Оператор присваивания
30. template <typename T1> friend std::ostream& operator<< (std::ostream& out, const Weights<T1>& mat); // Оператор вывод матрицы в поток
31. template <typename T1> friend std::istream& operator>> (std::istream& in, Weights<T1>& mat); // Оператор чтение матрицы из потока

34. // Деструктор ------------------------------------------------------------
35. ~Weights();
36. private:
37. // Величина производной функции ошибки
38. T d;
40. // Вес нейрона сдвига
41. T wbias;
43. };
45. template<typename T>
46. Weights<T>::Weights() : Matrix<T>(), d(0), wbias(0)
47. {
48. }
50. template<typename T>
51. Weights<T>::Weights(const int & i\_, const int & j\_, const int& wbisas\_) : Matrix<T>(i\_, j\_), d(0), wbias(wbisas\_)
52. {
53. }
55. template<typename T>
56. Weights<T>::Weights(T \*\* arr\_, const int & i\_, const int & j\_, const int& wbisas\_) : Matrix<T>(arr\_, i\_, j\_), d(0), wbias(0)
57. {
58. }
60. template<typename T>
61. Weights<T>::Weights(const Weights<T> & copy) : Matrix<T>(copy), d(copy.GetD()), wbias(copy.GetWBias())
62. {
63. }
65. template<typename T>
66. inline void Weights<T>::Out()
67. {
68. for (int i = 0; i < this->n; i++) {
69. for(int j = 0; j < this->m; j++){
70. std::cout << this->arr[i][j] << " ";
71. }
72. std::cout << std::endl;
73. }
74. if (wbias != 0) {
75. std::cout << wbias << std::endl;
76. }
77. }
79. inline void Weights<int>::Out()
80. {
81. using std::cout;
82. int max = this->Max();
83. int k = 2;
84. while (max > 0) {
85. k++;
86. max = max / 10;
87. }
88. for (int i = 0; i < this->n; i++) {
89. for (int j = 0; j < this->m; j++) {
90. cout << std::setw(k) << this->arr[i][j];
91. }
92. cout << std::endl;
93. }
94. if (wbias != 0) {
95. std::cout << std::setw(k) << wbias << std::endl;
96. }
97. }
99. inline void Weights<double>::Out()
100. {
101. using std::cout;
102. int max = (int)this->Max();
103. int k = 2;
104. while (max > 0) {
105. k++;
106. max = max / 10;
107. }
108. for (int i = 0; i < this->n; i++) {
109. for (int j = 0; j < this->m; j++) {
110. cout << std::setw(k+5) << std::setprecision(2)<< this->arr[i][j];
111. }
112. cout << std::endl;
113. }
114. if (wbias != 0) {
115. std::cout << std::setw(k + 5) << std::setprecision(4) << wbias << std::endl;
116. }
117. }
119. template<typename T>
120. inline Weights<T>& Weights<T>::operator=(const Weights<T>& copy)
121. {
122. if (this == &copy) {
123. return \*this;
124. }
125. if ((copy.n > this->n) || (copy.m > this->m)) {
126. for (int i = 0; i < this->n; i++) {
127. this->arr[i].reset();
128. }
129. this->arr.reset();
130. this->n = copy.n;
131. this->m = copy.m;
132. this->initMat();
133. }
134. else {
135. this->n = copy.n;
136. this->m = copy.m;
137. }
139. for (int i = 0; i < this->n; i++) {
140. for (int j = 0; j < this->m; j++) {
141. this->arr[i][j] = copy.arr[i][j];
142. }
143. }
144. d = copy.d;
145. wbias = copy.wbias;
146. return \*this;
147. }
149. template<typename T>
150. Weights<T>::~Weights()
151. {
152. }
154. template<typename T1>
155. inline std::ostream & operator<<(std::ostream & out, const Weights<T1>& mat)
156. {
157. out << (Matrix<T1>) mat;
158. out << mat.d << ' ' << mat.wbias << std::endl;
159. return out;
160. }
162. template<typename T1>
163. inline std::istream & operator>>(std::istream & in, Weights<T1>& mat)
164. {
165. in >> ((Matrix<T1>&) mat);
166. in >> mat.d;
167. in >> mat.wbias;
168. return in;
169. }

Func.h

1. #pragma once
3. template<typename T, typename Y>
4. class Func {
5. public:
6. Func() {};
7. virtual Y operator()(const T&) = 0;
8. virtual ~Func() {};
9. };

Functors.h

1. #pragma once
2. #include "Func.h"
3. #define DD\_Func Func <double, double>
4. #define ID\_Func Func <int, double>
5. #define DI\_Func Func <double, int>
6. #define II\_Func Func <int, int>

«Список используемых источников и литературы»

1. <https://ru.wikipedia.org/wiki/Искусственная_нейронная_сеть>
2. <https://ru.wikipedia.org/wiki/Нейронная_сеть>
3. <https://geektimes.ru/post/277088/>
4. <http://neuralnet.info/глава-1-введение/>
5. <http://neuralnet.info/глава-2-основы-инс/>
6. http://neuralnet.info/глава-3-персептроны/
7. <http://www.pvsm.ru/algoritmy/198448>
8. <https://habr.com/ru/post/348000/>
9. <https://habr.com/ru/post/348028/>
10. <https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/>
11. <https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199>